Fourth IEEE Workshop on Program Comprehension, Berlin, March 1996

The Gadfly: An Approach to Architectural-Level System Comprehension

Paul Clements, Robert Krut, Ed Morris, Kurt Wallnau
{clements, rk, ejm, kcw}@sei.cmu.edu
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Technology to support system comprehension tends to reflect either a "bottom-up" or "top-down" approach. Bottom-up approaches attempt to derive system models from source code, while top-down approaches attempt to map abstract "domain" concepts to concrete system artifacts. While both approaches have merit in theory, in practice the top-down approach has not yielded scalable, cost-effective technology. One problem with the top-down approach is that it is very expensive to develop domain models, and it is difficult to develop models that are sufficiently general to be applied to multiple systems (and hence amortize the development cost). This paper describes the Gadfly, an approach for developing narrowly-focused, reusable domain models that can be integrated and (re)used to aid in the process of top-down system comprehension.

1. Introduction

A primary purpose of program understanding technology is, ultimately, to assist maintainers to develop a system-level understanding of an application so that changes to the application can be introduced in a rational, consistent way. Unfortunately, although source code is often the most reliable arbiter of what a system does, it does not reflect all of the attributes of an application necessary to develop a true system-level understanding: there is more to understanding a system than understanding what function it computes. System characteristics such as performance, robustness, security, etc., must also be understood. We refer to such characteristics as quality attributes, and quality attributes are related more to the architecture of a system than to its code [16].

This paper describes a knowledge-based software assistant called the Gadfly. The Gadfly is intended to help designers *create* applications that attain a selected set of quality attributes[1], as well as to help maintainers *understand* how an existing application has achieved those properties. The construction and understanding guidance of the Gadfly is at the *architectural* level, which deals with allocation of functionality to components and inter-component interaction, rather than the internal workings of individual components. To motivate the architectural approach to system comprehension based on quality attributes, the paper makes the following points:

- Software architecture provides a level of understanding at which a system's quality attributes can be best managed and understood, because they most often depend on inter-component relationships and cannot be discovered from source code alone.
- Quality attributes represent coherent domains of specialized design knowledge that can be separately modeled and combined in different ways to support both forward engineering and system comprehension.
- Quality attribute knowledge is similar to the knowledge represented by specialized design schemas and similar concepts found in program comprehension literature, and is amenable to knowledge representation modeling.
- Software architecture can be used both as a framework for integrating sets of quality attribute domain models, and as a juncture between top-down and bottom-up strategies for program comprehension.

The rest of the paper is structured as follows: Section 2 provides an overview of current approaches to program and system comprehension, and describes the program understanding context for the Gadfly. Section 3 surveys the key concepts of software architecture, and outlines the potential role software architecture can play in both forward-engineering and system comprehension activities. Section 4 describes the Gadfly, and illustrates its use through an operational scenario. Section 5 summarizes the key contributions of the Gadfly to program comprehension, and outlines potential next steps.

1. The Gadfly prototype addresses only the information security attribute.

2. Program comprehension technology

2.1 Top-down and bottom-up approaches for system comprehension

Current program understanding models identify a number of different types of knowledge that a maintainer uses to comprehend software, including knowledge of programming, knowledge of the real-world situation represented in the software, and knowledge of the application domain. All of these types of knowledge are important to the maintainer, since they embody different abstractions and impart different kinds of understanding of software systems. These kinds of knowledge are used in a top-down or bottom-up manner, or some opportunistic combination of these approaches.

The bottom-up model of program comprehension suggests that a model of the application is built starting with program knowledge and works to produce higher abstractions, using strategies like "chunking". Program knowledge[1] reflects the maintainers' understanding of programming idioms, program structure, algorithms, and flow of control and data (the programming domain), and is typically related to a bottom-up approach.

Alternately, a top-down model of program comprehension suggests that comprehension proceeds from higher level abstractions down to lower level program idioms, algorithms, etc. Pennington [1] suggests that, in addition to program knowledge, expert maintainers also rely on an understanding of the real-world problem addressed by the software in order comprehend a particular software system. This world knowledge is referred to as a situation model, which describes the problem domain from a higher level of abstraction than program models. According to Pennington, program comprehension involves employing both bottom-up and top-down strategies to relate and coordinate information from the program model with that of the situation model[2].

