Applying Cyc: Using the Knowledge-Based Data Monitor To Track Tests and Defects

Nick Siegel, Gavin Matthews, James Masters, Robert Kahlert, Michael Witbrock, and Karen Pittman

Cycorp, Inc.
3721 Executive Center Drive, Suite 100
Austin, Texas 78731-1615
{nsiegel, gmatthew, chip, rck, witbrock, karen}@cyc.com

Abstract

An application of the Cyc system is described, in which the system contributes to the software engineering effort involved in its own construction. Using its Semantic Knowledge Source Integration (SKSI) facility, Cyc interacts with bug reports tracked using the standard Bugzilla defect management system, performing actions such as post-bug-fix tests, and creating and reopening bugs reports as needed. This is part of a simultaneous effort in the Cyc project to apply software engineering principles (in particular, the use of exhaustive unit testing) to the task of building an intelligent system, and to apply that intelligent system to the automated application of software engineering techniques.

Introduction

For nearly 20 years, researchers at MCC's Cyc Project and at its successor, Cycorp, have been developing a knowledge-based reasoning system called Cyc. Cyc's knowledge base (KB) is huge, containing almost 2 million assertions (facts and rules) that interrelate more than 135,000 concepts. Operations on the KB (additions, modifications, deletions, and queries) are stated using Cyc's formal, declarative representation language, CycL, which is based on second order predicate calculus. Cyc's ability to reason is provided by an inference engine that employs hundreds of pattern-specific heuristic modules, as well as general, resolution-based theorem proving, to derive new conclusions (deduction) or introduce new hypotheses (abduction) from the assertions in the KB. Most of the assertions in the KB are intended to capture "commonsense" knowledge pertaining to the objects and events of everyday human life, such as buying and selling, kinship relations, household appliances, eating, office buildings, vehicles, time, and space. The KB also contains highly specialized, "expert" knowledge in domains such as chemistry, biology, military organizations, diseases, and weapon systems, as well as the grammatical and lexical knowledge that enables Cyc's extensive natural language processing (parsing and generation) capabilities. Among the relatively specialized bodies of knowledge in the KB is a steadily growing ontology of tests called *KB Content Tests* (KBCTs). These tests are designed to check for regression or improvement in Cyc's ability to reason correctly, and since they are fully described by assertions in the KB, they are themselves entities about which Cyc can reason. Cyc also knows about all of the people who have ever added assertions to the KB, and for every current Cycorp employee, Cyc knows the projects on which the employee has worked, the employee's supervisors and supervisees, and (in many cases) particular job-related skills the employee possesses.

Cyc also includes specialized CycL vocabulary, inference modules and supporting connection management code (proxy server, database drivers) that together constitute a facility called *Semantic Knowledge Source Integration* (SKSI). SKSI's CycL vocabulary supports detailed semantic descriptions of external information sources, such as databases and web sites. These semantic descriptions render explicit the entities, concepts, and relations that often are only implicit in a source's implementation data model (e.g., in DBMS metadata), and which link these explicitly represented items into the vast amount of knowledge already in the KB. The inference engine can use this explicitly represented knowledge about external data sources, along with SKSI's specialized inference modules, to access the sources and reason with their data as if they were part of the KB. When presented with a CycL query, Cyc's inference engine is able to dispatch SQL SELECT statements to one or several relevant databases, send form submissions to relevant web sites, search the information contained in the Cyc knowledge base, and combine results obtained from all of these types of sources to provide an answer (a set of bindings).[1] One of the external data sources accessible to

[1] See [Masters 2002] and [Masters and Güngördü 2003] for more information about the development of SKSI. For information about Cyc

Cyc via SKSI is the *Cyc Test Repository* (CTR), a database that contains the archived results of weekly KB Content Test runs.

Cycorp currently uses Bugzilla[2] to track software defects. Cycorp's Bugzilla installation (which is fairly standard, with a few customizations unrelated to the capabilities described in this paper) includes the Bugzilla database, in which data about software defects is stored, and a Perl CGI script that generates a web interface via which users can view and modify the defect data. Since both the web interface and the database's data model have been described in the Cyc KB using the required SKSI vocabulary, Cyc can query or modify existing bug reports, and can open new bug reports. Furthermore, since the CycL vocabulary for defining KB Content Tests includes a predicate for associating one or more specific tests with one or more specific bug reports (#$testQueryForBugzillaBug), Cyc can track and reason about this association.

