Support for Managing Design-Time Decisions

Alexander Egyed, *Member*, *IEEE*, and David S. Wile

Abstract—The desirability of maintaining multiple stakeholders' interests during the software design process argues for leaving choices undecided as long as possible. Yet, any form of underspecification, either missing information or undecided choices, must be resolved before automated analysis tools can be used. This paper demonstrates how *Constraint Satisfaction Problem Solution Techniques* (CSTs) can be used to automatically reduce the space of choices for ambiguities by incorporating the local effects of constraints, ultimately with more global consequences. As constraints typical of those encountered during the software design process, we use UML consistency and well-formedness rules. It is somewhat surprising that CSTs are suitable for the software modeling domain since the constraints may relate many ambiguities during their evaluation, encountering a well-known problem with CSTs called the k-consistency problem. This paper demonstrates that our CST-based approach is computationally scalable and effective—as evidenced by empirical experiments based on dozens of industrial models.

Index Terms—UML, design choices, consistency checking, design alternatives, choice elimination.

1 Introduction

Design choices arise naturally during software development in decisions involving: conflicting opinions among two or more stakeholders [8], alternative implementations, partially known or understood goals (i.e., requirements), imprecise model semantics [26], partially known trace dependencies among design artifacts [9], and many others. The decisions made for these design choices can have the most telling consequences downstream in the development process [2], especially, if they are made during the early stages of the software life cycle.

If the ambiguities these decisions must ultimately resolve remain completely inside the designer's head—indeed if the choices available are not even characterized—one cannot effectively "backtrack" in the design process when dead ends are reached. We focus on supporting the maintenance of designs in which such ambiguities are formally identified as choices in the design space. Yet, ***we do not believe that it is possible for an "automated assistant" to make good or even adequate design decisions, because designers, in part, base their decisions on information too imprecise to be modeled***. In fact, we are unable to know the actual constraints that make any design choice the best—or a "good enough"—choice because decision factors remain locked in the head of the designer. Many of these constraints may not even be capable of being formulated formally, reflecting matters of taste, style, experience, personal preferences, discussions, expected evolution, "gut" feelings, etc. Yet, we do see a role for an automated assistant in telling the developer ***what choices are infeasible based on the current level of knowledge (i.e., the model) and currently known considerations (i.e., the known constraints on the model)***. Such an assistant would tell the designer what choices to ignore, as they are guaranteed to violate known design constraints, and it would leave it up to the designer to decide on the remaining choices in a manner that reflects his or her unmodeled beliefs.

To motivate and support the development of such an assistant, here we describe how to apply *constraint propagation algorithms* used in Constraint Satisfaction Problem (CSP) solution techniques [15] (CST) to automatically reduce the set of infeasible choices in software development activities. Our technology is useful in that it reduces the set of choices based on the known constraints such that it guarantees that all the choices the algorithm eliminates are indeed infeasible—that is, they cannot possibly satisfy the known constraints. Of course, this guarantee is valid even though not all constraints may be known or formalizable (because a provably false constraint will exist when the infeasible choice is eliminated); however, there is no guarantee that all of the remaining choices, the ones that are not labeled as infeasible, are indeed feasible.

The particular environment used to demonstrate the technology is the Unified Modeling Language (UML) [26] support environment, IBM Rational Rose™ [17]. This environment was chosen in part for its rich set of interrelated, alternative representations for software development artifacts. Although application of CST to UML models and to UML consistency and well-formedness rules is fairly straightforward, there are several issues that complicate matters:

1. UML consistency and well-formedness rules access a variable number of model elements (and, thus, a variable number of ambiguities), which makes this problem a k-consistency CSP, known to be NP-hard. Hence, the techniques would not appear to be scalable.
2. It is necessary to treat the UML consistency and well-formedness rules as black-box constraints, relying on a form of *model profiling* to discover the embedded ambiguities. Hence, the algorithm cannot

• *The authors are with Teknowledge Corporation, 4640 Admiralty Way, Suite 1010, Marina del Rey, CA 90292. E-mail: {aegyed, dwile}@teknowledge.com.*

Manuscript received 16 Sept. 2005; revised 19 Apr. 2006; accepted 24 Apr. 2006; published online 24 May 2006.
Recommended for acceptance by B. Cheng.
For information on obtaining reprints of this article, please send e-mail to: tse@computer.org, and reference IEEECS Log Number TSE-0248-0905.

predict ahead of time which model elements and ambiguities will be encountered during evaluation. Many CSP optimizations are based on having this knowledge upfront and, thus, are not applicable.

Other more technical issues will be discussed as the algorithm is explained.

It must be stressed that ***it is not our goal to find a "good" solution for a set of choices and constraints***. For example, CSTs could compute a possible solution that satisfies all known constraints. But, to the designer, this solution would be meaningless because it would not consider the designer's unmodeled or unknown constraints. We thus do not advocate using CSTs for this purpose.

Yet, we do advocate using CSTs for eliminating infeasible choices that the designer should not consider once they are guaranteed to be inconsistent with the choices already made. We know that many design choices are eliminated in the head of the designer without ever formulating them [14]. Yet, we also know that many projects fail because they did not adequately consider the feasibility of these choices. This paper demonstrates how to adapt CSTs to the UML ambiguity problem so that they avoid the complications discussed above. Below, we introduce our approach formally and illustrate it on an example. We also present empirical evidence that, with reasonable assumptions about design constraints, our approach is sufficiently fast to be used in a normal software development process. We demonstrate that its complexity is the size of the model times the number of constraints. And, we present empirical evidence based on 27 UML models with tens of thousands of model elements in total. Since CSTs are incapable of identifying all infeasible choices, we also present empirical evidence that shows that our approach removes most of the infeasible choices and it is near optimal in the UML domain. Given the wide use of UML, our approach is clearly relevant to a large community, but it must be noted that our approach will work similarly well with other design languages (e.g., DFD diagrams) or architecture description languages, where localized language constraints tend to relate small numbers of design artifacts.

2 Related Work

The problem we are trying to solve is a specific formulation of the constraint satisfaction problem (CSP) [15], tailored to constraints and choices commonly found in software engineering designs. A constraint-based problem is comprised of a set of variables (each with its own domain of values that can be used to fill the variables) and a set of constraints across one, two, or more of these variables. CSP solution techniques (CSTs) either compute a set of feasible values for all variables that satisfy all the constraints or fail, when no such solution exists.

CSTs typically do both *constraint propagation* and *domain reduction*. Constraint propagation eliminates many infeasible choices quickly. We make use of constraint propagation in this paper. However, constraint propagation cannot guarantee that the remaining choices, the ones that are not labeled as infeasible, are indeed feasible. Domain reduction then decides on these remaining choices. It does so through a depth-first search and backtracking when a dead end is reached. We do not make use of domain reduction here because it is expensive and because we are near optimal in the UML domain without it (see evaluation). This work is, in essence, analogous to the arc consistency problem (AC3 in particular) [20]. Beyond the basic algorithm, the most common optimizations are to:

- Limit constraints to one or two variables only (node and arc consistency); but UML constraints are typically k-consistency rules involving a variable number of ambiguities.
- Bound the domain of choices for continuous (or large) domains, treating multiple choices as a single entity. UML choice sets are typically small and discrete, so this is of no use here.
- Use semantic knowledge of the problem domain to optimize for it. This is typically done to minimize backtracking, which our approach avoids altogether.

It cannot be predicted ahead of time what model elements (including ambiguities) will be encountered during the evaluation of UML consistency constraints (i.e., they are black-box constraints). Moreover, depending on what choices are substituted for ambiguities, UML consistency constraints may encounter different ambiguities thereafter. Basic CSP does not consider such variability, but some variations begin to approach the issue. Dynamic CSP [21] defines required and optional variables and constraints that define when optional variables are required. This variation is not applicable to UML because all ambiguities are expected to be always active (and because there is no notion of a constraint that activates an ambiguity). Composite CSP [27] is closer to our problem in that hierarchies of variables are defined. Depending on the value chosen for one ambiguity, other variables and choices appear. However, this concept is also not directly applicable to the UML domain, where this hierarchy is not known a priori.

There are several areas for which specialized ambiguity resolution schemes have been invented. For example, type inference has matured to the point where it is a standard tool in modern compilers, e.g., for Java, C++, and Visual Basic. Perhaps the most advanced of these is the Hindley-Milner type algorithm used in the Haskell language implementations [16]. The unification algorithm used in logic programming languages is similarly well-developed [4], [18]. Other specialized applications include C++ template resolution and polymorphism resolution.

Our approach also relates to truth maintenance systems (TMS) [7] that maintain support relations for facts that make constraints hold. The obvious similarity is that TMSs build a dependency representation similar to ours for maintaining choice dependencies. Again, as with other exhaustive search systems, they (generally) use dependency-directed backtracking when confronted with new facts. The Assumption-based TMSs (ATMS) [19] do not backtrack and their hierarchy for support relationships is closely related to what we build dynamically. However, ATMSs require the existence of dependencies in advance. No such dependencies are known in model-based software development and, thus, ATMSs are not readily applicable to solving problems in this domain. Our work resembles a lightweight version of some of the ATMS concepts, but it identifies dependencies dynamically.

Model checking applications seem as though they should be related as well [12], but we have been unable to make an appropriate reformulation of our problem into it. One distinction from that work is that model checkers are seeking a single inconsistency; our work seeks to eliminate all inconsistent choices automatically. But, a more fundamental distinction seems to reside in the mechanisms for implementing model checkers (and other forms of abstract interpretation as well): They rely on having an abstract representation of the constraint that can be manipulated and reasoned about. For example, CTL labels states with subformulas of the property to be verified [3]. Our approach is independent of the constraint formulation; it simply relies on having been provided an evaluation function for constraints that aborts when an uninitialized choice variable is encountered and otherwise returns true or false.

Clearly related to our work is consistency checking. Our approach builds on top of an existing UML consistency checking approach [10]. However, it must be stressed that not every (UML) consistency checking mechanism would suffice. The iterative nature of eliminating choices suggested in this work ideally requires incremental consistency checking as also proposed in xLinkIt [24] and ArgoUML [25]. Batch consistency checking approaches would work with our approach, but certain types of changes, such as adding choices or removing constraints, would become computationally more expensive.

An orthogonal approach that is complementary to ours, but shares some of the same concerns, is that of automatic repair of constraints (see work on consistency control in deductive databases [22], repair framework [13], and repair actions [23]). Here, the idea is to allow people to make mistakes in modeling and then show them options for fixing them. This could be made to fit tactically in a system based on our algorithm when inconsistencies arise. However, the choices they identify do not necessarily correspond to our ambiguity-choice pairs.