Brooks suggests that, in order to comprehend a programming problem, experts employ a top-down, hypothesis driven problem solving approach[3]. In applying this approach, Soloway and Ehrlich found that expert programmers employ high-level schemas (plans) that strongly influence expectations about what a program should look like[5]. Koenemann and Robertson demonstrated that for experienced programmers, program comprehension occurred primarily in a top-down manner using such schemas; however, programmers resort to bottom-up strategies when they lacked hypotheses, when hypotheses failed, or for close scrutiny of relevant code[4]. Subjects determined what program segments were relevant based on their knowledge of the task domain, general programming knowledge, and their current understanding of the program. Letovsky suggests that program comprehension can best be viewed as an opportunistic application of bottom-up and top-down strategies[7].

Guindon, Curtis, and Krasner also addressed the question of opportunistic system comprehension, but from the standpoint of the design of highly complex systems[6]. In experiments requiring the design of logic to control the functioning of lifts (elevators), the authors found that the primary determinant of performance was the presence (or lack) of computational techniques, called specialized design schemas, that correspond to characteristics of the application domain. These specialized design schemas encode a solution template and the situations under which the solution is appropriate. Examples of specialized design schemas employed by the experiment subjects included scheduling and routing, message communication, and concurrency. Guindon, etl al., found that subjects applied these schemas in a highly opportunistic manner, building partial solutions at various levels of abstractions.

Clearly, program comprehension relies on the application of a set of domain models (variously called specialized design schemas, situation models, program models, etc.). We expect that program comprehension relies on processing analogous to those suggested by Guindon for design.

2.2 Approaches to building domain models

Current tool support for the application of domain knowledge to aid in program comprehension suffers from several limitations. While a number of approaches to codifying domain knowledge have been developed (e.g., [9][10]), to date they have demonstrated only limited success. Nor are tools based on source code parsing sophisticated enough to produce domain models or recognize architectural designs within systems.

The existing approaches to supporting the maintainer by supplementing their domain knowledge can be classified into two broad categories:

- approaches that attempt to automatically extract the high-level domain knowledge from source code and other system artifacts; and,
- approaches that attempt to codify and organize the knowledge of experts about specific systems.

The former approach (automatic extraction of high-level abstractions from source code) has proven difficult. For example, automatic recognition of algorithms is complex due to the wide variance in the manner in which a specific algorithm can be encoded, and the huge volume of code in which the algorithm may be embedded. In addition, this approach still requires an expert maintainer to relate any algorithms found to domain concepts.

The latter approach (building knowledge bases from expert input) has led to a number of interesting tools that

provide some support for software maintainers. However, the effort necessary to create the knowledge base is extremely high, relying on time-consuming interviews involving system experts and often "knowledge engineers" who specialize in the organization of knowledge into appropriate rules.

In addition to being expensive to develop, the completed knowledge bases are inflexible and hard to maintain [11]. They mix information that spans multiple views or abstractions in a system (e.g., algorithms, architecture, requirements), domain knowledge that crosses multiple software domains (e.g., security, fault-tolerance, distribution, performance), knowledge unique to the application domain (e.g., banking, health informatics, command and control), and knowledge specific to a single system (e.g., a specific air-traffic control system). It is hard to see how information within the resulting knowledge bases can be generalized to other systems within (or outside of) the application domain. Thus, their heavy development cost cannot be amortized across other applications.

2.3 A new approach for knowledge-based system comprehension

In this paper, we suggest a new approach to developing domain knowledge to support system understanding, and describe a prototype implementation of this approach, called the Gadfly. The Gadfly is based on these premises:

- There is a strong symmetry, largely unexploited to date, between developing a system and comprehending it after the fact. Comprehension seeks to understand the artifacts produced during construction. Hence, the knowledge structures that served to guide the construction tend to be the same ones that provide the framework against which the legacy artifacts can be understood.
- Systems are comprehended, at least in part, from the vantage of codifiable domains of knowledge. The Gadfly recognizes that more than one kind of domain applies to a system. For example, to build a secure command-and-control system requires knowledge about command-and-control systems as well as methods for achieving security in computer systems. These domains may be orthogonal in many ways; in any case, knowledge about them can be separately modeled and combined in different ways to reveal different aspects of a system under investigation.
- Just as domain knowledge can be partitioned into different kinds of expertise (e.g., security, fault-tolerance, command-and-control), so, too, it can be partitioned and mapped to systems in terms of different views or kinds of understanding (e.g., code, architecture and problem statement views). Thus, system comprehension involves understanding a system, through various abstractions, in terms of different kinds of domain knowledge.
- The architecture of system is an abstraction that is particularly fruitful as a basis for system comprehension, and the concepts of software architecture can provide a foundation for structuring the investigation of a system, and for integrating supportive domain knowledge.

