Functional Representation of Software Architecture Knowledge

John Hartman B. Chandrasekaran

Laboratory for Artificial Intelligence Research, Dept. of Computer and Information Science
The Ohio State University, 2015 Neil Ave., Columbus, OH 43210-1277
http://www.cis.ohio-state.edu/hypertext/LAIR/lair-page.html
hartman *or* chandra@cis.ohio-state.edu

Abstract

Software architectures specify how high-level system components interact and behave. Architecture evolution tasks require knowing an architecture's design intentions. Existing architecture description languages (ADL's) specify architectures without reference to intentions. We describe the use of Functional Representation to capture knowledge of design intentions and their implementation in an architecture. An example functional representation is given which captures knowledge of how the Two-Phase Commit protocol is implemented in a Rapide architecture. It incorporates knowledge in domains such as transaction processing, the X/Open standard, concurrent computing, and distributed computing. The explanatory value of such representations is demonstrated with question answering and a tool design which support architecture evolution. This approach will reduce the cost of designing, evolving, and implementing architectures by improving human communication, and by providing more useful tools and environments for applications such as browsing, documentation, debugging, simulation, design verification, and rationale capture.

1 Software Architecture Knowledge Representation

Architecture is the high-level design of complex software systems. It addresses how large-scale system components interact and behave, independent of implementation and non-architectural details. Architecture-based methods design and evolve systems at the architecture level, with the systems specified by *architecture description languages* (ADL's) [5, 11, 4].

Architecture evolution tasks require knowledge about design intentions which is not present in ADL architecture descriptions. Such architecture knowledge is open-ended and exceedingly complex. It includes assertions at many abstraction levels, in many descriptive frameworks, incorporating knowledge and inference from many domains, including computer science and engineering, mathematics, and the programming problem and its domain. *Intentions* are assertions about the architecture that designers (and evolvers) may reasonably have. For example, requirements are intentions about the role of the system in the problem domain. *Explanations* are assertions and inference links which connect intentions to each other and to the architecture. They are the logical structure which answers questions about how intentions are achieved.

1.1 Representation Goal

We narrow our representation objectives in several ways. First, we start with an architecture specified with an ADL. Our goal is to capture intentions that are relatively close to the architecture, and which do not involve large amounts of problem domain or other information or knowledge, e.g. computational requirements like, "Preserve system consistency regardless of transaction order," rather than problem domain requirements like, "Retrieve patient data." Second, we are only interested in capturing understanding that can be practically and usefully recorded. Rather than attempting to capture the vast fabric of potential human understanding, we want to record a small subset that is powerfully useful for communication and tools, and which can be practically acquired, represented, and used. Finally, we narrow our focus to a particular class of intentions and explanations – architecture *causal processes*. Roughly, these are intentions regarding

[0]This work was supported by ARPA, Order No. A714, monitored by USAF Materiel Command Rome Laboratories–Contract F30602-93-C-0243.

temporal sequences of events and states in the architecture. This focus requires an *executable architecture* which specifies temporal behavior. Our goal, then, is to **capture causal knowledge of architectures** to help people design, implement, modify, and otherwise evolve them.

1.2 FR-Rapide

Functional Representation (FR) is a theory and language for representing and reasoning about functionality and causal processes in devices. It has been successfully applied to a variety of tasks and devices, e.g. chemical plant failures, medical diagnosis, engineering design verification, and program debugging [2, 1, 9].

Functional Representation is organized around definitions and representations for *structure, behavior, and function.* Structure is the assumed bottom-level description of a device. Behavior is temporal, causal change in the device, particularly changes in its states ("how"). Function is an interpretation of structure and behavior as serving a role in a more abstract context ("what"). FR represents device knowledge using causal description, functional abstraction, and components. Applying FR involves: 1) *authoring* a functional representation which represents a particular human understanding of a device, and 2) *applications*, in which the functional representation is used to perform needed tasks.