It must be emphasized that dealing with choices in design models is about more than just reducing infeasible choices. It is outside the scope of this paper to discuss other aspects for managing the designer's choices, such as keeping a formal history of the choices [5] for later maintenance, using version control and rollback mechanisms [29], recording why each alternative is rejected—e.g., why a service was modeled as a "class" rather than as a "method" [30], or managing the problem-domain dependencies among the decisions [1]. We simply provide the foundational work for systems that allow designers to record what they believe to be the significant design choice-points and the alternatives to be considered.

3 Illustration

In this paper, we assume that ambiguities are allowed in any place in a model; for example, choices for types, elements of sets, and even for methods of types for which the type itself is ambiguous are all allowed. Also, no preference exists that declares certain choices to be better than others. The designer is, however, allowed to remove choices manually or add new constraints at any time to help further eliminate choices.

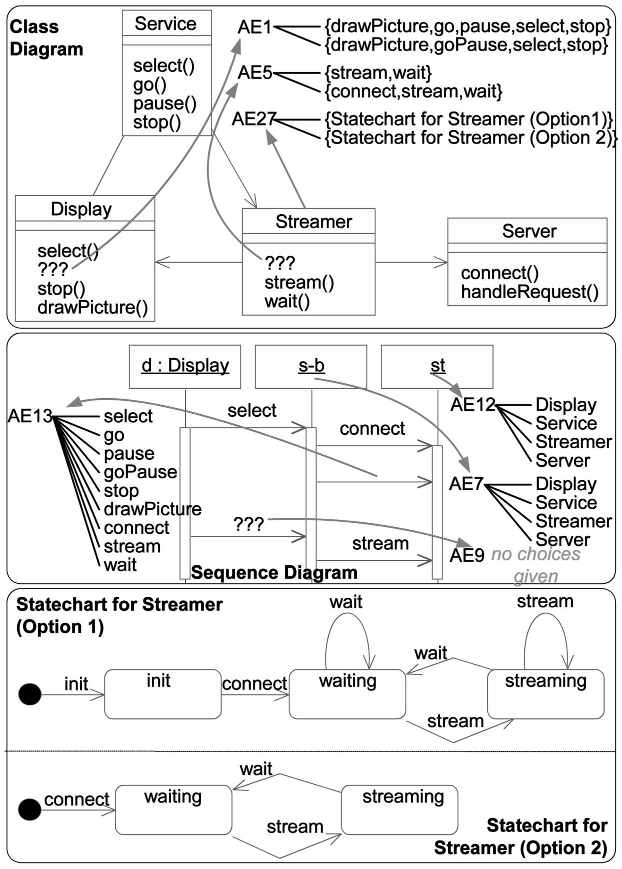


Fig. 1. UML model containing various ambiguities.

To illustrate the kinds of choices we want to support, Fig. 1 introduces a running example comprised of several related ambiguous models written in the Unified Modeling Language (UML) [26]. The given models represent an early design time snapshot of a real, albeit simplified, video-on-demand system [6]. It includes a class diagram (top), a sequence diagram (middle), and a state chart diagram (bottom). The class diagram represents the component structure of the system: a *Display*, a *Service*-providing component, a *Streamer* that processes data, and a *Server* that provides the data. Three ambiguities are embedded in the classes. For example, *Display* has the mandatory set of operations *select, stop*, and *drawPicture*, but it also has a design choice, indicated by "???", of whether to include certain combinations of *go, pause*, and *goPause* (the arrow leading from the class to the upper right of the figure points to all choices of operations for *Display* that the designer wants to allow). This ambiguity is the result of the design-time uncertainty of whether the go (play) button of the video player should also double as the pause button while playing or whether there should be separate buttons for play and pause. Note that the "???" label is only a visual cue for the designer since modeling tools generally do not support ambiguities and choices.

1	**Message must be an operation in receiver's class**
	operations = message.receiver.base.operations
	return operations->name->contains(message.name)

2	**Message calling direction must match class association**
	in = message.receiver.base.incomingAssociations
	out = message.sender.base.outgoingAssociations
	return in.intersectedWith(out)<>{}

3	**Sequence of message actions must match sequence of statechart events (**definition sketched in discourse)

Fig. 2. Sample consistency rules.

The class *Streamer* contains ambiguities as well. In UML, the class's behavior must be described in the form of a single statechart diagram, but two conflicting choices are available. Option 1 assumes a connection delay and allows an interaction protocol where *wait* and *stream* events may occur anytime. Option 2 is different in that no explicit connection delays are modeled and *wait* and *stream* events may occur in a predefined order only. Both statechart options have pros and cons and the designer thought it best not to eliminate either choice at this development stage. Related to this, the designer also declared the *connect* method for the class *Streamer* as optional, i.e., ambiguous as to its presence.

The sequence diagram describes the process of selecting a movie and playing it. During design time, redundant specifications are often a hindrance to the design process. Hence, a common activity is to ignore the actual types of objects during the design process. Ideally, the system can fill them in (as in Haskell [16]). Here, the designer has omitted the types for two of the three objects (only the type of the first object is known to be *Display*). Also, the message arrow between *connect* and *stream* was perhaps misspelled and it now remains unknown what action name belongs to it. All these cases define ambiguities where the class names are given as choices for the object types and the method names as choices for the missing message action (see arrows to choices). For realistic use, we find it necessary to also support ambiguities without choices (i.e., if the designer did not yet capture alternatives). The ambiguity AE9 in the sequence diagram is such an example.

Although all choices in the given model are valid, user-defined choices, we will illustrate later that certain combinations of choices are invalid. Consistency and well-formedness rules for UML describe conditions that every model must satisfy for it to be considered a valid UML model, comprised of syntactic and semantic constraints. Fig. 2 lists three such consistency rules for validating sequence diagrams on class and statechart diagrams. The next section demonstrates how the given, user-defined choices in models are reduced while evaluating these kinds of constraints.

4 Algorithms

4.1 Definitions

A few definitions are useful. CSP calls ambiguities "variables" whose choices are values from a "domain." In this discussion, we adhere to the ambiguity/choice terminology to avoid terminology conflicts with typical software engineering terms. Each ambiguity is designated to be a unique element of the type Ambiguity. The choices available to an ambiguity are similarly unique elements of the type Choice. The type Pairings is defined as sets of Ambiguity-Choice pairs, specified:

$$\mathrm{Pairings} = \mathrm{Ambiguity} \leftrightarrow \mathrm{Choice}.$$

(The Z notation constructs used in the definitions will be described as they are encountered.) This defines Pairings to be a *relation* between ambiguities and choices. For example, the object "s-b" in the sequence diagram in Fig. 1 is ambiguous (AE7) in that it does not define its class type (called the base type in UML). This means that we do not know what class this object instantiates and there are four choices: *Display, Service, Streamer*, or *Server*. Similarly, the object "st" is also ambiguous (AE12) because it does not define its base type. Although both ambiguities have the same choices, these ambiguities and their choices are independent of one another. That is, choosing the class *Display* for ambiguity AE7 is independent of choosing the class *Display* for AE12.

In CSP, a constraint is a Boolean condition over a set of variables—a CSP constraint typically identifies the variables explicitly. In UML, a constraint is a black-box function that abstractly defines the correctness of a model. A UML consistency rule does not identify ambiguities directly, but instead provides navigation instructions for the UML model. For example, consistency rule 2 evaluates whether the message calling direction is consistent with the association direction required by the class structure. This consistency rule does not identify any model elements in Fig. 1 explicitly nor does it identify ambiguities there. This consistency rule can be invoked multiple times in Fig. 1 to separately determine the truth values for all messages depicted there (messages are the horizontal arrows among the objects). Each of these invocations will be referred to as a distinct "constraint" in the model below. Depending on what message the rule is evaluated on, it may or may not encounter ambiguities. If it does encounter ambiguities, it may encounter different ones. It is generally not feasible to predict in advance whether any given consistency rule encounters an ambiguity during its evaluation.

If a model contains ambiguities, then the evaluation of a constraint may encounter one or more of its ambiguities. The constraint then requires the use of an Assignment to resolve the ambiguities to determine the truth value. An Assignment is a decision on which choice to take for any ambiguity encountered during the evaluation of a constraint. An assignment is thus a *function* that given an Ambiguity, returns a Choice:

$$\mathrm{Assignment} = \mathrm{Ambiguity} \rightarrow \mathrm{Choice}.$$

For example, a valid assignment for the ambiguities AE13, AE12, and AE7 in Fig. 1 is $\{\mathrm{AE13}| \rightarrow \mathrm{select}, \mathrm{AE12}| \rightarrow \mathrm{Display}, \mathrm{AE7}| \rightarrow \mathrm{Service}\}$ (notice Z's maplet symbol "$| \rightarrow$" that indicates pairs, e.g., AE13 maps to select). This assignment defines exactly one choice per ambiguity.

A constraint is then simply a function from an Assignment onto a truth-value, written:

$$\mathrm{Constraint} : \mathrm{Assignment} \rightarrow \mathrm{Boolean}.$$

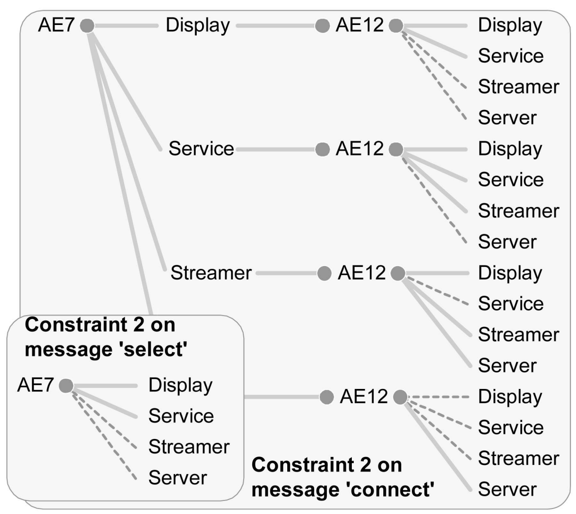


Fig. 3. Constraint 2 evaluated on message *select* (left) and message *connect* (right).

That is, given a set comprised of a choice for each ambiguity, determine whether the constraint is true or not. The application of such a constraint to an assignment will be written, simply, "c d." Think of d as a table lookup function that c must use whenever it encounters an ambiguous reference. For example, constraint 2 evaluates whether the message calling direction is consistent with the association direction required by the class structure (in Fig. 1). Assume the designer selects the choice *Service* for ambiguity AE7 in Fig. 1 (the Assignment). Notice that constraint 2 must hold for all messages of all classes. If constraint 2 is applied to message *select* of the *Display* class, then it navigates to its receiver and sender objects and requests their base types. For message select, the sender object's type is *Display* and the designer requires that the ambiguous receiver object's type be *Service*. Thus, the constraint evaluates to true because the *Display* class in Fig. 1 is allowed to call the class *Service*, indicated by the bidirectional association between the two classes in the class diagram. On the other hand, this same constraint would not evaluate to true if the receiver object's type were *Streamer* because *Streamer* is not allowed to call *Display*.