The Gadfly is a system that guides its user through an analysis of a system, based on separate knowledge bases dealing with the application domain and relevant system quality attributes. The prototype Gadfly was built to render analytical assistance with secure command-and-control systems; hence, it was armed with one knowledge base about kinds of command-and-control systems, and a second knowledge base about computer security.

3. Software architecture and comprehension

Software architecture refers to a view of a system that focuses on the nature and interactions of the major components. While not a new concept— the fundamental notion dates back at least to 1968 when Dijkstra pointed out that carefully structuring a system imparts useful properties and should be considered in addition to just computing the right answer [13]—software architecture as a topic of study is enjoying a flurry of interest. See, for example, [14].

A software architecture represents the integration of application domain concepts with system design expertise to ensure that the application will meet (or, in the case of program understanding, how it has met) its requirements. System design expertise is used to make (or understand) design trade-offs, e.g., performance vs. modifiability or security vs. ease of use. These and other quality attributes are manifested at the architectural level of systems, and cannot be discerned or analyzed from individual system components.

More generally, an architecture represents a body of knowledge with multiple uses for both the designer and maintainer:

- Architecture enables communication and can be used to convey the decisions of designers to maintainers.
- Architecture represents a transferable abstraction of a system that can be applied to other systems exhibiting similar requirements. *Domain-specific software architectures* describe the features of a family of systems [15].
- Architecture suggests a recipe book for designers and maintainers to assist them in selecting and identifying the design idioms that guide the organization of modules and subsystems into complete systems.
- Architecture simplifies system construction and guides program understanding by acting as a framework that constrains the manner in which components interact with their environment, receive and relinquish control, manage data, communicate, and share resources.

• Architecture enables a system to satisfy its quality attributes. For example, modifiability depends extensively on the system's modularization, which reflects the encapsulation strategies; performance depends largely upon the volume and complexity of inter-component communication and coordination, etc.
• Architectural constructs are institutionalized in the development and maintenance organization's team structure, work assignments, management units, etc. Therefore, crucial information about the social context of a system, vital for understanding, is embodied in its architecture.

Most research in software architecture has tended to focus on forward engineering. Architecture description languages (ADLs) continue to be an active area of research [12]. The key challenge for ADLs is to express the unchanging characteristics of a system in addition to describing allowable variation. Closely related to work on ADLs is research on automated composition of systems from architectural models [17][18]. Architecture-level composers tend to view system building as an exercise in constraining the variability of an underlying design until no variation remains and the result is an executable system.

In contrast, relatively little research has been undertaken to understand how software architecture can be used to aid in system comprehension. The software architecture analysis method (SAAM) [16] is an architecture comprehension technique that designers can use to validate that design decisions support selected quality attributes. SAAM is essentially a guide to architecture-level comprehension if quality attributes. However, SAAM is focused mostly on the methodology for comprehension; there is less emphasis on codifying design heuristics associated with any particular quality attribute.

The Gadfly draws upon advances in both areas of software architecture research (forward engineering and comprehension) by recognizing that the kinds of knowledge needed to compose a system are, by and large, the same kinds of knowledge needed to comprehend an existing system. The kinds of analysis a designer subjects a hypothetical design are similar to the analysis of operational (fielded) designs, whether the intention is to perform a design trade-off for a particular quality attribute (forward engineering), or to discover the presence of a quality attribute (system comprehension).

4. The Gadfly

The Gadfly prototype is a knowledge-based software assistant (KBSA)[2] that supports the development and comprehension of command, control and communications (C3) systems. These functions are supported in this way:
• Development: portions of command centers[3] can be semiautomatically composed from components and a generic command center architecture.
• Comprehension: specific command center designs can be evaluated from an information security perspective.
• Integrated composition and comprehension: comprehension services may be invoked from composition services to provide guidance in the composition process.

We first describe the knowledge and computational models used by the composition function of Gadfly, since these models are used (though extended) by the comprehension function. We then describe the overall Gadfly architecture and how the composition and comprehension functions interact. Finally, we annotate a sample session using the Gadfly for system comprehension purposes.