We investigated FR for architecture using the *Rapide* architecture definition language [12, 13]. Rapide is used to specify an architecture's components, connections, constraints, and abstract behaviors. *FR-Rapide* uses FR to add intention knowledge to Rapide architectures. Figure 1 shows schematically how an FR connects abstract intentions, e.g. a requirement for the Two-Phase Commit Protocol, to the Rapide code.

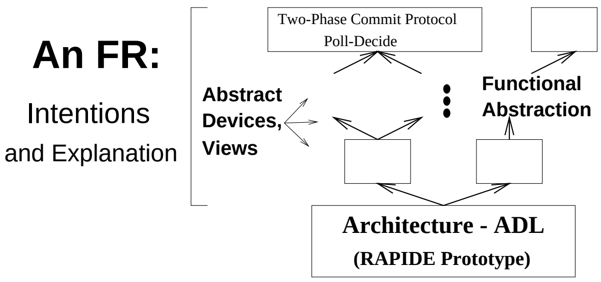

Figure 1: FR Connects Intentions, Architecture

Capturing architecture knowledge in FR has many benefits for both human and automatic processes. To perform specific specific architecture evolution tasks it is necessary to know certain intentions. For example, to modify a component, it is necessary to know the component's functionality, and how and where these intentions are realized in the architecture.

We created an FR for a Rapide distributed transaction processing architecture, X/Open. The top-level of this FR, X/Open-PD6, abstractly describes the Poll-Decide procedure in the Two-Phase Commit Protocol domain. Suppose we wish to modify the component which identifies processing errors. The FR can be used to produce necessary explanations, the following answers for example:

```
Q> What does X/Open-PD6 do?

A: Poll-Decide for Two-Phase Commit protocol...

Q> How is Poll-Decide implemented?

A: Poll-Decide-ok is followed by Commit, or
   Poll-Decide-error is followed by Rollback.
...
Q> How does B3_Pattern cause B3-some_error?

A: Semantics of Rapide behavior rule B3.
```

Such knowledge and explanations can be delivered and exploited by general and task-specific tools. For example, FR-based architecture tools can answer questions, guide browsing, generate dynamic, structured documentation, and perform inferences forward or backward in causal chains.

The remainder of this paper briefly describes writing and using functional representations of executable architectures, as demonstrated in FR-Rapide and the X/Open example. Details can be found in [7]. A related paper discusses representing representing system requirements with FR [3].

2 Rapide X/Open Example

Rapide is a language and environment for constructing prototype architectures. Architectures are executed by an interpreter which creates architectural events and their dependencies. These are recorded in partially ordered sets of events called posets.

A Rapide architecture prototype consists of interfaces of modules, connections which describe communications between the modules, and constraints which specify correctness conditions. An interface may contain an abstract behavior specifying the behavior required by every implementing module. Connections and abstract behaviors are given by *exe-*

cutable reactive rules which recognize situations in the execution and create new events in response.

2.1 Rapide X/Open Architecture

X/Open is a standard for distributed transaction processing (DTP). We investigated FR-Rapide using the Rapide X/Open reference architecture created by Kenney [10, 12]. The standard and the architecture incorporate the *Two-Phase Commit Protocol* to ensure that transactions will be atomic, i.e. they will be completed correctly, or they will be aborted and rolled back.

Imagine an X/Open architecture implemented in a bank. The ATM application program performs transactions involving checking and savings account databases, each accessed through a resource manager (RM). The transaction manager (TM) mediates between the application program and resource managers. *Poll-Decide* is the part of Two-Phase Commit by which the TM polls the RM's for their approval or disapproval of the transaction, and decides to commit or rollback the transaction. Figure 2 summarizes the Poll-Decide description in a Rapide X/Open prototype with two resource managers (NumRMs).

In the figure, XA connection rules define the XA1 set of connections between TM and RM1. The figure also shows abstract behavior rules defining the Poll-Decide behavior in the Transaction Manager interface. Interpreter execution of Poll-Decide begins when TM receives the `tm.tx.commit_call` event. This event matches Rule B1 in TM, generating events `tm.xa1.prepare_call` and `tm.xa2.prepare_call`, corresponding to TM issuing concurrent calls to the RM's. Execution continues similarly.