Constraint 2 can only be evaluated on message *select* if the ambiguity AE7 is resolved. The bottom-left of Fig. 3 depicts the four choices for that ambiguity. Since the construct only contains one ambiguous reference, there are four singleton assignments for this constraint that result in the constraint being either true or false. That is, the assignment $\{\mathrm{AE7}| \rightarrow \mathrm{Display}\}$ results in the constraint being true, while the assignment $\{\mathrm{AE7}| \rightarrow \mathrm{Streamer}\}$ results in it being false.

Constraint 2 may also be evaluated on message *connect* of the class Display. In this case both object types (sender/receiver) are ambiguous. To evaluate this constraint, a single Assignment must include both ambiguities. For example, the assignment $\{AE7| \rightarrow Display, AE12| \rightarrow Display\}$ causes the constraint to evaluate to true, whereas the assignment $\{AE7| \rightarrow Streamer, AE12| \rightarrow Service\}$ renders it false. Fig. 3 (right) shows that, if the sender object type is assumed to be *Display*, then the receiver object type must be either *Display* or *Service*.

Observe that constraint 2 evaluated on message *select* is like a unary CSP constraint (one ambiguity encountered) while the same constraint evaluated on message *connect* is a binary CSP constraint (two ambiguities encountered). We will also see that this structure may be irregular where the choices in one ambiguity affect what other ambiguities are encountered later (e.g., constraint 1 on message *connect* in Fig. 6). This aspect is not typical of traditional CSP problems, but it does not preclude the use of CSTs for dealing with ambiguous UML models. Indeed, this is beneficial for scalability because it implies that ***k-ary UML constraints are not as expensive to evaluate as typical k-ary CSP constraints***. We discuss this in detail later.

4.2 What the Developer Needs

Our vision is that the designer is exploring a design space, largely manually, by laying out new model elements, adding/removing ambiguities and choices, adding/removing constraints, and making choices for individual ambiguities or whole sets of ambiguities—until, finally, there is a single assignment with which he or she is satisfied. Naturally, this assignment must satisfy the CSP, but, by the time the designer is done, there may remain myriad assignments that would still be valid solutions. ***We reemphasize, the designer is really only interested in one and no tool is able to know the actual constraints that make this the best—or a "good enough"—assignment***. Formulating them to reflect matters of taste, style, or systematic design rules would be too expensive and distracting. Hence, the focus of our support is on helping the designer to explore this space.

So, when a designer removes a choice from consideration, if this implicitly causes other choices to become infeasible, our approach will allow the user to examine the consequences before allowing the system to remove them. It must be possible for the user to "unmake" the choices and have the system respond reasonably, i.e., in some sense, "incrementally," [10] rather than requiring the designer to start again from scratch.

For example, the evaluation of constraint 2 on the message *connect* in Fig. 3 implies that $AE12| \rightarrow Server$ is only feasible if $AE7| \rightarrow Streamer$ is chosen. Thus, without considering any other constraint, a user's decision to eliminate $AE7| \rightarrow Streamer$ also requires that $AE12| \rightarrow Server$ be eliminated. We know this because all other assignments of this constraint that include $AE12| \rightarrow Server$;

$$\{AE7| \rightarrow Service, AE12| \rightarrow Server\},$$
$$\{AE7| \rightarrow Display, AE12| \rightarrow Server\},$$

and $\{AE7| \rightarrow Server, AE12| \rightarrow Server\}$ cause the constraint to evaluate to false. This kind of reasoning is typical for constraint propagation in CST. Fortunately, this kind of reasoning is computationally tractable because only individual constraints have to be investigated.

One must reformulate the problem statement above slightly to express our support for the developer during software development. Our support hinges on the maintenance of the set of *ambiguity-choice pairs*, called

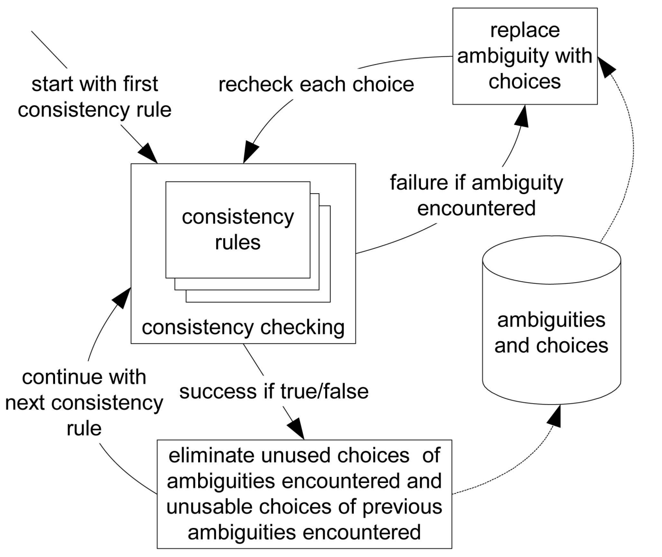

replace ambiguity with choices
start with first consistency rule
recheck each choice
consistency rules
failure if ambiguity encountered
consistency checking
ambiguities and choices
continue with next consistency rule
success if true/false
eliminate unused choices of ambiguities encountered and unusable choices of previous ambiguities encountered

Fig. 4. Workflow on consistency checking and choices elimination.

TheChoices, containing tagged choices for ambiguities that have not yet been eliminated by the designer or our support tools; TheChoices will always be a subset of the initial set of choices, ICh (note that we often refer to "a choice" but really mean an ambiguity-choice pairing). If one examines this set, there may be some ambiguities for which there is only a single pairing remaining; effectively, choices have been made for those ambiguities. If the entire set is comprised of such ambiguities, TheChoices will be an Assignment and the CSP will have been solved.

Fig. 4 depicts an overview of the workflow of our approach. We evaluate consistency rules one at a time. If a rule succeeds, then it returns true or false, depending on the correctness of the model. However, if an ambiguity is encountered, then the evaluation fails (an exception is thrown) and our approach replaces the ambiguity with one of its choices and reevaluates the consistency rule. The rule either succeeds now or it fails again if another ambiguity is encountered. Ambiguities are thus replaced systematically until the evaluation succeeds. The evaluation is then repeated for all choices and choice combinations of the encountered ambiguities to explore the space of "feasible" choices—those are the choices that satisfy the consistency rule and, thus, those are the choices a designer is interested in. The remaining, unused choices are eliminated. Also eliminated are dependent choices.

4.3 Choice Eliminations, One Constraint at a Time

For each constraint, CSTs keep track of all assignments that satisfy that constraint (i.e., cause it to evaluate to true). This set of all of the assignments satisfying a constraint is called the "positive assignments."

$$\begin{aligned} &\text{pos : Constraint} \rightarrow [\text{Assignment}]| \\ &\quad \text{pos c} = \{\text{d : Assignment}|\text{d} \subseteq \text{ICh} \wedge (\text{c d}) \bullet \text{d}\}. \end{aligned}$$

This says that Pos is a function from Constraints to sets of Assignments; it is defined to be (Z symbol: "|") the set of all possible assignments made from elements of ICh, filtered by those that satisfy constraint c. Notice the expression after the dot (Z symbol: "•") is the element collected up in the set in the Z notation, here the filtered Assignment d.

```
extendAssignment(c:constraint, d:assignment)
try
    satisfied=evaluate(c,d) // traditional constraint checking
    if satisfied then add(d,assignments(c))
catch exception for ambiguity a
    for all choice∈choices(ambiguity)
        extendAssignment(c, d ∪ {a|? choice})
        pos c = assignments(c) = Φ
        extendAssignment(c, Φ)
    return assignments(c)
```

Fig. 5. Algorithm for constraint evaluation and minimal, positive assignments.

For example, the set of positive assignments for constraint 2 applied to message select is the subset of the assignments from Fig. 3 that caused the constraint to evaluate to true: $\{\{\text{AE7}| \rightarrow \text{Display}\}, \{\text{AE7}| \rightarrow \text{Service}\}\}$. And, that the set of positive assignments for constraint 2 applied to message *connect* is

$$\begin{aligned} &\{\{\text{AE7}| \rightarrow \text{Display}, \text{AE12}| \rightarrow \text{Display}\}, \\ &\{\text{AE7}| \rightarrow \text{Display}, \text{AE12}| \rightarrow \text{Server}\}, \\ &\{\text{AE7}| \rightarrow \text{Service}, \text{AE12}| \rightarrow \text{Display}\}, \ldots, \\ &\{\text{AE7}| \rightarrow \text{Server}, \text{AE12}| \rightarrow \text{Server}\}\}. \end{aligned}$$

The approach actually maintains, for each constraint, the set of *smallest* assignments that satisfy the constraint. As was discussed earlier, UML constraints are black-box constraints and it cannot be predicted what ambiguities they will encounter.

Our approach determined these assignments through *model profiling*. During the evaluation of the black-box UML constraints, the model profiler observes the runtime behavior of these constraints. The constraint evaluation *succeeds* if no ambiguities are encountered (i.e., it evaluates to true or false). But, it aborts if an ambiguity is encountered. The exact details of the profiling are discussed in [11], but we essentially use instrumentation technology to spy on commercial modeling tools, such as IBM Rational Rose, to observe, in real-time, what model elements are accessed. If we observe that a constraint accesses an ambiguous model element, then we abort the evaluation of that constraint and report back the ambiguity. The ambiguity is then systematically replaced by its choices in the constraint and revalidated. Fig. 5 describes this algorithm. It recursively explores assignments through trial and error. If a constraint aborts, then the encountered ambiguity is replaced followed by a recursive reevaluation of the constraint (note: in our implementation, the constraint evaluation is aborted through an exception caused by accessing an ambiguous model element). The recursion continues, substituting choices made for the required ambiguities until all assignments have been explored. Once successful, the assignment consists of all choices that were encountered and substituted. This mechanism guarantees assignments to be minimal (a set of assignments is minimal if no subset of it exists that could also successfully evaluate the same constraint) because no ambiguities are ever replaced by choices unless required for successful constraint evaluation. It is important to mention that the choices for any ambiguity are chosen from TheChoices. Notice now that **Pos c** is really just the set of assignments that can be made by extending the empty assignment via extendAssignment.