4.1 The Gadfly computational model

The composition function of the Gadfly is built upon a domain model—a model which describes, in this case, the structure and operational context of command centers. The command center domain model is represented in RLF [20], which employs a structured-inheritance network (similar to Brachman's KL-ONE [21]) and a specialized forward-chaining rule-based inferencing system. The domain model includes descriptions of command center tasks (e.g., situation monitoring and threat assessment), links between these tasks and architectural components in a command center (e.g., geographic information system) and links from architecture components to specific technologies (e.g., DeLorme mapping system).

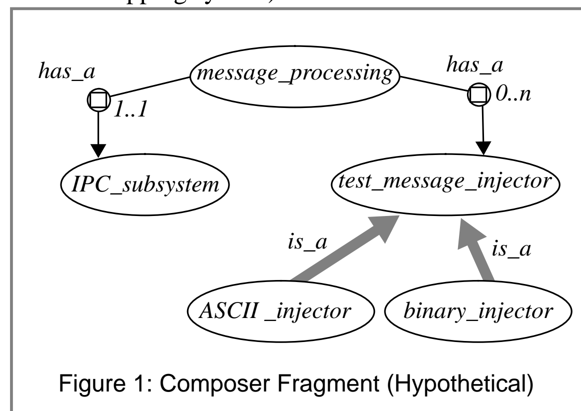

Figure 1: Composer Fragment (Hypothetical)

The composer allows command center designers to interactively develop portions of command centers through a refinement process: navigating among, and converging decision points in a domain model. These decision points represent various alternatives in the design and implemen-

2. In general, a KBSA is an application that uses deductive reasoning to provide expert assistance to humans engaged in knowledge-intensive activities.

3. A specific (headquarter) function within a C3 system.

tation of a family of command centers described by this domain model. The composition process is strongly analogous to various hardware composition systems developed in the 1980's [19].

To illustrate the knowledge and computational models of the composer, consider the simplified fragment of the command center domain model illustrated in Figure 1. This fragment represents a small portion of the generic architecture encoded within the domain model. It asserts that the message processing component of the architecture has exactly one inter-process communication (IPC) system and zero or more components for injecting test messages into the IPC subsystem. Further, there are exactly two kinds of injectors: one injects ASCII-encoded messages, one injects binary-encoded messages.

The composer works by navigating through such network models, asking questions pertinent to the current "focus" (the semantic network concept it is examining) of the composer, and acting upon these answers. At the point when the focus of the composer is at the message processing system, for example, the designer might be asked whether a test message injector is desired, and, if so, how many and of what kind.[4] Similar questions might be asked about the IPC subsystem, for example if the model described specific products that could provide this functionality. As the designer answers questions, the composer emits an instantiation of the generic model (a refinement) to record the decisions made by the designer and any consequences of these decisions; in some cases it can also emit build scripts for automatically constructing prototype systems.

We refer to the semantic network (as illustrated in Figure 1) as encoding structural knowledge. Extra-structural knowledge is also encoded in the model as different types of rules that are linked to the structural model. Rules are used to capture domain knowledge not easily encoded in a semantic network, and are used to propagate design decisions through the network. For example, the decision to select a binary injector might be made automatically if an earlier design decision determined that the class of messages processed by the command center included binary messages; this, in turn, could have been deduced (and propagated) from a still-earlier decision regarding the mission of the command center (also modeled in the domain model).

We developed a proof-of-concept composer based upon the model just described. However, we discovered that application domain knowledge alone was an insufficient foundation for the composer. While the domain model described alternative components and compositions, it provided little engineering guidance on how to select among these alternatives. Frequently, such decisions could be made on the basis of desired quality attributes. To help designers make such decisions, knowledge about these quality attributes and how they can be achieved by different design decisions must also be consulted. The Gadfly prototype is an extension of the original composer that augments the application-specific domain model (C3) with quality attribute domain knowledge.

4.2 The Gadfly architecture

The initial customer for the Gadfly was concerned with evaluating systems (proposed and existing) from an information security perspective. They had already developed a domain model of information security principles to aid in analysis and evaluation efforts, but found the model difficult to employ because it lacked an application-specific context. This problem was the complement to limitations of the composer prototype, which had an application context but lacked quality attribute models.

The purpose of the Gadfly prototype was to demonstrate the re-use of security expertise for designing new systems, and for evaluating existing systems, from a security perspective. The Gadfly architecture reflects the integration of application-domain knowledge with different kinds of highly-specialized design knowledge; it also reflects our contention that there is a symmetry between system design and system comprehension, and that a single technology can accommodate both kinds of activities.