The integration of Cyc's declarative, semantically rich representation of KB Content Tests with the ability to access, via SKSI, both the archived test results in the Cyc Test Repository, and the defect reports in Bugzilla, has been the near-term development goal of a broader Cycorp effort called *Knowledge-Based Data Monitoring* (KBDM). The next section of this paper describes our KBDM effort in more detail, focusing specifically on the pieces that contribute to the defect tracking framework currently under development, and describing scenarios for its use.

Knowledge-Based Data Monitoring

The primary goal of Cycorp's KBDM effort has been to use Cyc's knowledge base and inference engine to develop software applications (a) that can integrate multiple, structured data sources; and (b) that can monitor, and act upon, user-specified changes (triggering conditions) in the information those sources contain. Over the past two years, our KBDM work has focused mainly on laying the foundation for SKSI, and on developing the CycL vocabulary, schema mapping tools, and conventions (abstraction layers, both in code and in the knowledge base) required to allow Cyc to access external data sources and use their information in inference. We have only recently turned to the task of implementing a prototype monitor application that can check for, and respond to, user-specified changes in the various information sources accessible to Cyc (including both Cyc's own knowledge base, and the external sources described in it). Building a prototype that enables Cyc to monitor its own test results, and to create or modify bug reports associated with specific tests, seemed a worthwhile first step. It exercises SKSI, since it requires Cyc to access an external web site and two external databases as well as the test definitions currently stored in the KB. It also imparts to Cyc a degree of agency, an ability to participate in its own maintenance and improvement, that we intend to extend much further.

In the Introduction, we indicated some of the conceptual and programmatic pieces that make up the KBDM prototype in its current state of primitive nascence. In the next subsection, Architecture, we provide a more complete list of these components, describing the nature and function of each. Then, in the Scenarios subsection, we explain how the components figure in a few specific data monitoring (test and defect management) situations.

Architecture

KB Content Tests. At the time of writing, the Cyc knowledge base contained 2606 KBCTs, with new tests being added at a rate of dozens per week. Each test is a fully reified, declaratively represented instance of the collection (type, class) denoted by the CycL term #$KBContentTestSpecification. Each test definition includes the specification of a CycL query and the results (bindings) that would constitute a successful test run. The current vocabulary makes it possible to state and capture dozens of metrics for each test, such as the total number of bindings returned, the number of bindings returned in 30 seconds, the total time required to retrieve the results that constitute "success", and a token indicating the reason inference halted. Each test definition also includes an assertion identifying the person (or agent) responsible for the test, and (as mentioned in the Introduction) assertions identifying any associated bug reports. Tests are organized (grouped) in a hierarchy of collections, with the primary structuring principle being the Cycorp project (administrative category or research effort) to which a test pertains. Individual tests may be linked to a Cyc project via the CycL predicate #$testResponsibleProject. Entire collections of tests may be linked to a project via the predicate #$testCollectionProjectResponsible. Tests may be run individually, or by collection (*i.e.,* in batch mode). See Figure 1 for an example of a KB Content Test as it appears in Cyc's Query Library interface.

Projects. At Cycorp, the word "project" generally means a distinct administrative category or research effort. Projects are denoted by terms in the Cyc KB, such as #$SKSIProject. Each such project is a member of the collection denoted by the term #$Project. KB Content Tests are typically created for specific projects, and are typically organized in the KB according to the project they serve. Also, projects are associated with Bugzilla products via the CycL predicate #$bugzillaProductForProject.

Cyc Test Repository. The Cyc Test Repository (CTR) is the database in which the results of KB Content Test runs are stored. Each test result is time-stamped, and includes a record indicating the specific test version (tests may be run with varying inference parameter settings, resulting in

---

and its capabilities, see the white papers and other material available at Cycorp's web site, http://www.cyc.com.

[2] For general information about Bugzilla, go to http://www.bugzilla.com. An overview of Bugzilla's data model is available at http://www.ravenbrook.com/project/p4dti/master/design/bugzilla-schema/.