2.2 X/Open Architecture Intentions

Some of the intentions realized in the Rapide X/Open Poll-Decide architecture are listed in Table 1.

None of these intentions is explicitly present in the Rapide code. Representing them requires interpreting the Rapide code in domains beyond Rapide syntax and semantics. Architecture knowledge representation needs can be seen by comparing the architecture and the intentions. For example, 1) Intentions are achieved with various amounts of Rapide code, 2) Intentions involve different domains of discourse including distributed transaction processing, transaction atomicity and correctness, distributed computing, and the particular instance architecture with two RM's, 3) Different domains must be described with different languages, formalisms, models of computing etc., 4) Intentions and domains may be seen as special-purpose views of the architecture, and 5) Intentions can often be best understood in terms of other abstract domains, not the base Rapide architecture.

1. Perform transactions
2. Ensure atomicity
3. Implement Two-Phase Commit
4. Implement Polling
5. Poll RM's concurrently
6. Define distributed TM and RM components
7. Define connections between components
8. Pass specific messages, e.g. TM requests approval or disapproval from RM1
9. Get all polling results before deciding

Table 1: X/Open Poll-Decide Intentions

In summary, Rapide specifies an architecture at a level that is appropriate for definition and execution. Tasks involving Rapide architectures require knowledge that is not present in the Rapide code. The intentions present in the X/Open architecture reveal many complex issues for architecture knowledge representation.

3 Functional Representation of the X/Open Architecture

The FR author uses the FR language to represent understanding of the Rapide architecture prototype for a purpose, e.g. general documentation or supporting a task like protocol evolution. This involves the *authoring activities* in Table 2.

We constructed a functional representation for X/Open Poll-Decide (PD), using these activities, as described in [7]. For example, intentions such as those in Table 1 were chosen to emphasize understanding the algorithmic implementation of the Poll-Decide procedure. Other views and FR's would be useful for other purposes. At the bottom level, X/Open events and Rapide semantics suggested language, formalism, and vocabulary. These were blended into terminology from successively more abstract domains, terminating with the top-level vocabulary used to discuss Two-Phase Commit in the DTP domain.

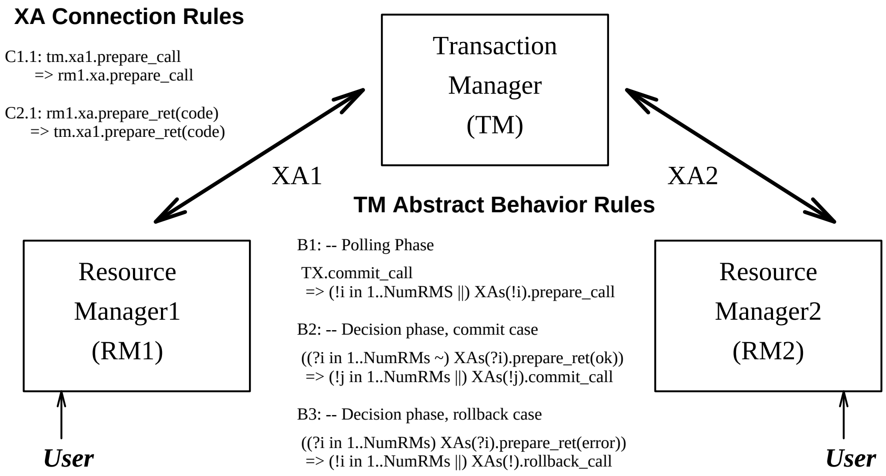

Figure 2: X/Open Poll-Decide

1. Acquiring needed understanding of the architecture,
2. Choosing top-level and intermediate intentions and views to represent,
3. Choosing state languages, formalisms and predicate vocabulary,
4. Writing functional specifications for states at various levels,
5. Forming causal sequences among states at each level
6. Creating a functional abstraction hierarchy,
7. Modularizing the representation with abstract components,
8. Validating the representation.