Similarly to the C3 domain model, the information security domain model was encoded in a structural model augmented with extra-structural rules. The structural model encodes information such as:

- a threat model, which describes a range of potential security threats that confront systems, e.g., disruption, deception and disclosure; each threat is the root of its own taxonomy (e.g, there are many kinds of disruption);
- a security service model, which describes basic classes of countermeasures for meeting various threats, e.g., hardware redundancy, cryptographic checksum, password protection; and,
- a security mechanism model, which describes and links various "approved" mechanisms that may be useful for implementing all or part of one or more security services.

Extra-structural rules encode procedural knowledge, referred to as strategies in [8], for applying this knowledge in specific contexts. These strategies include the kinds of information that security analysts will seek regarding the operational and maintenance context of a system, as well as concrete analysis processes, such as mathematical models for deriving the seriousness of a threat (for example, bal-

4. The following questions can be deduced from the structure of the model. Other questions, derived from extra-structural knowledge, might also be asked.

ancing factors such as the potential gain for the intruder, the damage incurred by the system, the risk of detection to the intruder, and the cost of detecting the intruder).

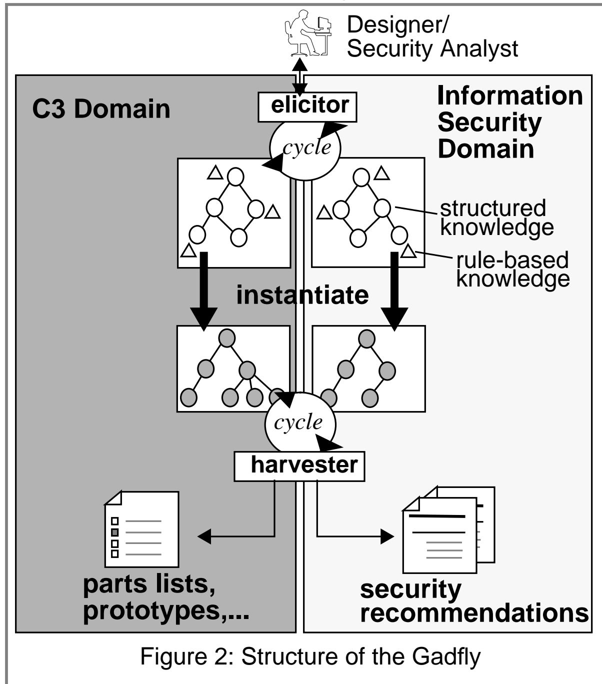

Figure 2: Structure of the Gadfly

The Gadfly architecture is illustrated in Figure 2. The elicitor (the top-most box in Figure 2) is the function that manages the dialogue between the Gadfly and the designer. To conduct this dialogue the elicitor needs to modulate between C3 domain knowledge and information security domain knowledge. The elicitor "walks" the structural models, asking questions depending upon rules and facts associated with various concepts in the structural model, emitting instantiations of concepts where appropriate, and shifting focus to new nodes in the structural model. Links between concepts instantiated from the C3 domain model and information security domain model represent the assignment of security concepts to the application architecture[5]. The process continues in cyclic fashion (fire the rules, ask questions, shift focus in the network) until the session is complete (no more nodes to visit or questions to ask).

The Gadfly can be used for system composition, in which case security knowledge can be consulted as a means of determining, for example, which components to select for a given system. Alternatively, Gadfly can be used when for constructing a cognitive model of security concepts within an existing system. Moreover, it is not even strictly necessary that the system description be encoded in a domain model to use the Gadfly in this way. If an architectural model did not exist for a system, the execution of the Gadfly would result not in an assignment of security concepts to an architectural description, but rather in a framework for investigating the system from a security perspective. That is, the questions asked by the elicitor and the instantiated security model generated from the dialogue would provide a basis for further investigation of the system using whatever system artifacts are available (code, design notes, or the developers themselves). In effect, then, the Gadfly helps maintainers by allowing them to re-use highly specialized system comprehension strategies [8].

4.3 Annotated output from a Gadfly session

The ten-page report that forms the basis of the following annotations was generated from a session in which an analyst was using the Gadfly to investigate the security properties of the message processing component of the command center architecture.[6] There are six sections of the Gadfly-generated report (not counting a prologue which provides context information on the report itself), each illustrated in turn in Figures 3a through 3f.[7]

The message processing component is itself an aggregate concept comprised of several kinds of components, including: message translators and validators, interprocess communication components, message generation components, human-machine interface components, etc. In the following scenario, specific off-the-shelf components that implement these functions had already been selected. Thus, the scenario reflects a comprehension task: the analyst is attempting to infer security properties of a design where several key decisions have already been made.