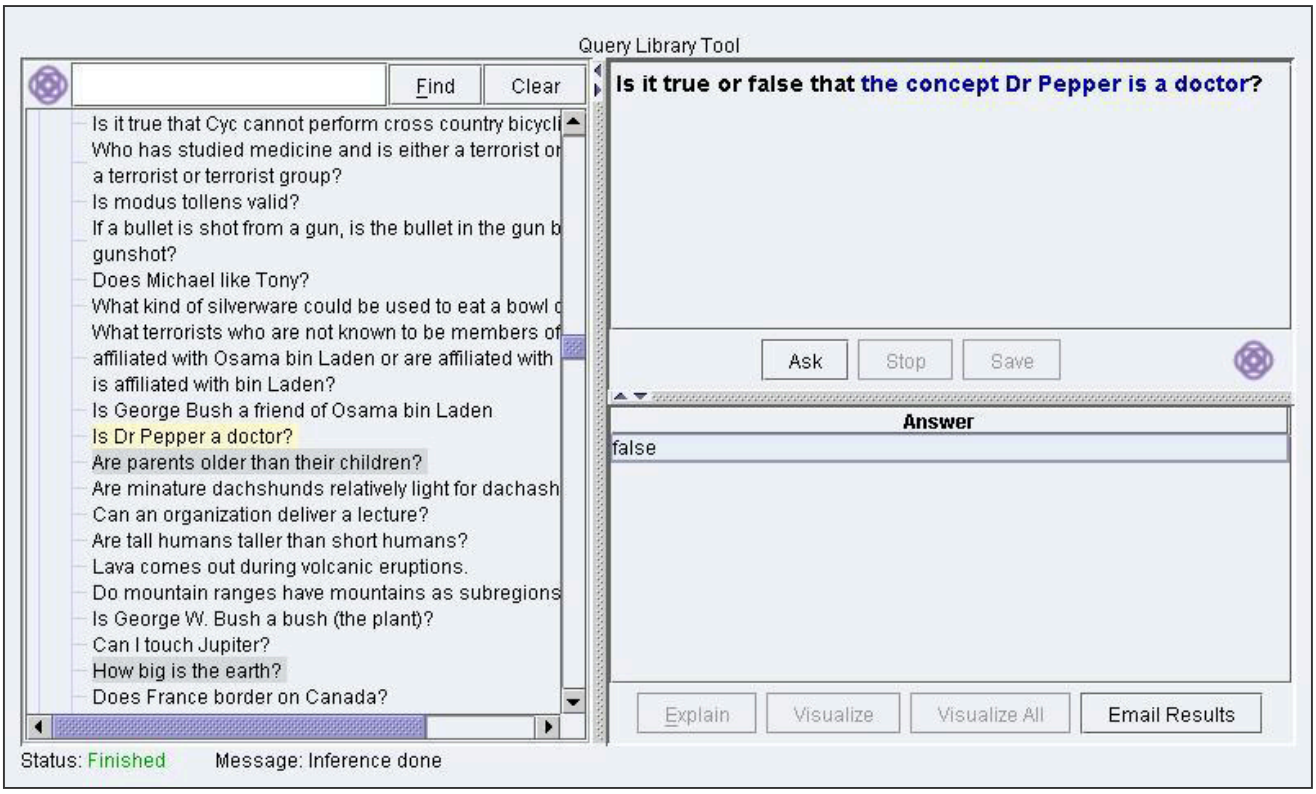

Figure 1: An example of a KB Content Test as it appears in the Query Library interface.

different test versions), KB version, Cyc System version and (when relevant) UI version in which the test was run. Cyc understands the contents of the CTR and is able to query it via SKSI.

Cyc Test Repository Schemata. In order for Cyc to access and use (reason about) an external data source, the source's data model (table schemata and other metadata) must be represented in the Cyc KB with the specialized vocabulary created for the SKSI framework. This has been done for the CTR's data model.

Task Scheduler. The Cyc Task Scheduler is a subcomponent of the Cyc System that enables users to declaratively define and schedule system actions. The CycL predicates

#$taskSchedulerCondition,
#$taskSchedulerAction, and
#$taskSchedulerTaskExpression

make it possible to define task initiation conditions, specify primary task actions, and (if necessary) cause named, parameterized blocks of executable code to be run, by adding assertions to the KB. Other predicates allow the user to fully specify the task's temporal characteristics. One application of the Task Scheduler is the scheduled running of suites of KB Content Tests. In future, the task scheduler will be extended to integrate Cyc's existing process description and planning mechanisms to increase the sophistication with which tasks may be planned and executed.