Table 2: FR Authoring

3.1 Functional Abstraction Hierarchy

The Poll-Decide functional representation explains how Two-Phase Commit Poll-Decide is implemented in the Rapide X/Open architecture. Figure 3 gives an overview of the FR.

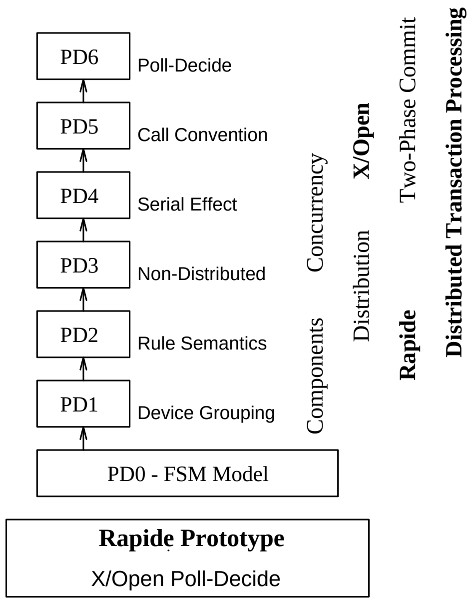

Figure 3: Poll-Decide FR Hierarchy

The FR can be visualized as seven layers on top of the Rapide prototype architecture. The bottom level, PD0, is a state machine causal process description capturing all possible architecture behavior. Successively higher levels are successively simpler state machine CPD's. Each is formed by a particular kind of functional abstraction, shown to the right of each box. The top level, PD6, is a specification of the Poll-Decide part of the Two-Phase Commit protocol. The right side of the figure roughly shows some of the domains reflected in the CPDs' abstractions and terminology. Material from lower domains is gradually supplanted with material from more abstract domains.

3.2 Bottom Level – PD0

The bottom level of the every functional representation gives the device's primitive structure and/or behavior. Figure 4 presents PD0, representing the Rapide X/Open architecture's Poll-Decide behavior. PD0 is a distributed, concurrent finite state machine recorded with FR causal process description syntax.

The states of PD0 are the complete, mutually exclusive states that can occur during prototype execution, subject to concurrency. Each state is a set of architecture events that have occurred. For example, the state `[tm.xa1.prepare_ret(ok) tm.xa2.prepare_ret(error)]` means that those two events have occurred. Other events which have occurred may also be in the state, but are not shown in the figure. CPD transitions have annotations giving their implementation or justification. In PD0 all of the transitions are due to Rapide connection or abstract behavior specification rules, or to concurrency in the architecture.

PD0 is behaviorally equivalent to the architecture for all possible executions of a single transaction, in the sense that its states and transitions completely capture the possible states (sets of events) and transitions in the architecture. PD0 has many conceptual and representational advantages over the original Rapide code.

3.3 Intermediate Levels

Here the intermediate levels of the FR are briefly described to give a sense of their abstractions and FR features. Detailed descriptions are in [7].

PD1 – Abstract Components PD0 is divided into components corresponding to the architecture components – TM, RM1, and RM2. This produces disjoint regions in TM for the polling and decision behaviors. To distinguish these computations, PD1 is created with separate abstract components for the polling and decision phases of TM. The result is four connected causal process descriptions, TM:Query, RM1:Respond, RM2:Respond, and TM:Choose.

PD2 – Rule Logical States In Rapide a rule precondition can be satisfied by various sets of events. There is no event for acceptance by the interpreter. In PD2 we introduce abstract states corresponding to acceptance in rules B2 and B3. This represents knowledge of Rapide rule semantics and the usefulness of such states. The result is a new CPD with added intermediate states, e.g. `B2-all_ok` corresponding to the acceptance condition of rule B2.

PD3 – Distribution Removal Rapide produces events for connection traversal. For algorithmic understanding, we wish to abstract away these artifacts of architecture component distribution. PD3 eliminates states and transitions due to connections. This represents knowledge of distribution in the architecture. The result is a simpler CPD for an equivalent non-distributed device.