If a constraint encounters an ambiguity that is undefined (i.e., for which there are no defined choices), then it cannot be evaluated. Such a constraint is reported to the user as an unresolved constraint with a pointer to the undefined ambiguity (i.e., the last ambiguity encountered by that constraint). If a constraint encounters too many ambiguities, then its evaluation will also abort. Such a constraint is reported to the user with a list of all encountered ambiguities (which is not necessarily a complete list of ambiguities as the evaluation was aborted). As was discussed previously, it is not necessary for the user to handle unresolved constraints as they do not negatively impact the conservative properties of our algorithm.

4.4 Mandatory and Optional Ambiguities

In CST, if a variable is used by a constraint, then all assignments of that constraint include a choice for that variable. Certainly, there are UML constraints where a particular ambiguity must be resolved in every positive assignment satisfying the constraint (i.e., those ambiguities in Pos c). Recall, e.g., that constraint 2 evaluated on message *select* was a unary constraint where every positive assignment included a choice for ambiguity AE7, and constraint 2 evaluated on message *connect* was a binary constraint, where every positive assignment included a choice for AE7 *and* AE12 (Fig. 3). The ambiguities that are referenced in every positive assignment are referred to as ***mandatory ambiguities*** (the following reads in Z, take the intersection of the domains of all Assignments in pos c):

$$\text{mand} : \text{Constraint} \rightarrow [\text{Ambiguity}]|$$
$$\text{mand c} = \bigcap\{\text{d} : \text{Assignment}|\text{d} \in \text{pos c} \bullet \text{dom d}\}.$$

All CSP formulations deal with mandatory ambiguities. The important observation to be made of a mandatory ambiguity is that: *If a choice for a mandatory ambiguity does not occur in at least one assignment in the set of positive assignments for the constraint, then that choice is invalid and can be eliminated safely* (see [20]). That is, since it is never used in an assignment that satisfies that particular constraint, it can never be used in an assignment that satisfies all constraints (i.e., because we know of at least one constraint that cannot be satisfied). This choice can be eliminated from consideration permanently. This elimination is based on the evaluation of a single constraint and does not depend on the evaluation of any other constraint. For example, the set of positive assignments for constraint 2 on message *select* includes the mandatory ambiguity *AE7* and it only uses the choices *Display* and *Service*. Since the choices *Streamer* and *Server* never satisfy this constraint, they are infeasible and can be eliminated (independently of all other constraints and choices). Another example is the list of positive assignments of constraint 2 evaluated on the message *connect*. It contains the two mandatory ambiguities, *AE7* and *AE12*. Since all their choices are used in at least one assignment, no choices can be eliminated. Recalling that the designer provides the set of Pairings, ICh, the following algorithm describes which choices can safely be eliminated based on mandatory ambiguities for each constraint in IC (this is an adaptation of AC-3 [20]):

$$\text{InvalidChoices} : \boldsymbol{P}\text{ICh} \rightarrow \boldsymbol{P}\text{ICh}|$$
$$\text{InvalidChoices TheChoices} = \cup\{\text{c} : \text{Constraint}|\text{c} \in \text{IC}\bullet$$
$$\cup\{\text{q} : \text{Ambiguity}|\text{q} \in \text{mand c}\bullet$$
$$(\{\text{q}\} \lhd \text{TheChoices})\}\setminus \cup\,\text{Pos c}\}.$$

This can be read, for each constraint union together all values for that constraint's mandatory ambiguities that occur in TheChoices and are not in some positive assignment for that constraint (in Z, the "\" is the set difference operator and the restriction of TheChoices to those pairs with q in the domain is written {q} ◁ TheChoices). These choices may safely be removed from the set, TheChoices, because each element is identified with a particular constraint that must fail if the choice is included. Notice that the union over the positive assignments for c is really the set of remaining choices valid in some assignment for c, so it is like a specialized version of TheChoices for each constraint. In fact, the implementation takes advantage of this fact and keeps track of the union rather than the individual assignments themselves. Notice also that, as the set difference is taken in the formula above, if all choices for a mandatory ambiguity are removed, that particular constraint will not be satisfiable and the problem will become inconsistent. It is easy to keep track of this and report it to the designer.

It is an important property of the algorithm that this precise description of where the removed choice causes failure is available; not all proof mechanisms have this capability and the designer could be at a loss for discovering exactly what went wrong. Again note that the actual algorithm will have to keep track of the invalid choices eliminated to show them to the designers so they can determine whether their choices had otherwise undesirable or surprising effects.

In most CSP formulations, all ambiguities are mandatory. Yet, in languages involving first-order logic and quantifiers (such as UML consistency rules), some ambiguities may not participate in all positive assignments of a constraint. The ambiguities that are referenced in some but not all positive assignments are referred to as ***optional ambiguities***. Optional ambiguities are conditionally dependent on the choices for other ambiguities. As an example for an optional ambiguity, consider the evaluation of constraint 1 on message *connect* (this constraint ensures that messages used in sequence diagrams are defined as methods in their corresponding classes). While applying this constraint to the message *connect* (see sequence diagram), it again encounters the ambiguity for the object's type. From before, we know that there are four types available for the object: *Display, Service, Streamer*, and *Server*. The constraint is thus applied to all four types with the following finding: *Service* does not have a method with the name *connect*. The set of methods for *Display* is ambiguous, but, after evaluating all its choices, none of them has a method with the name *connect*. *Streamer* has a method called

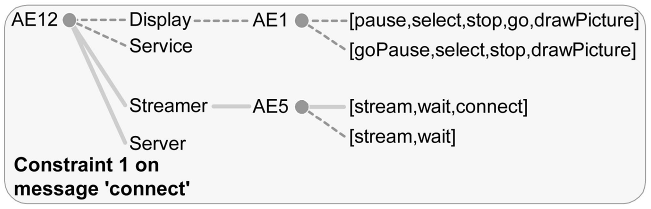

AE12
Display
Service
AE1
[pause,select,stop,go,drawPicture]
[goPause,select,stop,drawPicture]
Streamer
AE5
[stream,wait,connect]
[stream,wait]
Server
Constraint 1 on message 'connect'

Fig. 6. Assignments after evaluating constraint 1 on message *connect*.

connect, but only if the first choice of its ambiguous set of methods is chosen. Finally, *Server* also has a method called *connect*. In summary, the set of positive assignments is short and is comprised of only the two assignments

$$< \{AE12| \rightarrow Streamer, AE5| \rightarrow [connect, stream, wait]\},$$
$$\{AE12| \rightarrow Server\} >$$

(see Fig. 6). The ambiguity $AE12$ is part of all positive assignments and is thus mandatory, but the ambiguity AE5 is not because it is used in only one of the two positive assignments.

It is important to distinguish between mandatory and optional ambiguities. Obviously, choices $AE12| \rightarrow Display$ and $AE12| \rightarrow Service$ are infeasible because they are part of a mandatory ambiguity but not used in any positive assignment. It is, however, not possible to eliminate the choice $AE5| \rightarrow [stream, wait]$ because ambiguity AE5 is optional. That is, this choice only becomes infeasible if a developer decides on choice $AE12| \rightarrow Streamer$. Since there are two available choices for $AE12$ and the user has not picked one of them, it is not possible to safely eliminate the optional choice. It is not possible to eliminate the choices of $AE1$ for the same reason. Optional ambiguities may appear to be a hindrance to CSTs, but there are, in fact, two factors that are beneficial:

1. Optional ambiguities happen in parallel and do not add to the computational complexity in the same way as mandatory ambiguities do.
2. Optional ambiguities may become mandatory ambiguities and, thus, be eliminatable.

In CSP, if a constraint uses three variables, then it is a 3-consistency problem, which is much more complex to solve than a 2-consistency problem. The UML constraint 1 evaluated on message connect encounters three ambiguities, but it is a 2-consistency problem because AE1 and AE5 never occur in the same positive assignment. This is a very significant computational benefit. How optional ambiguities become mandatory ambiguities is discussed next.

4.5 Promoting Optional to Mandatory Ambiguities

CST eliminates choices based on the evaluation of individual constraints without investigating ambiguities not part of the positive assignments of that constraint and without investigating other constraints and their assignments.

The definitions of pos and (therefore) of mandatory have relied on the set of all possible assignments of *individual* constraints only. However, we know that constraints influence one another because they overlap in their use of common choices. Until now we have not taken into account the possible reduction of the choice space by eliminating the invalid choices from the positive assignments of each constraint when they are eliminated from TheChoices. In CST, this is the familiar *constraint propagation*. In other words, if a choice is eliminated, then it should no longer be part of any positive assignment of *any* constraint (neither mandatory nor optional). This can be accomplished by making the definition of positive assignment sensitive to the context of previously eliminated choices. To do this, after the designer eliminates choices from TheChoices, a process called EliminateChoices must remove from TheChoices those choices determined to be InvalidChoices.

So, to introduce context sensitivity into the computation of the assignments, they too are recomputed based on the dynamically changing value of the variable TheChoices. Notice that this also has an effect on the meaning of mandatory, so ambiguities that were optional may *become* mandatory.

$$\mathrm{pos : Constraint \rightarrow [Assignment]|}$$
$$\mathrm{pos\ c = \{d : Assignment|d \subseteq TheChoices \wedge c\ d \bullet d\}}.$$

The updated pos c now includes the condition that a positive assignment must evaluate a constraint to true and it must consist only of choices not yet eliminated from consideration, TheChoices—actually, those "not known to be invalid." Since TheChoices is a global set (i.e., choices are permanently eliminated from all constraints that use them), we have now created a feedback loop between Pos c and EliminateChoices. Hence, EliminateChoices must be applied until TheChoices converges to a unique set. That is, EliminateChoices eliminates choices in context of single constraints; however, this elimination has global consequences because it affects the positive assignments of all other constraints (previously evaluated ones and the ones not yet evaluated).

This implies that choices eliminated by one constraint need not be considered during the evaluation of other constraints. However, Fig. 3 or Fig. 6 did not consider this effect. Fig. 7 depicts the assignments of the same three constraints and their feedback loops during the elimination of choices. The top of Fig. 7 shows the assignments for constraint 1 on message *connect*, which was depicted in Fig. 6. Two choices can be eliminated: $AE12| \rightarrow Display$ and $AE12| \rightarrow Service$. The middle of Fig. 7 shows the assignments for constraint 2 on message *connect*; however, the already evaluated choices need not be considered. Fig. 3 previously depicted many positive assignments for this constraint with no unused choices—no choices could be eliminated. Without $AE12| \rightarrow Display$ and $AE12| \rightarrow Service$, however, $AE7| \rightarrow Display$ becomes infeasible and can be eliminated.

The feedback loop also affects positive assignments of previously evaluated constraints. For example, after evaluating the third constraint (Fig. 7 bottom), we find that only $AE7- > Service$ remains feasible. Yet, $AE7- > Streamer$ and $AE7- > Server$ were both used in positive assignments of the second constraint. It is necessary to adjust the positive assignments of all constraints that previously used the now invalid choices. This feedback is depicted in Fig. 7 (right side), where now only one positive assignment remains for the second constraint.