Figure 3a reflects a security prioritization scheme for the particular system under investigation. This information represents requirements and design assumptions for the command center: comprehension of more detailed security properties (and the relationship of these properties to other aspects of the command center design) is not possible without this kind of information.

This is an important feature of the Gadfly: it addresses information that is best specified (or found) in architectural-level specifications, i.e., issues of system and component context and inter-component relationships. Since this information is not likely to be found in code, this aspect of Gadfly reflects the reuse of a system comprehension strategy: the application of the strategy produces a framework

5. The prototype did not go so far as to create these links, since the operational concept of the composer was that instantiated networks were transient, and existed only so long as needed by the harvester. The links exist conceptually, and are illustrated in the annotated report generated by Gadfly (Section 4.3 of this paper). The generalization noted here is easily achieved, however.

6. The report corresponds to "security recommendations" depicted in Figure 2.

7. The content of the report has been edited slightly for formatting purposes.

for investigating security properties of the command center in question.

Figure 3b summarizes the specific threats to which this

```
You specified the following sets of
threat consequences as being the most
important to counter:

* disruption via incapacitation
* disruption via corruption
* disruption via falsification
* disclosure via intrusion
* disclosure via interception
* disclosure via exposure
```

Figure 3a: Threat Context and Prioritization

command center must respond. As was stated about the threat context and prioritization information depicted in Figure 3a, this information reflects design context; however (as will be illustrated) this information provides a basis for concept assignment of specific security threats to specific components in the command center architecture.

Figure 3c summarizes aspects of a command center that

```
Specific threats most concerned about:

*disclosure: intrusion penetration
*disclosure: interception scavenging
*deception: falsification insertion
*deception: falsification substitution
*disruption: corruption tamper malicious
*disclosure: intrusion cryptoanalysis
(etc.)
```

Figure 3b: Known Threats

might be associated with system-level documentation, but seldom with software-level documentation: the physical environment in which the software will execute. This information, too, is crucial for comprehending the security aspects of the software.

```
You specified the component would operate
in the following environment/context:

The factor: has the attribute(s):

component_info:
   source code available to: nobody
   outside net connection: satellite
physical site:
   network components in: unsecure area
   component housed in: secure area
   spot checks by guards: not performed
(etc.).
```

Figure 3c: Physical System Context

Armed with this context information (Figures 3a-c), the Gadfly can proceed with the task of assigning security concepts to elements of the command center. Further, the Gadfly can *infer* new threats not explicitly specified by the analyst. Figure 3d is an excerpt of the security concepts directly assigned to command center components.[8] Figure 3e is an excerpt of the threats inferred from the system context. These inferences result from sometimes subtle interactions between environmental context, threat priority and component attributes. These inferred threats are then assigned to the appropriate components.

```
Threats for component: DEC_Message_Q:
   deception->falsification->substitution
   disruption->corruption->
      mallicious_logic_corruption
   disclosure->interception->wire_tapping
(etc.)
Threats for component PRISM_MTV:
   deception->falsification->substitution
   disclosure->interception->penetration
(etc.)
Threats for component ASCII_PRISM_MSG_GEN:
   deception->falsification->insertion
(etc.)
```

Figure 3d: Threat (Concept) Assignment

Finally, the Gadfly is able to derive a set of security services that should be present in a system if it is to meet the assigned threats. As in Figures 3d and 3e, the Gadfly is able to make a direct assignment of security concepts (services, in this case) to components: it is also able to infer the need for additional services. For brevity, only the former is illustrated in Figure 3f.

As noted earlier, the information security domain model underlying the Gadfly also maps security services (Figure 3f) to approved security mechanisms (e.g., software components). As a result, the kinds of mechanisms needed in the architecture to achieve a specific set of quality attribute objectives (security in this case) have been identified; the identify of these mechanisms can be used as a basis for a more fine-grained pattern matching within the code (e.g., search for cryptographic or password services in code).