PD4 – Concurrency Removal Similarly, the Poll-Decide algorithm is obscured by the mechanisms for concurrent polling in the architecture. PD4 abstracts away concurrent polling. This represents knowledge of concurrency in architecture. The result is a simpler CPD for an equivalent serial device.

PD5 – X/Open Standard Call Removal The architecture was based on the X/Open standard, and has rules for C calling conventions used in the standard. To focus on the protocol implementation independent of the calling mechanism, we created a CPD without the states caused by calling and returning. This represents knowledge of the roles of calls in the standard and the architecture. The resulting simpler CPD, PD5, gives a view of the algorithm as if it were contained in a single program unit.

3.4 Top Level – PD6 Poll-Decide

As a result of these abstractions, PD5 displays conditions leading to alternative logical states and actions. With an understanding of Two-Phase Commit and its realization in the architecture, this can be interpreted and abstracted to give a CPD, PD6 in Figure 5, which concisely represents the essence of Two-Phase Commit Poll-Decide.

The states and transitions in PD6 are specifications of Two-Phase Commit polling leading to approval and commitment, or to disapproval and rollback. They use vocabulary with specific technical

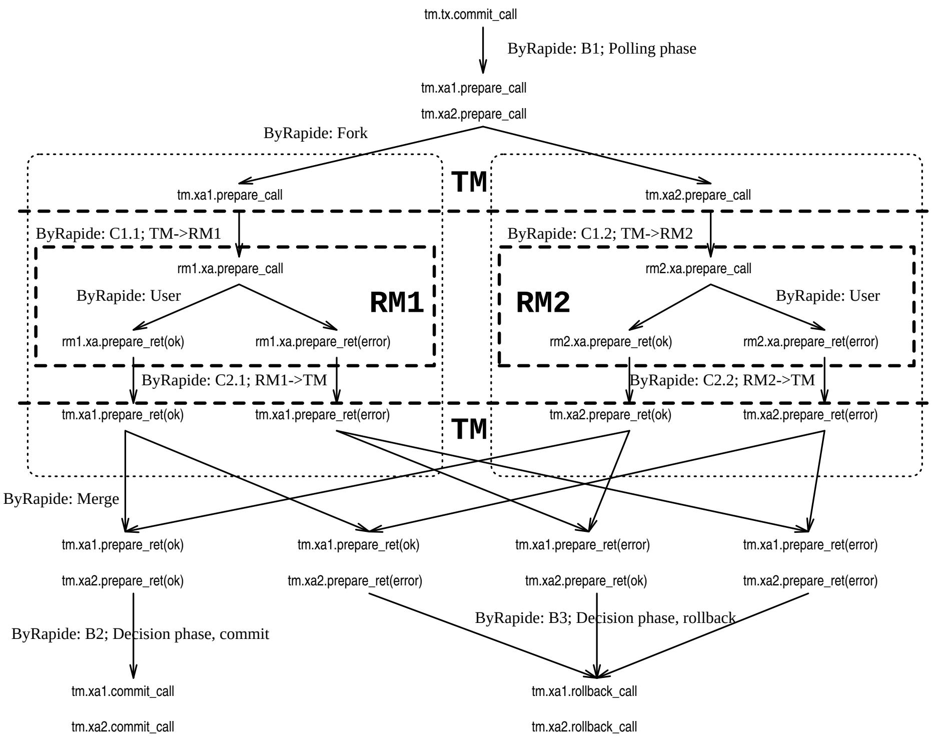

Figure 4: PD0 – Poll-Decide Finite State Machine

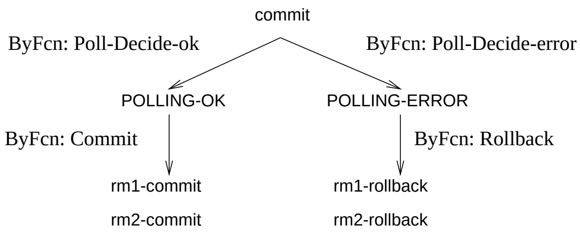

Figure 5: PD6 – Poll-Decide Top-Level

meaning in the Two-Phase Commit DTP domain. For example, the abstract state POLLING-OK is defined to mean that all RM's approved finalizing the transaction. The whole CPD describes the Poll-Decide procedure of Two-Phase Commit, which ensures transaction atomicity.