Cyc Browser, Fact Entry Tool, and Query Library Interfaces. Cyc currently supports several user interfaces. The Cyc Browser consists of dynamic (CGI-generated) HTML pages that allow experienced users to query, browse, edit, and add to the contents of the knowledge base. The Fact Entry Tool is a template-based Java interface that allows even relatively inexpert users to add new facts (ground assertions) to the knowledge base, and edit exsiting facts. The Query Library is a Java interface that allows users to pose pre-formulated queries (such as the libraries of queries already defined for KB Content Tests), or to compose new, arbitrarily complex queries by assembling and editing pre-existing query fragments. All of these interfaces employ Cyc's natural language generation capabilities to render assertions, queries, and query answers in English, shielding users from the underlying, and sometimes dauntingly complex, CycL formalisms. Currently, some KBDM configuration tasks (*e.g.,* Task Scheduler configuration, KB Content Test creation) require use of the Cyc Browser, but the capabilities of the Fact Entry Tool and Query Library are being extended so that they will become the primary interface components for interacting with the KBDM framework.

Bugzilla. As noted in the Introduction, Cycorp's Bugzilla installation includes a web interface (implemented by a

Perl CGI script) and a relational database in which bug report information is stored. The database may be installed on any RDBMS. Cycorp's version currently runs on MySQL.

Bugzilla Schemata. To enable Cyc to access Bugzilla, we take advantage of two different interfaces to the underlying Bugzilla database. To query Bugzilla, we use SKSI to translate CycL query expressions into SQL SELECT statements, which are then dispatched to the database directly through its hosting DBMS (MySQL). However, for the creation or modification of bug reports, we use SKSI to convert declarative "action statements" -- actions defined by special CycL vocabulary items -- into valid HTTP POST requests, which are sent to Bugzilla's Perl CGI interface. The two CycL action predicates supported thus far are #$createABugzillaBugReport and #$updateABugzillaBugReport. CycL terms that denote supported Bugzilla action types include the following:

#$CreatingABugzillaBugReport
#$UpdatingABugzillaBugReport
#$PostingACommentToABugReport
#$MarkingABugzillaBugReportAsADuplicate
#$ClosingABugzillaBugReport
#$VerifyingABugzillaBugReport
#$ReassigningABugzillaBugReport
#$ResolvingABugzillaBugReport
#$AcceptingABugzillaBugReport
#$ReopeningABugzillaBugReport
#$ConfirmingABugzillaBugReport

We use the web interface rather than SQL for DB modifications, because this allows us to take advantage of the Bugzilla application's data integrity enforcement and its mechanism for sending email notifications to relevant parties.

Scenarios

In its current implementation, the KBDM can perform a limited number of actions based on the results of KB Content Tests. It will be easy to increase this number once control of the Task Scheduler is made more fully declarative. What follows is a brief description of the logic and actions currently supported by the KBDM.

At the beginning of a scheduled regression test task, Cyc runs a series of KB Content Tests that are all members of a common collection. Each test, or collection of tests, is associated with some Cycorp project, and possibly with one or more Bugzilla bug reports. Depending on the outcome of each test (success or failure), and on the existence and status of associated Bugzilla bug reports, Cyc may perform a variety of actions to modify a report's regression status. We will consider four of the many possible cases.

(1) If Cyc runs a test that fails and has not yet been associated with a bug report, Cyc creates a new Bugzilla bug report under the appropriate product and component, and posts the test results as the bug's initial description. After the new bug report has been created, Cyc updates the knowledge base to associate the report with the failing test. Note that Cyc currently leaves unspecified several Bugzilla parameters, such as the person to whom the new bug is assigned. Since each component in Bugzilla has a default owner, this information is not required. Future work will permit Cyc to override this and other defaults when it determines such action is appropriate (see Significance and Future Work, below).

(2) If Cyc runs a test that fails, and the test is already associated with a Bugzilla bug report marked as being resolved (by virtue of its having been fixed), Cyc reopens the bug report and posts the test results as a new entry in the report's comment field. Again, in this case (and in those below), Cyc leaves the other optional parameters at their default values.

(3) If Cyc runs a test that succeeds, and the test is already associated with a Bugzilla bug report marked as being resolved (fixed), but which has not yet been marked as verified, then Cyc updates the bug report to mark it as verified.