5. Conclusions

5.1 Gadfly Contributions

The Gadfly is a knowledge-based assistant for helping designers create command centers, and for helping security

8. DEC_Message_Q, PRISM_MTV, etc., are the names of specific off-the-shelf components used to implement this instantiation of the command center architecture.

```
Since:
   component_info:
      component on network machine:
      outside of building
   and you are worried about disclosure
infer new threat:
   disclosure->exposure->logic_tapping

Since:
   component_info:
      component on network machine:
      outside of building
   and you are worried about disclosure
infer new threat:
   disclosure->intrusion->
      reverse_engineer
(etc.)
```

Figure 3e: Inferred Threats

analysts comprehend the security properties of existing (and perhaps evolving) command center systems. The Gadfly makes three separate but related contributions to program understanding: a focus on architecture-level specifications, a partitioning of domain models into separately-modeled and individually-selectable knowledge bases, and a demonstration of the symmetry between system design and system comprehension.

```
A primary service for disclosure->
   corruption->tamper_malicious
for DEC_Message_Q component
   is data_redundancy

A primary service for deception->
   falsification->substitution
for DEC_Message_Q component
   is password
(etc.)
A primary service for disclosure->
   intrusion->reverse_engineering
for ASCII_PRISM_MSG_GEN
   is access_control
(etc.)
```

Figure 3f: Service Assignment and Inference

Architecture is the appropriate locus for specifying and comprehending system-wide properties. Continuing with security as an example, a component that is susceptible to logic tampering may represent a vulnerability in one system, but if it is enclosed within a more secure component (in inter-component relationship) or within a secure operating environment (a system boundary relationship), then it will not be a vulnerability. Thus, the property of vulnerability needs to be assigned to a specification of the system at a level that spans individual components: namely, the architecture level.

The second contribution of Gadfly—separable domain models—is as much an economic contribution as it is a technical one. The idea of developing separable, reusable domain models is not new—it is a founding principle of the Knowledge Sharing Initiative, which is developing techniques for creating “shareable ontologies” [22]. The economic and technical justifications for shareable ontologies are strong: cost amortization, community standards, evolutionary refinement of shared models, etc.

While we are not suggesting that the information security model is a shareable ontology—it lacks some of the characteristics specified by [22] that would make it one—we do claim it plays the role of shareable ontology within the Gadfly system. That is, constraining Gadfly domain models (currently, information security and C3) in various ways makes it possible to develop domain models that are focused on, for example, comprehension strategies and concept assignment to architectural components (as opposed to lines of code).

Thus, it is not hard to envision generalizations of the Gadfly that would allow designers to consult construction or comprehension strategies focused on fault-tolerance, distribution, real-time performance, or other quality attributes of systems. The development of specialized domain (comprehension strategy) models is more economically feasible than developing one-of-a-kind, system-specific mixed-content domain models that do not easily transfer to new applications.

Finally, the Gadfly demonstrates that the same kinds of human expertise needed to design systems are also needed to comprehend systems. Although design requires a synthesis of many kinds of expertise, system comprehension can be (and in practice often is) narrowly focused to the search for specific kinds of properties. The Gadfly demonstrated how one technology framework could re-use knowledge for both constructive (forward-engineering) and de-constructive (reverse-engineering) activities.

5.2 Future Direction

Although the Gadfly architecture admits the possibility of integrating arbitrarily many domain models to support construction and comprehension of systems, the current system requires that the elicitor have knowledge of the specific knowledge-bases being employed. Ideally, the elicitor would be able to independent of domain models. However, while it might be simple to implement this feature, it is equally important not to subject designers to “information overload.” Some way of pruning or focusing the dialogue will be important, and this will be more difficult to accomplish. Similarly, modeling and managing the interaction

between domain models (e.g., distribution and fault tolerance) will also be difficult, as these interactions imply trade-off reasoning that may be difficult to formalize.

A more practical extension of the Gadfly would be the development of domain models covering other kinds of quality attributes. While some work has been done to formalize static quality attributes such as modifiability, it would be interesting to see if this work could be formalized in such a way that it could be used by the Gadfly. Similarly, design heuristics for narrow ranges of issues such as real-time and fault tolerance could also be developed.

Acknowledgments. Special credit to: Mark Simos (Organnon Motives), who originated the Gadfly concept in the mid 1980's; Paula Matuszek (Loral), whose expertise in reasoning systems made the latest Gadfly possible; and Brian Koehler (US. Government) for his security expertise. The SEI is sponsored by the US Department of Defense.