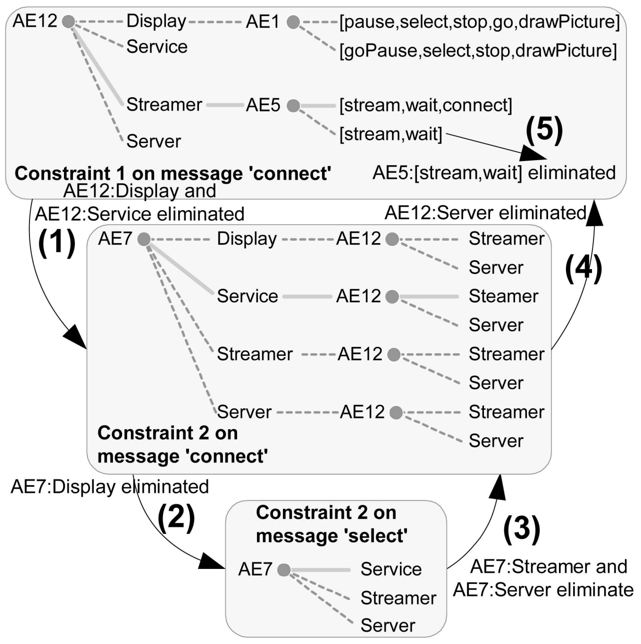


Fig. 7. Feedback loop of eliminating choices on past constraints.

We know from the previous discussion that a reduction of the set of positive assignments may also reduce how choices of mandatory ambiguities are used in context of that constraint. Indeed, we discover that the choice $AE12| \rightarrow Server$ is no longer used by any positive assignment of the second constraint—it can thus be eliminated also. The elimination of the choice $AE12| \rightarrow Server$ causes another feedback loop. As before, all positive assignments of constraints must be adjusted that used this latest eliminated choice. We find that $AE12| \rightarrow Server$ was used previously during the evaluation of the first constraint (see the top of Fig. 7). Recall that the list of positive assignments for constraint 1 on message *connect* was

$$< \{\mathrm{AE12}| \rightarrow \mathrm{Streamer}, \mathrm{AE5}| \rightarrow [\mathrm{stream}, \mathrm{wait}, \mathrm{connect}]\},$$
$$\{\mathrm{AE12.Server}\} >.$$

This included the mandatory ambiguity AE12 and the optional ambiguity AE5. However, after eliminating the now-invalid assignment $\{\mathrm{AE12}| \rightarrow \mathrm{Server}\}$, we find that AE5 is no longer optional. The status of the optional ambiguity is elevated to a mandatory one and now its unused choices become invalid: $\mathrm{AE5}| \rightarrow [\mathrm{wait}, \mathrm{stream}]$ is thus eliminated from TheChoices and no further action is required because no positive assignment of any constraint used this choice.

Formally, this implies that our algorithm eliminates choices based on the interplay among Pos c, Mand c, and InvalidChoices—considering the effects of mandatory and optional ambiguities. This is essentially constraint propagation in CSP and it stops when a stable set is found where no more choices can be eliminated and all positive assignments do not contain infeasible choices. Constraint propagation is deterministic. It also never eliminates a choice in error as long as the user's choices do not render the problem unsatisfiable.

CSTs have very fast algorithms to implement this feedback loop through the use of tables and counters. However, optional ambiguities are not part of the CST repertoire. Our implementation thus maintains the list of positive assignments for every constraint. The implementation also maintains dependencies among the positive assignments, ambiguities, choices, and constraints such that it is possible to navigate among them. For example, if a choice is eliminated, the choice knows all the positive assignments that use it (i.e., it was told by every positive assignment at the time of creation). Thus, there is no need for expensive searches across the entire space of positive assignments. In turn, the elimination of a positive assignment updates all its ambiguities and choices about its deletion such that a later choice elimination does not revisit this assignment again. Thus, an assignment is visited only once in the feedback loop, during its elimination—a linear complexity.

During the elimination of an assignment, it becomes necessary to recompute the unused choices of mandatory ambiguities for they must be eliminated also. Our implementation uses a simple counter system (adapted from AC-3 and later versions [20]) to quickly compute "mandatory" and "used." That is, ambiguities and choices maintain their separate counters for every constraint that uses them. If an assignment is created that uses a particular ambiguity and choice, then their counters are incremented for that constraint. At the time of the elimination of an assignment, a mandatory ambiguity is the one where the counter equals the size of positive assignments (an ambiguity is mandatory if it is being used in every assignment; otherwise, it is optional). Similarly, a choice is "used" if the choice counter of the constraint is greater than zero (there is at least one assignment for that constraint that uses the choice).

4.6 Impact of Unevaluated Constraints

Finally, it is worth noticing that, since the algorithm only ever eliminates choices and the addition of a new constraint could never actually make an eliminated choice suddenly satisfy the constraint in which it was mandatory and false, the algorithm will never have eliminated a choice that should "reappear" if new constraints are added, even if new ambiguities are introduced or new constraints are added. The only situation in which a decision made by the algorithm is wrong occurs when a previous constraint is determined by the user to be unnecessary.

5 Defining Models, Ambiguities, and Their Choices in IBM Rational Rose

Our approach to ambiguous reasoning is integrated with IBM Rational Rose for creating and maintaining UML models. Since Rose is not capable of modeling choices, we created a separate Rose plug-in for defining choices. The plug-in links the Rose model elements and the choices. The ambiguities reasoning tool is yet another Rose plug-in that uses the Rose model and its choice information as was described in this paper. Fig. 8 depicts Rose on the top and a screen snapshot of our reasoning tool on the bottom. The top portion of our tool shows the positive and negative determination of a selected constraint and the bottom

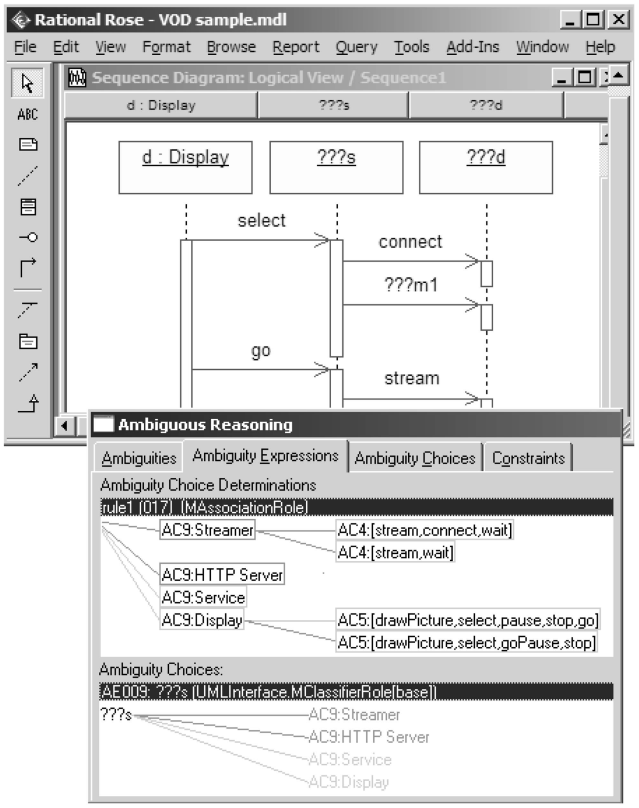

Rational Rose - VOD sample.mdl
File Edit View Format Browse Report Query Tools Add-Ins Window Help
Sequence Diagram: Logical View / Sequence1
d : Display
???s
???d
select
connect
???m1
go
stream
Ambiguous Reasoning
Ambiguities
Ambiguity Expressions
Ambiguity Choices
Constraints
Ambiguity Choice Determinations
rule1 (017) (MAssociationRole)
AC9:Streamer
AC4:[stream,connect,wait]
AC4:[stream,wait]
AC9:HTTP Server
AC9:Service
AC9:Display
AC5:[drawPicture,select,pause,stop,go]
AC5:[drawPicture,select,goPause,stop]
Ambiguity Choices:
AE009: ???s (UMLInterface.MClassifierRole[base])
???s
AC9:Streamer
AC9:HTTP Server
AC9:Service
AC9:Display

Fig. 8. Ambiguous reasoning integrated with IBM Rational Rose.

portion shows an ambiguity with its remaining valid choices. The reasoning tool is also integrated with the UML/Analyzer consistency checking tool [10] (another Rose plug-in) which is used for constraint validation. In the next section, we demonstrate the tool's scalability, having evaluated over 27 third-party models with tens of thousands of model elements and over 16,000 constraints.

6 Validation

This section demonstrates that the computational complexity of our approach yields a method whose use is inexpensive enough that developers can use it comfortably. We also demonstrate that its results are a near-optimal approximation of the ideal solution. Our goal here is to demonstrate that typical models that one encounters in practice, even models with tens of thousands of elements, can be evolved using our tools.

Table 1 lists 25 third-party models (some scraped from the Internet, others provided by industrial partners) and two in-house developed models that were used to evaluate our approach. These fully developed models differ substantially in model size and types of model elements used, yet almost all of the models are large and purport to accomplish useful tasks. The models were evaluated using an existing UML consistency checking tool, the UML/Analyzer [10]. The UML/Analyzer tool was preprogrammed to handle 24 consistency rules [28] (including the ones in Fig. 2) and, in total, these rules were evaluated over 16,000 times for the 27 sample models. The 24 consistency rules were chosen because they covered the needs of our industrial partners, foremost the Boeing Company. The rule set is reasonably complete for dealing with the consistency of sequence diagrams.

TABLE 1
Characteristics of the 27 Evaluation Models, Number of Constraints, and Constraint Sizes

	$M_{\#}$		$C_{\#}$	C_{size}		
	instances	fields		avg	min	max
ANTS Visualizer	919	481	231	4.8	1	54
ATM	168	268	86	6.3	1	59
Biter Robocup	1817	448	222	2.6	1	52
BMS	1251	1572	591	9.9	1	95
Calendarium	1910	2065	802	12.6	1	117
DESI	1339	466	207	3.1	1	127
eBullition	255	257	94	10.8	1	58
DSpace	4747	955	549	2.3	1	128
OODT	8215	2273	1131	2.8	1	206
HDCP Defects	613	432	196	3.1	1	34
HMS	1328	1630	422	10.7	1	96
Insurance System	10045	307	118	3.3	1	16
iTalks	1270	911	466	16.2	1	55
Microwave Oven	133	217	46	12.7	1	98
MVC	242	161	59	10.1	1	42
NZ Airport	517	489	196	3.6	1	22
Obstacle Game	1214	810	251	14.9	1	101
Teleop. Robot	545	464	168	7.3	1	47
Vacation/Sick Leave	878	964	324	11.4	1	67
Wordpad	6541	397	180	3.1	1	44
Home Appliance	488	470	307	15.6	1	57
Inventory/Sales	627	673	274	26.0	1	152
LCA Project	963	1147	322	7.3	1	349
UML Tutor	297	312	115	6.7	1	41
Curriculum	369	514	184	25.0	1	130
Boeing PCES	16988	6868	3104	9.9	1	151
Boeing MoBIES	33135	12728	6219	10.5	1	151

6.1 Computational Complexity

Our approach consists of essentially two steps: The first step identifies positive assignments for every constraint individually and the second step eliminates choices.