(4) Finally, if Cyc runs a test that succeeds, and the test is already associated with a Bugzilla bug report marked as being both resolved (fixed) and verified, then Cyc performs no action.

In each of the cases considered above, Cyc needs some background information in order to perform the proper action. In order to create a new Bugzilla bug report for a given test, Cyc needs to know, at minimum, the product and component under which it should create the bug. This information is recorded in the knowledge base in the following way: Each test (or collection of tests) is associated with some Cyc project, and each Cyc project is associated with some Bugzilla product (which is also reified in the knowledge base). Finally, each reified product in the knowledge base has an associated default component. With this data, Cyc can determine the product and component to which a new bug report should belong.

Significance and Future Work

The work on KBDM described above, while tentative and immature, nonetheless suggests new possibilities for both software engineering and the development of intelligent agents.

Limitations of Current Systems

Bugzilla has a number of significant limitations[3], which are shared by other bug tracking systems to varying degrees: meager vocabulary for relating bugs; poor handling of sets of bugs by project or component, especially when components are shared between projects; limited

[3] Cycorp has used Bugzilla for several years and currently runs version 2.14.1; these comments are offered primarily with respect to that release. More recent versions are available, and may offer many improvements. Cycorp also uses CVS heavily, and has explored the use of Microsoft Project.

extensibility for gathering and analyzing information; inflexibility of workflow and assignment; and inadequate prioritization and scheduling.

Bugzilla supports only a simple "blocks" relationship between bugs. Other bug tracking systems support only "related". Experience indicates that a much richer ontology of inter-task relationships would offer significant benefit. These relationships might include: alternative solutions (*e.g.* the "quick hack", and the long term solution); tasks that should share work or co-ordinate on design; tasks to follow-on from or monitor others. Bugzilla also supports a "duplicate" relationship, but this does not satisfy some of the more obvious requirements.

It is sound software engineering practice to reuse components and maximize the code in common between products and projects. The disadvantage is that it is necessary to track the effects of modifications and to identify synergies. Unfortunately, few bug tracking systems provide useful support for this, preferring to model each product as entirely separate, leading users to create "internal" products and deliberately duplicate bugs[4]. In most bug tracking systems, the life cycle overhead of each bug report is non-trivial, and there is therefore a tendency to aggregate multiple tasks in one bug.

Bugzilla does not provide extensive support for the addition and analysis of extra fields, such as: estimated, budgeted or actual effort; customer impact; links to source control or resolution along multiple code branches; detection method; and defect type and cause analysis. The recording and analysis of such information is a crucial part of software process improvement.

Bugzilla does support a four-stage workflow: confirmation, acceptance, resolution and verification. As with any feature of Bugzilla, this workflow can be modified, but not in a convenient or flexible way. It is often necessary to have more complex workflows for specific tasks; the Bugzilla solution is to hand-create multiple bugs and link them together with dependencies. Bugzilla only supports assignment of a bug to one developer and one tester. Software engineering practices like pair programming (which is extensively used at Cycorp) require more flexibility.

Bugzilla provides little support for scheduling work – it does provide target milestones, but there is no direct support for associating work completion with specific dates. In particular, milestones cannot be inherited via dependencies, so the assignee of a blocking task is missing information about its urgency. Bugzilla also does not assist in assigning people to tasks, either by matching skills to suspected defect locations, or by comparing estimated work load to time available.

[4] This is analogous to the problem caused by file systems whose directory structures do not support multiple inheritance, resulting in file duplication or, at best, imperfect solutions such as links or symbolic links.

Using AI to Integrate Process Management Tools

The conventional wisdom is to recommend that such problems be resolved by purchasing a more expensive bug tracking system, or to regard project management and defect tracking as independent corporate functions. Neither of these provides a comprehensive solution. Cycorp's approach is to take Bugzilla as a basic system, and to build additional intelligence on top of it using Cyc, as well as other development support tools managed by Cyc. Another example of our ongoing development is the integration, via SKSI, of our internal "WebCal" calendar system with Cyc. A motivating use case for this is to provide a means to notify developers of milestones against which they have assigned bugs. In time this integration process will evolve a software configuration management system that is powerful in its own right, and that can be integrated into a variety of defect tracking systems. Cyc's rich and extensible ontology can be used to represent arbitrary relationships. Rules can be used to spawn additional tasks as required for any workflow while working to keep down their administrative overhead.