References

1. Pennington, N. Comprehension strategies in programming. In G. M. Olson, S. Sheppard, and E. Soloway (editors.) Empirical Studies of Programmers: Second Workshop. Norwood N.J. Ablex Publishing Co. 1987. pp 100-112.
2. Pennington, N., Grabowski, B., The Tasks of Programming. In Psychology of Programming, J.M. Hoc, Green, T., Samurcay, R., and Gilmore, D., editors, Academic Press 1990, ISBN 0-12-350772-3.
3. Brooks, R., Towards a Theory of the Comprehension of Computer Programs, in International Journal of Man-Machine Studies, vol. 18, pp. 543-554.
4. Koenemann, J., Robertson, S., "Expert Problem Solving Strategies for Program Comprehension," in Proceedings of Computer Human Interaction (CHI'91), New Orleans, LA, April 1991, pp. 125-130.
5. Soloway, E. & Ehrlich, K. "Empirical studies of programming knowledge" IEEE Transactions on Software Engineering, SE-10(5), September, 1984.
6. Guindon, R., Curtis, B., Krasner, H., A Model of Cognitive Processes in Software Design: An Analysis of Breakdowns in Early Design Activities by Individuals, Microelectronics and Computer Technology Corporation (MCC) technical report STP-283-87, August 1987.
7. Letovsky, S., Cognitive Processes in Program Comprehension, in Empirical Studies of Programmers, Soloway, E., Iyengar, S. eds., pp. 58-79, 1986, Ablex publishers, Norwood, NJ.
8. von Mayrhauser, A. & Vans, A.M. "Comprehension processes during large-scale maintenance" in Proceedings of the 16th International Conference on Software Engineering. Sorrento, Italy, May 16-21, 1994. IEEE Computer Society Press. Los Alamitos, CA. 1994. pp. 39-48.
9. Lutz, E. "The Knowledge Base Maintenance Assistant" in Proceedings of the Eight Knowledge-Based Software Engineering Conference, Chicago, Illinois, September 20-23, 1993. IEEE Computer Society Press, Los Alamitos, CA., 1993. pp. 86-95.
10. Layzell, P., Freeman, M., and Benedusi, P. "Improving reverse-engineering through the use of multiple knowledge sources." Software Maintenance: Research and Practice. Vol. 7, 1995. pp. 279-299.
11. Yen, J. and Hsiao-Lei, J. "An approach to enhancing the maintainability of expert systems." in Proceedings of the Conference on Software Maintenance 1990. IEEE Computer Society Press. Los Alamitos, CA. 1990. pp. 150-160
12. Clements, Paul, "A Survey of Architecture Description Languages," to appear in Proceedings of the 8th International Workshop on Software Specification and Design, Paderborn, DE, 1996.
13. Dijkstra, E., W., "The structure of the 'T.H.E.' multiprogramming system," CACM, vol. 11, no. 5, pp. 453-457, 1968.
14. IEEE Transactions on Software Engineering, special issue on software architecture, April, 1995.
15. Hayes-Roth. "Architecture-Based Acquisition and Development of Software: Guidelines and Recommendations from the ARPA Domain-Specific Software Architecture (DSSA) program," http://www.sei.cmu.edu/arpa/dssa/DSSAexp.html, 14 January, 1994.
16. Abowd, G., Bass, L., Kazman, R., Webb, M., "SAAM: A Method for Analyzing the Properties of Software Architecture," in proceedings of the 16th International Conference on Software Engineering, Sorrento, Italy, pp. 81-90, May 1994.
17. Parnas, D., "On the design and development of program families," IEEE Transactions on Software Engineering, vol. SE-2, no. 1, pp. 1-9, 1976.
18. Moriconi, M., Qian, X., Riemenschneider, R., "Correct Architecture Refinement," IEEE Transactions on Software Engineering, vol 21, no. 4, April 1995.
19. Searls, D., Norton, L., "Logic-Based Configuration with a Semantic Network," in The Journal of Logic Programming, Vol. 8, 1990, pp. 53-73.
20. Wallnau, K., Solderitsch, J., Simos, M., "Construction of knowledge-based components and applications in Ada," in proceedings of AIDA-88, Fourth Annual Conference on Artificial Intelligence and Ada, pp. 3/1-21.
21. Brachman, R., Schmolze, J., "An overview of the KL-ONE knowledge representation system," Cognitive Science, Vol. 9, No. 2, pp. 171-216.
22. Neches, R., Fikes, R., Finin, T., Patil, R., Senator, T., Swartout, W., "Enabling Technology for Knowledge Sharing," AAAI, Fall 1991.