Step 1: Depending on how many ambiguities are encountered per constraint, APC, and how many choices per ambiguity exist, $ChPA$, there are $DPC = ChPA^{APC}$ assignments per constraint. The number of assignments per constraint increases exponentially and it is important for scalability to ensure that the exponential factor APC is small and does not increase with the size of the model. For a given number of constraints $C_{\#}$, there are thus $D = C_{\#} * ChPA^{APC}$ total assignments, of which we only record the positive assignments, a subset of D. The cost of a single evaluation of a constraint is proportional to the number of fields visited C_{size} and a constraint must be evaluated for every assignment. The computational complexity of computing all assignments of all constraints is thus $O(C_{\#} * C_{size} * ChPA^{APC})$.

Step 2: For every eliminated choice, it is necessary to identify all previously computed, positive assignments that are affected. Only the assignments of those constraints are affected that use the eliminated choice. Since an eliminated choice is equivalent to a changed ambiguity, we define CPA—constraints per ambiguity—to be the number of constraints affected by a change in an ambiguity. Only those assignments must be eliminated that use the choice, the assignments per choice, $DPCh = ChPA^{(APC-1)}$ (i.e., the eliminated choice and all its combinations with the $APC - 1$ other ambiguities). After the assignments have been eliminated, it is necessary to check whether some optional ambiguities have become mandatory and whether some of the choices of mandatory ambiguities are no longer used (i.e., these choices may then be eliminated also). We demonstrated that this computation can be ignored for computational complexity, which implies a worst-case cost for Step 2 of $APC * ChPA$. The computational complexity of eliminating a choice is thus $O(CPA * (ChPA^{(APC-1)} + APC * ChPA))$.

In the following, we will present empirical evidence that C_{size}, CPA, and APC are small values that do not increase with the size of the model. We will further demonstrate that $C_{\#}$ is linear in the size of the model. Only the value of $ChPA$ is unknown as it is a user-definable value, but, in software design efforts, it is expected to be small (i.e., the user likely does not want to define a large number of choices per ambiguity). We thus conclude that the computational cost of Step 1 is linear in the size of the model and that the cost for Step 2 is not affected by the size of the model (i.e., despite the exponential factor).

6.2 Scalability Drivers

The scalability (computational complexity) of *Ambiguous Reasoning* is that of *Consistency Checking* plus the overhead imposed by creating and navigating the dependency network discussed in this paper.

Our approach does not improve on consistency checking. We thus inherit its computational complexity, which, fortunately, is linear in the size of the model and the number of consistency rules used, $C_{\#} * C_{size}$. That is, we applied the consistency checker (the UML/Analyzer) to the 27 sample projects and measured the number of constraints $C_{\#}$ and constraint sizes C_{size}. The number of times constraints are evaluated depends on the number and types of available model elements. Recall that most of the constraints in the UML model come about from instantiating rules about classes or methods in general, so one can expect a given number of constraints to arise whenever a class is ambiguous, for example. However, given that the 27 sample projects were very diverse in contents and size, we expected some variability in $C_{\#}$, but found that it followed a linear trend closely, that is, $C_{\#}$ rose linearly with the size of the model $M_{\#}$ (see Table 1). Similarly, we expected C_{size} to be variable for the same reasons. Indeed we measured a wide range of values between the smallest and largest C_{size} (min/max), but found that the average did not increase with the size of the model (see Table 1). Thus, we conclude the factor $C_{\#} * C_{size}$ to be linear with the size of the model in these typical applications. Note that C_{size} not increasing with the size of the model seems counterintuitive. One would expect that constraints would evaluate more model elements the larger the model. However, this turns out to be wrong. The increase in model size is fully absorbed by an increase in $C_{\#}$. This finding implies that the model size may increase, but the relative complexity of individual model elements do not.

TABLE 2
Relationship between CPA and APC Illustrated on the VOD Sample

		CPA = 1.29						
		A1	A5	A7	A9	A12	A13	A27
APC = 2.25	C2(select)			X				
	C2(connect)			X		X		
	C3(st)					X	X	X
	C1(connect)	X	X			X		

Notice from the incremental (second) formula above that the two factors, CPA and APC, are key evaluation parameters for the computational complexity of ambiguous reasoning. These factors are related in an interesting way. Consider the simple example in Table 2. The columns and rows of the table depict the constraints and ambiguities of the illustration we used throughout this paper. If a field contains an "X," then the constraint (row) encountered the ambiguity (column) during the constraint's evaluation. Again, APC represents how many such ambiguities are encountered per constraint *on average*. That is, we find that the four constraints in the illustration encountered $1 + 2 + 3 + 3$ ambiguities, which is equal to 2.25 ambiguities per constraint in average. *CPA* is the average number of constraints per ambiguity. The seven ambiguities in the table encountered $1 + 1 + 2 + 0 + 3 + 1 + 1$ constraints for an average number of 1.29 constraints per ambiguity.

If we denote $A_{\#}$ to be the number of ambiguities (columns of the table), then we can say that there are exactly $C_{\#}$ times $A_{\#}$ fields in the table and at most that many "X"s. The total number of "X"s can be tallied in two ways, yielding the equality: $C_{\#} * APC = A_{\#} * CPA$.

In addition, we can compute APC in a simple way that involves the constraint size, model size, and number of ambiguities:

$$APC = \frac{A_{\#}\, C_{size}}{M_{\#}}.$$

That is, the ratio of $A_{\#}$ to $M_{\#}$ defines the likelihood that a single model element is ambiguous. Since C_{size} defines the number of model elements used per constraint, the product of C_{size} and the likelihood must be the number of ambiguities per constraint. Hence,

$$CPA = \frac{C_{\#}\, APC}{A_{\#}} = \frac{C_{\#}\, C_{size}\, A_{\#}}{A_{\#} M_{\#}} = \frac{C_{\#}\, C_{size}}{M_{\#}}.$$

There are two interesting observations about the two formulas for CPA and APC. First, both involve the ratio C_{size} to $M_{\#}$. Second, there is no information about ambiguities in the formula that computes CPA! It must be emphasized that

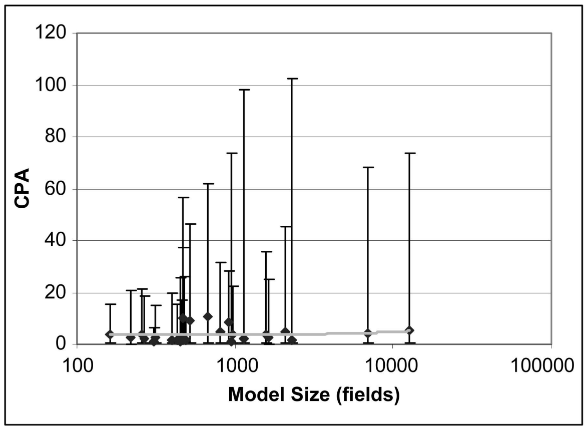


Fig. 9. Linear behavior of CPA.

$M_\#$, $C_\#$, and C_{size} are scalability factors of consistency checking that are inherited by ambiguous reasoning only. In effect, CPA is not just the number of constraints per ambiguity. Since an ambiguity is a special model element, CPA is in fact the number of constraints per model element. That is, if a model element changes (whether it is ambiguous or not), CPA tells how many constraints are affected by the change. This explains why CPA is computable without any information about ambiguities.

For the scalability of ambiguous reasoning, we need to prove that CPA and APC are scalable with respect to the model size $M_\#$. This is straightforward because we know that C_{size} does not increase with the size of the model and $C_\#$ grows linearly with the size of the model. That is, since both are divided by the model size $M_\#$, this suggests that CPA should not increase with the size of the model because the growth of $M_\#$ will cancel out the growth of $C_\#$.

To demonstrate this linear behavior empirically, we computed the values of CPA for the 27 sample projects. Fig. 9 depicts the linear behavior for CPA and shows that its value is, on average, 3.7 and that it is not affected by the size of the model. This CPA value implies that there are only few constraints affected by a change in the model (i.e., an ambiguity change such as the user's deletion of a choice).

Given that APC and CPA only differ in $A_\#$ and $C_\#$, this suggests that $A_\#$ is allowed to grow linearly with the size of the model for APC to not be affected by the increased size. Thus, the size of the model does not negatively affect APC and the number of ambiguities in the model may rise linearly with the model size, which is important for usability. That is, a user may likely want to define more ambiguities for larger models than for smaller ones and, for computational scalability, the user may do so for as long as the ratio $A_\#$ to $M_\#$ stays constant.

Fig. 10 empirically confirms the linear behavior of APC on the 27 sample projects (note the exponential x-axis). The figure shows APC in context of different ratios of ambiguities ($A_\#$ to $M_\#$ ratios). This is important because the number of ambiguities is a user-defined factor and it may differ significantly among projects (i.e., note that the number of constraints was not user-defined, but implied by the types and instances of model elements). It is thus

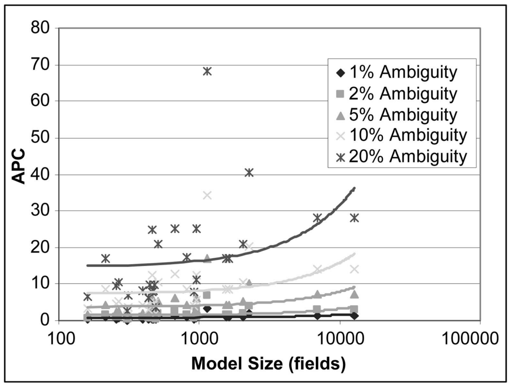


Fig. 10. Linear behavior of APC.

necessary to consider the effect of APC in relationship to the different ratios of ambiguities.

	1%	2%	5%	10%	20%	30%
Average *APC*	0.09	0.19	0.47	0.94	1.87	2.81

Thus far, we have presented the scalability of APC and CPA in that its growth is independent of the model size. While there is some fluctuation among the 27 sample projects, this fluctuation is small and we discuss the worst-case behavior below. Recall that this is important for scalability because the first step computes the set of positive assignments for every constraint and there is an exponential number $ChPA^{APC}$ of potential assignments per constraint, where $ChPA$ is the number of choices per ambiguity.