Furthermore, PD6 points to rich knowledge of how the architecture implements Poll-Decide, including the reasoning supporting the top-level specification.[1] The connection to lower levels is given by the transition annotations in PD6. More detail can be obtained by referring to the functions which cause the state transitions. For example, the transition from the architecture's abstract `commit` state to the `POLLING-OK` state is caused by the Poll-Decide-ok function, as shown by the ByFcn annotation. The definition of Poll-Decide-ok then shows how approving the transaction is justified and realized by more concrete views and Rapide architectural implementation.

Tracing all the functions downwards through the hierarchy give a complete explanation, presenting all of our understanding of the architecture. This explanation contains captured views, abstractions, and intentions, and suggests simple inferences such as:

1. Concurrent polling computations merge.

2. The `prepare_call` states produce a Two-Phase Commit "polling" computation.

3. Polling's sole effect is a decision branch in the Two-Phase Commit protocol.

[1] Note that some of the necessary functionality may not be implemented in the architecture. In this case the FR captures designer intentions which must be realized in the final system.

4 Explanations From Architecture Knowledge

Authoring an FR for an architecture creates and systematizes understanding for documentation and communication. Furthermore:

- The FR provides explicit answers or explanations for certain questions.
- These explanations are necessary or useful for architecture evolution tasks.

4.1 Questions and Answers

The smallest explanations are the primitive relationships of FR. These relationships can be seen as answering important simple questions. For example, in Figure 6 certain questions can be asked involving the indicated state transition. e.g. “How does the transition occur?” The FR represents the functionality

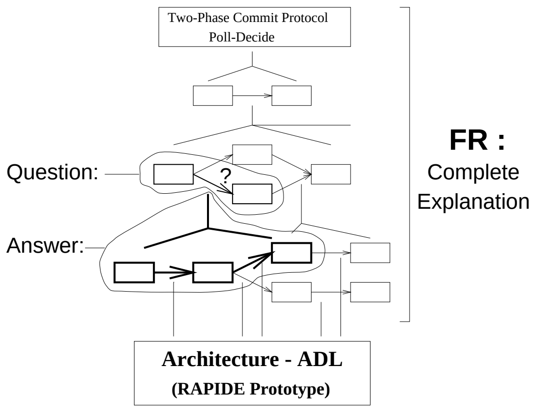

Figure 6: A Primitive Explanation From an FR

of the state change and its elaboration by a lower level. Therefore the question is answered by presenting the functionality and its transition annotation. The function and justification are “How” and, for example, “How Implemented” relationships between the questioned transition and its realization.

In general, *questions* are partly specified relationships. *Answers* are bindings to the missing elements provided by the FR. It would be possible to construct a general facility to answer meaningful questions, e.g. like a relational database query language. A stronger tool can can be based on the relatively small set of relationships and questions that are especially important for architecture evolution tasks, such as those shown in Table 3. Complex answers and explanations can be produced by combining such primitive relationships and question types.

1. Where is state1-2?
2. When does state1-3 occur?
3. What does Fcn1 do?
4. How is state2-2 achieved (starting with state2-1)?
5. How is Fcn1 implemented?
6. Why is state-1 used?
7. Why is Rapide rule B1 used?
8. Where is poset event explained?

Table 3: FR-Rapide Questions and Relationships

4.2 Explanation Tool Design

Question answering models explanation, and is attractive for practical tools, e.g. because it is general, easy to implement, and self documenting. Some of the advantages for simple novice use become disadvantages for more complex explanations and expert use, e.g. verbosity, narrow bandwidth and single, textual modality.