Anticipated Benefits

Current best practice enjoins developers, on resolving a bug, to reflect on several questions: What test would have detected this bug? How can we avoid this type of bug in future? Where might we find similar bugs? (For a slightly different list, see [Van Vleck 1989]). One of the most powerful software-engineering related extensions we believe will be possible with Cyc is the provision of extensive metrics on the development process. We expect that ability to access these metrics using Cyc's existing, general, query mechanisms will provide developers and managers with a straightforward means to answer questions like: What types of bug occur most often? Which code is generating many defects and may require re-factoring? What are the main causes of bugs? How do estimates compare with actual effort? Which developers work together most effectively? What circumstances lead to failure?

In a normal programming development context, actually tracking this information would be difficult and prohibitively expensive. With the introduction of an adequately detailed ontology of potential software defects, and knowledge about the development team, it may become possible to represent and use this information for process improvement, planning future development, training, and self-improvement. The ability to represent and reason about arbitrary meta-knowledge will also support other software engineering practices such as formal inspection and process improvement.

By modeling defect characteristics, the varying levels of expertise among the members of a development team, and the progress each team member is making, an intelligent workflow system could arrange for a developer who repeatedly makes a certain type of error to be paired with a

more highly skilled person who never makes that error. The defects understood to be most urgent, or to have the most dependents, could be matched with the programmers having the most appropriate skills and requiring the shortest preparation time.

Early in the development of Cyc, we realized that it would be necessary to forgo both complete reasoning and global formal consistency. The size and complexity of the knowledge base (especially when external knowledge sources are available) precludes complete reasoning in most cases, and in particular, precludes an exhaustive determination of consistency. The Cyc ontology permits a modeling of general common sense rules, which are overridden for special-case exceptions; this argumentation process provides some robustness against inconsistencies. Moreover, the assertions in the knowledge base are distributed among hundred of formal contexts, or *microtheories*, each of which is internally consistent. Cyc reasons for itself about whether each new assertion is consistent with the facts that are visible (accessible) within a given microtheory, and many common types of error or inconsistency can readily be detected and sometimes automatically repaired.

There are clearly inherent weaknesses in using a system to test its own behavior. It is harder to ensure that tests are not subject to the same flaws as are being tested. At the same time, this significant in-house use of Cyc for testing and other purposes is a vital day-to-day driver for the quality of the system. Cyc's rich representation and powerful deductive capabilities support a sophistication of testing approach that would be impossible with any less powerful external mechanism. It should also be noted that, because the tests and related information are represented purely in CycL, it is not necessary for knowledge engineers using Cyc to become familiar with any new system or language, but instead it is possible for them to use the same techniques as are used directly for AI research.

Conclusion

A powerful knowledge-based system, such as Cyc, can contribute to software engineering at two distinct levels: it can support and extend tools that implement software development processes, and it can model and reason about the process itself and its motivation – at a level essentially absent from current systems – to assess the effectiveness of its implementation and identify shortfalls. By automating the integration of existing corporate tools, and providing additional representational and computational facilities outside the scope of those tools, we believe that AI techniques can substantially improve both the efficacy and flexibility of software engineering practice. Not insignificantly, this improvement can contribute to speeding the development of AI systems.

Acknowledgements

Support for the development of SKSI and KBDM was provided by a Small Business Innovation Research grant (GCN: F30602-02-C-0033), which was administered by the Air Force Research Laboratory in Rome, New York. Support for the development of SKSI, and for the production of this paper, was provided by ARDA's NIMD program (GCN: 2003*H264900*000).

References

Masters, J., and Güngördü, Z. 2003. Structured Knowledge Source Integration: A Progress Report. In *Proceedings of the International Conference on Integration of Knowledge Intensive Multi-Agent Systems (KIMAS '03)*, 562-566. Piscataway, N.J.: IEEE Press.

Masters, J. 2002. Structured Knowledge Source Integration and its applications to information fusion. In *Proceedings of the Fifth International Conference on Information Fusion,* 1340-1346. Sunnyvale, CA: International Society of Information Fusion.

Van Vleck, T. 1989. Three Questions About Each Bug You Find. *ACM SIGSOFT Software Engineering Notes* 14(5):62-63.