This exponential growth would pose a significant problem to our approach if the exponential factor were to increase with the size of the model. As was demonstrated, APC is essentially a small, constant factor and it is not affected by the size of the model. It is only affected by how ambiguous the model is and how many types of constraints are used. That is, APC stays relatively constant, while the model size, $M_\#$, and the number of ambiguities, $A_\#$, increase for as long as $A_\#$ does not increase faster than $M_\#$. We believe that this is a reasonable assumption.

This reasoning has, however, one significant flaw. We made the argument in favor of average numbers (e.g., average number of ambiguities per constraint). We have not actually demonstrated our approach's computational performance under worst-case conditions (e.g., constraint size is large). The table below depicts the APC for the worst constraints of the over 16,000 evaluated constraints. While the APC is still a relative constant, the exponential factors are unreasonable for all but the smallest ratios of ambiguity ($< 2\%$ ambiguity).

	1%	2%	5%	10%	20%	30%
Worst Case *APC*	3.42	6.83	17.08	34.17	68.34	102.51

To make matters worse, we previously assumed an equal distribution of ambiguities across the entire model. It is, however, possible that there are areas in the model with a higher concentration of ambiguities. In these areas, even a smaller constraint would encounter a larger number of ambiguities and run into scalability problems. It is thus unlikely that our approach is able to evaluate all constraints.

Fortunately, a basic assumption of our approach is that the algorithm does not even have all the constraints that influence the choices being made. So, the algorithm can "at will" decide not to evaluate some constraints. The conservative properties of our approach are guaranteed regardless of how many constraints are actually validated. However, the effectiveness of our approach decreases with the fewer constraints it can evaluate (i.e., it becomes less optimal, albeit it is still correct). It is thus desirable to evaluate as many constraints as possible.

While it is not possible to predict the worst-case execution scenarios, we do know that worst-case scenarios increase with the constraint sizes. A constraint that is twice as large is likely to encounter twice as many ambiguities. The true scalability issue here is thus the constraint size, C_{size}. While there are some outliers where C_{size} is large, we found that the likelihood of C_{size} being large decreases drastically (95 percent of all 16,000 constraints used fewer than 26 model elements). This implies that most constraints can be evaluated with a small APC factor, as depicted in the table below. The table shows APC for $C_{size} = 25$ (95 percent of all constraints) and $C_{size} = 10$ (80 percent of all constraints).

	1%	**2%**	**5%**	**10%**	**20%**	**30%**
***APC* for $C_{size} \leq 25$**	0.25	0.5	1.25	2.5	5	7.5
***APC* for $C_{size} \leq 10$**	0.1	0.2	0.5	1	2	3

In summary, while some constraints may be large (even global), most consistency and well-formedness constraints are actually very small conditions that involve only a handful of model elements. Seventy-four percent of all constraints evaluate five or less model elements. In the worst-case scenario, these constraints encounter five or fewer ambiguities (every model element is ambiguous), but this worst-case scenario is not meaningful to a user (i.e., what is the meaning of a model, or subset thereof, that is 100 percent ambiguous). For comparison, the illustration in this paper used seven ambiguities on 89 model elements. This implies an ambiguity of less than 1 percent. We do not believe that users would define models more ambiguous than this, but we see that, computationally, we can handle 5 percent ambiguities for more than 95 percent of all constraints. We believe that this is a very reasonable assumption.

To illustrate what these values for APC imply in practice during the evaluation of constraints, we performed another experiment where we again randomly seeded ambiguities into models with arbitrary ratios of ambiguities ($A_{\#}/M_{\#}$). We then evaluated constraints of various sizes on the model and observed the likelihood of these constraints becoming unsatisfiable because of too many assignments. Thus, for $A_{\#}/M_{\#}$ being between 0 percent and 20 percent and C_{size} being between 5 and 50, what percentage of the total number of constraints could not be evaluated because they exceeded the threshold of $DPC \leq 100$ (number of assignment above 100)? The result of this experiment (not depicted) showed that, for small ratios of ambiguities, most of the constraints could be evaluated without exceeding the threshold. We observed a nonlinear relationship between the ratio of ambiguities and the percentage of unscalable constraints, which is not surprising given the exponential factor involved (i.e., $DPC = (ChPA)^{APC}$). Recall that 95 percent of all constraints evaluate less than 25 model elements. Of these 95 percent of constraints, 90 percent can be evaluated without encountering scalability issues for models with less then 8 percent ambiguity. Of the 80 percent of constraints that evaluate less than 10 model elements, 90 percent can be evaluated without encountering scalability issues for models with less than 17 percent ambiguity. A ratio of ambiguity of 8 percent or more is huge. It would support thousands of ambiguities in the 27 sample projects (e.g., for the Boeing MoBIES project, 8 percent ambiguity allows for 1,018 ambiguities). We expect that this limitation in the number of ambiguities is more than adequate for designers. After all, each designer is, in some sense, balancing the trade-offs of all the ambiguities at once. Although large projects involve several designers, this is still a very large number of design axes to maintain control over.

6.3 Effectiveness

We evaluated the effectiveness of our approach by comparing its results with the optimal solutions on thousands of scenarios. Deriving an optimal solution is easy, but computationally very expensive. We thus had to limit the validation of effectiveness to small-scale models. To ensure that our findings are applicable to large-scale models, we scaled case studies down, but maintained the critical ratios. That is, the effectiveness (not the computational cost) of using our approach on an $M_{\#} = 10,000$ model element model with $A_{\#} = 1,000$ ambiguities evaluated with constraints of average size $C_{size} = 40$ is the same as the effectiveness of using an $M_{\#} = 100$ model element model with $A_{\#} = 10$ ambiguities evaluated with constraints of average size $C_{size} = 4$ (a proof for this is omitted, but can be inferred from the previous section). Thus, our validation method was to randomly inject ambiguities into small-scale models. Through this method, we systematically explored the effectiveness of our approach in relationship to different values for $ChPA$, APC, and CPA on hundreds of scenarios.

Most importantly, this analysis confirmed that our approach is indeed conservative in reducing choices and detecting inconsistencies. Regardless of the usage of our approach (ratios), our approach always reduced a subset of the choices also reduced by the optimal approach and it identified a subset of the identifiable inconsistencies, even in the presence of unscalable constraints or undefined ambiguities.

This analysis also determined how well our approach reduced choices and identified inconsistencies in relationship to the optimal solution. In addition to the already known variables APC, CPA, and $ChPA$, this analysis also needed to consider the likelihood that constraints evaluated to true $C_{\%=true}$. To understand the meaning of $C_{\%=true}$, consider the following extreme scenarios: If a constraint always returns true, then all potential assignments are positive assignments and no choices are ever reduced (neither optimal nor our approach). On the other extreme, if a constraint always returns false, then there cannot be a positive assignment, which implies that all choices remain unused and are eliminated. Clearly, the likelihood that a

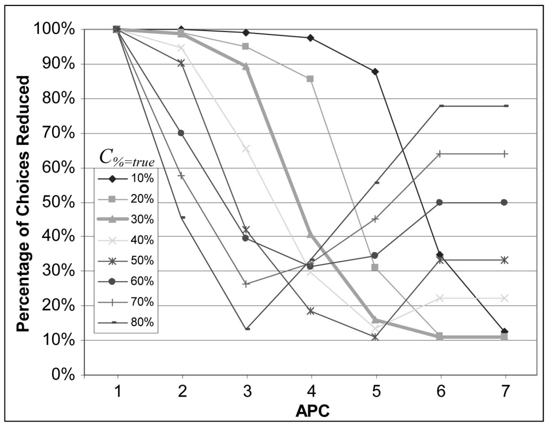


Fig. 11. Effectiveness of our approach in reducing choices with increasing APC.

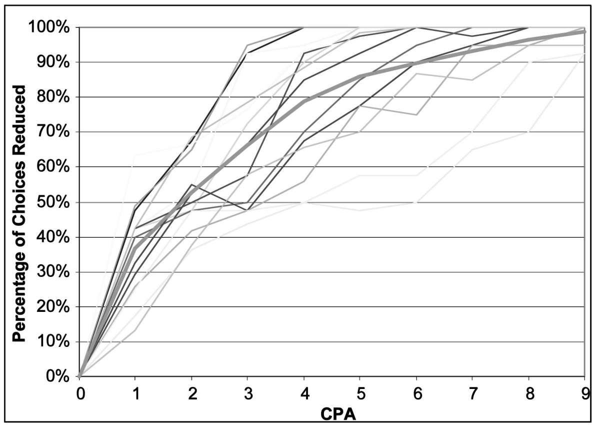


Fig. 12. Effectiveness of our approach in reducing choices with increasing CPA.

constraint evaluates to true affects the number of choices that can be reduced. The lower $C_{\%=true}$, the more choices are eliminated. Our approach's effectiveness is affected by $C_{\%=true}$ and, thus, we considered this factor during analysis.

We made three interesting observations while investigating the effectiveness on the sample models. First, a higher APC reduces the optimality of our approach. Second, the optimality of the approach is not affected by CPA. And third, our approach is always optimal for $APC = 1$. Fig. 11 summarizes these observations where the x-axis depicts APC and the y-axis depicts the percentage of choices reduced in relationship to the optimal. For example, for $APC = 1$, our approach reduced the optimal number of choices. For $APC = 2$, our approach is near optimal if $C_{\%=true}$ is between 10 percent and 30 percent. In the case of the 16,000 constraints of our 27 models, we found that $C_{\%=true}$ is less than 30 percent. For this value, our approach is 99 percent optimal for $APC = 2$ or 89 percent optimal for $APC = 3$. In the context of UML consistency constraints, and given what we know about the expected values for APC and $ChPA$, our approach is thus near optimal (recall that the worst-case APC is below 3 with 10 percent or less ambiguity).

Another interesting observation in Fig. 11 is that our effectiveness increases with higher APC, which is counterintuitive but easy to explain. With higher APC factors, even the optimal approach fails to reduce choices since there are too many combinations such that there are always some positive assignments for every choice. This is further evidence that it is of little interest to the designer to have arbitrary high APC factors. APC should be less than 3 depending on the value for $C_{\%=true}$.

In the reasoning above, our computation of optimality assumed that all constraints are evaluated. However, we know that there is often a small subset of unscalable constraints that cannot be evaluated. These constraints are excluded, but, thus far, we have not discussed how this exclusion affects the effectiveness of our approach in relationship to the optimal approach. If a constraint cannot be evaluated, then it cannot reduce the choices of the ambiguities used in its assignments. This affects CPA in that the fewer constraints are evaluated the smaller it gets (a linear reduction in that half the constraint evaluated reduces CPA by a factor of 2). Fig. 12 depicts the relationship between CPA and the percentage of choices reduced. Again, we derived this data by randomly seeding ambiguities into hundreds of models. We observed that there is a nonlinear growth associated with the reduction of choices: the higher the value for CPA, the less its effect. This is very good for our analysis because it implies that a subset of all constraints will account for the reduction of most choices. For example, $CPA = 5$ reduces 85 percent of all choices, whereas $CPA = 10$ reduces 99 percent of all choices. This is a nonlinear advantage in that half the number of constraints will reduce more than half of all choices. If we account for the small likelihood of large constraints and the small likelihood of unscalable constraints (see Section 6.2), then we can evaluate around 85 percent of all constraints for the 27 sample projects. While this omits 15 percent of all constraints, it still accounts for 97 percent of all reducible choices. This observation has a beneficial side effect: It is of little significance to the overall effectiveness of our approach, if we are unable to evaluate all constraints.

The above discussion solely emphasized our approach's effectiveness in reducing choices. In addition, our approach also identifies inconsistencies. We thus empirically evaluated the effectiveness of identifying inconsistencies and found it to be equal to reducing choices. Since the data is largely repetitive, it is omitted here for brevity.

6.4 Other Domains and Constraints

Our approach was evaluated for a wide range of domains. While the evaluation was limited to the UML class, sequence, and statechart diagrams (arguably the most widely used modeling notations today), it must be emphasized that our approach applies to any modeling situation where CPA and APC are small values that do not increase with the size of the model. This is extremely common, for modelers attempt to reduce complexity to "bite-sized chunks" that people can understand and reason about easily, leading to few constraints per ambiguity and few ambiguities per constraint. Moreover, the sizes of such constraints, C_{size}, is almost always small compared to the size of the descriptions built using the model. We demonstrated that infeasible constraints or constraints that are overly expensive to evaluate—e.g., requiring proofs or infinite domains—can be ignored. We also demonstrated that a small percentage of feasible constraints might eliminate a large portion of the infeasible choices. Thus, even a significant percentage of infeasible constraints may not significantly reduce the effectiveness.

7 Conclusion

We believe that, in general, ***automated tools are incapable of making good (or even adequate) design decisions*** for a designer because such tools do not understand the many factors that affect a good decision—factors involving matters of taste, personal preference, gut feeling, experience, corporate rules, and a range of other issues that are usually not defined formally. While decision factors can be modeled as constraints on the model, it is unreasonable to expect a developer to define a complete set of such constraints.

This paper presented an adaptation of solutions to the constraint satisfaction problem for UML that allows a designer to delay making decisions that affect these (and other) factors by recording design choices. The designer contributes a model with choices and a set of known constraints that the model must satisfy. Of these choices, our algorithm then identifies the choices that are infeasible and should not be chosen based on the known constraints. Our algorithm also identifies dependencies among the choices (and choice combinations) so that the designer understands trade-offs among the remaining choices (the ones that have not been eliminated).

Even though the approach only eliminates a subset of all bad choices, we demonstrated that it is a very complex task nonetheless, because of the many constraint dependencies involved. Despite this complexity, we demonstrated that the following properties are preserved by the approach:

- No valid choice is ever eliminated from the designer's consideration.
- No choice reported to the designer as inconsistent could possibly be consistent with the decisions made to that point.
- Determining what choices should be removed is near optimal.
- The algorithm is computationally scalable.
- The algorithm guarantees all of the above properties despite the fact that it is not able to evaluate all constraints (e.g., undefined constraints or unscalable constraints) or even know about all constraints (e.g., taste and preferences).

While the approach was shown to work well within design suites of a single modern language, it will be applicable in any situation satisfying the parameters discussed in Section 7. Understanding how to make the algorithm incremental in changes to design axes was not considered here—such as removal of constraints or ambiguities. In addition, how to marry the algorithm with various design methodologies—such as stakeholder trade-off or version-based development—has been left as future development topics. We also intend to explore maintenance issues of design choices and design reuse issues in future work.

References

[1] M.L. Begeman and J. Conklin, "The Right Tool for the Job," *Byte*, vol. 13, no. 10, pp. 255-266, 1988.

[2] B. Boehm, A. Egyed, J. Kwan, and R. Madachy, "Using the WinWin Spiral Model: A Case Study," *Computer*, pp. 33-44, 1998.

[3] G. Bruns and P. Godefroid, "Generalized Model Checking: Reasing about Partial State Spaces," *Proc. Int'l Conf. Concurrent Theory*, pp. 168-182, 2000.

[4] E. Börger and D. Rosenzweig, "The WAM—Definition and Compiler Correctness," *Logic Programming: Formal Methods and Practical Applications*, 1994.

[5] A.G. Cass and L.J. Osterweil, *Requirements-Based Design Guidance: A Process-Centered Consistency Management Approach*, Mar. 2002.

[6] K. Dohyung, "Java MPEG Player," http://peace.snu.ac.kr/dhkim/java/MPEG/, 1999.

[7] J. Doyle, "A Truth Maintenance System," *Artificial Intelligence*, vol. 12, no. 3, pp. 231-272, 1979.

[8] S. Easterbrook and M. Chechik, "A Framework for Multi-Valued Reasoning over Inconsistent Viewpoints," *Proc. 23rd Int'l Conf. Software Eng.*, pp. 411-420, May 2001.

[9] A. Egyed, "A Scenario-Driven Approach to Trace Dependency Analysis," *IEEE Trans. Software Eng.*, vol. 29, no. 2, pp. 116-132, Feb. 2003.

[10] A. Egyed, "Instant Consistency Checking for the UML ," *Proc. 28th Int'l Conf. Software Eng. (ICSE)*, May 2005.

[11] A. Egyed and B. Balzer, "Integrating COTS Software into Systems through Instrumentation and Reasoning," *Int'l J. Automated Software Eng. (JASE)*, vol. 13, no. 1, pp. 41-64, 2006.

[12] A. Finkelstein, D. Gabbay, A. Hunter, J. Kramer, and B. Nuseibeh, "Inconsistency Handling in Multi-Perspective Specifications," *IEEE Trans. Software Eng.*, vol. 20, no. 8, pp. 569-578, Aug. 1994.

[13] M. Gertz, "An Extensible Framework for Repairing Constraint Violations," *Proc. Workshop Foundations of Models and Languages for Data and Objects (FMLDO)*, pp. 41-56, 1996.

[14] R. Guindon, H. Krasner, and W. Curtis, "Breakdown and Processes during Early Activities of Software Design by Professionals," *Proc. Second Workshop Empirical Studies of Programmers*, pp. 65-82, 1987.

[15] P. Henteryck, "Strategic Directions in Constraint Programming," *ACM Computing Surveys*, vol. 28, no. 4, 1996.

[16] P. Hudak, *The Haskell School of Expression*. Cambridge Univ. Press, 2000.

[17] IBM, "Rational Rose," http://www.rational.com/, 2006.

[18] D. Jackson, "Alloy: A Lightweight Object Modelling Notation," *ACM Trans. Software Eng. Methodology*, vol. 11, no. 2, 2002.

[19] J. Kleer, "A Perspective on Assumption-Based Truth Maintenance," *Artificial Intelligence*, vol. 59, no. 1, pp. 63-67, 1993.

[20] A.K. Mackworth, "Consistency in Networks of Relations," *J. Artificial Intelligence*, vol. 8, no. 1, pp. 99-118, 1977.

[21] S. Mittal and B. Falkenhainer, "Dynamic Constraint Satisfaction Problems," *Proc. Eighth Nat'l Conf. Artificial Intelligence*, pp. 25-32, 1990.

[22] G. Moerkotte and P.C. Lockemann, "Reactive Consistency Control in Deductive Databases," *ACM Trans. Database Systems*, vol. 16, no. 4, pp. 670-702, 1991.

[23] C. Nentwich, W. Emmerich, and A. Finkelstein, "Consistency Management with Repair Actions," *Proc. 25th Int'l Conf. Software Eng. (ICSE)*, pp. 455-464, 2003.

[24] C. Nentwich, L. Capra, W. Emmerich, and A. Finkelstein, "xlinkit: A Consistency Checking and Smart Link Generation Service," *ACM Trans. Internet Technology*, vol. 2, no. 2, pp. 151-185, 2002.

[25] J. Robins et al., "ArgoUML," http://argouml.tigris.org/, 2006.

[26] J. Rumbaugh, I. Jacobson, and G. Booch, *The Unified Modeling Language Reference Manual*. Addison Wesley, 1999.

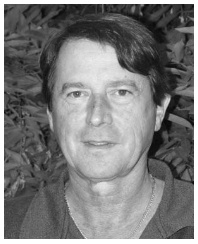
[27] D. Sabin and E.C. Freuder, "Configuration as Composite Constraint Satisfaction," *Proc. First Artificial Intelligence and Manufacturing Research Planning*, 1996.

[28] A. Tsiolakis and H. Ehrig, "Consistency Analysis of UML Class and Sequence Diagrams Using Attributed Graph Grammars," *Proc. Joint APPLIGRAPH/GETGRATS Workshop Graph Transformation Systems (GRATRA 2000)*, pp. 77-86, Mar. 2000.

[29] A. Van der Hoek, D. Heimbigner, and A.L. Wolf, "A Generic, Peer-to-Peer Repository for Distributed Configuration Management," *Proc. Int'l Conf. Software Eng. (ICSE '96)*, p. 308, 1996.

[30] D. Wile, "Program Developments: Formal Explanations of Implementations," *Comm. ACM*, vol. 26, no. 11, 1983.

Alexander Egyed received the MS and PhD degrees in computer science from the University of Southern California in 1996 and 2000. He is a research scientist at Teknowledge Corporation. His research interests include software design and architecture modeling (including model transformation and analysis), traceability among models, requirements engineering, and component-based software development. He is a member of the IEEE, IEEE Computer Society, ACM, and ACM SIGSOFT.



David S. Wile received the ScB degree from Brown University in applied mathematics in 1967 and the PhD degree in computer science from Carnegie Melon University in 1974. He is a senior research scientist at Teknowledge Corporation. Previously, he was a research professor in the Software Sciences Division within the USC/Information Sciences Institute. His recent research interests include the adaptation of COTS tools for the specification and implementation of domain-specific languages, developing formal specification languages for software architectures, and the use of architectural specifications in requirements engineering. He is a member of the ACM, SIGPLAN, SIGSOFT, Sigma Xi, and IFIP's Working Group 2.1 on Algorithmic Languages and Calculi.