Graphical hypertext navigation offers complementary qualities. We can move freely over the FR, expanding or contracting context as needed. Multiple relationships and complex CPD’s are visualized from diagrams. Lewis Johnson’s I-Doc is an example of answering questions and delivering explanations of programs by hypertext browsing, using the World-wide Web [8].

We propose an explanation tool that combines features of question answering, graphical hypertext navigation, and I-Doc. The user sees the FR primarily as a hierarchy of CPD’s, with controllable detail. Clicking on a function gives its justification. Where it’s meaningful, question answering is provided in parallel with point and click navigation. For example, a menu of relevant questions is displayed beside the graphical browser. When the cursor is on graphical elements, e.g. a function name, questions light up giving the meaning of possible mouse clicks, e.g. What and How questions based on the function name. Similarly, when the answer to a question is displayed graphically, the natural language answer can often be given in an answer box under the question.

5 Discussion

Software architecture is a new field which has focused on ADL's and associated ADL level tools. While the need for deeper, knowledge-based architecture tools and representations is clear, we know little directly relevant architecture knowledge representation work. For example, research on architecture extraction from existing source code focuses on reverse engineering rather than knowledge representation.

Less directly, deep and/or wide-spectrum software engineering knowledge representations are relevant to the extent to which they can adapt to architecture difficulties like distribution, concurrency, and weak behavioral specification, or to the extent that the architecture representation eliminates these difficulties. For example, requirements, domain analysis, hierarchical program understanding, and plan representations each focus on parts of what must be represented for architecture. FR provides a comprehensive, principled framework unifying such types of knowledge and representations for architecture – and for software engineering as well.[2]

FR benefits for architecture are discussed above. Writing an FR creates and systematizes understanding. The extant FR is a notation which records and recalls this understanding for the author. Similarly, the FR is a notation for communication between the author and others. It serves the role of documentation, but with added formalized structure and conventions. Finally, the FR allows captured understanding to be manipulated, delivered, and exploited by automatic tools and environments. In all these roles, the captured understanding is beneficial if it makes it easier to perform architecture evolution tasks. This is described more precisely and operationally by the question answering paradigm.

Practical benefit depends on empirical issues such as scalability, language, validity, and cost. Some beneficial FR's can obviously be created, e.g. FR's for existing documentation which add add formalized structure. Practical benefits have been shown for FR's in non-software domains. The FR for the X/Open Rapide architecture described here has scale, language, and validity that is practically useful for tasks involving X/Open Poll-Decide. However, the benefits must be worth the cost of creating and using the FR. In many cases, the cost of creating needed understanding can be fully or partially discounted, e.g. if the understanding already exists, or if it is useful independent of FR. Similarly, FR's can be created using generic, reusable components. FR cost-benefits can be compared with alternative existing or proposed approaches.

[2] We are investigating FR for software as well as for architecture, e.g. by creating an FR for the plans and other knowledge in an automatic reengineering example, as described in [6] and subsequent work. Insights from architecture, program, and device understanding have helped each other. For example, we see ontological equivalence of architectures, plans, and abstract devices.

Specific applications could be need driven. We suggest asking:

> *"What explanation does your application need?"*

If the needed explanation involves causal reasoning, FR can, in principle, support it. Applications include processes and tasks like design, maintenance, evolution, testing, analysis, verification, validation, and system implementation. For each of these, FR can benefit many specific tools and uses such as debugging, documentation, interactive explanation, simulation, rationale capture, trace interpretation and explanation, design verification, various kinds of causal inference, and task-specific tools, e.g. an intention-based configurer for a domain-specific family of architectures and system products.

Acknowledgments

The Stanford Rapide project, including David Luckham, Larry Augustin and John Kenney, provided helpful insights and information.

References

[1] Dean Allemang and B. Chandrasekaran. Functional representation and program debugging. In *PROCEEDINGS of the 6TH ANNUAL KNOWLEDGE-BASED SOFTWARE ENGINEERING CONFERENCE*, pages 136–152. IEEE Computer Society Press, 1991.

[2] B. Chandrasekaran. Functional representation and causal processes. In Marshall C. Yovits, editor, *ADVANCES in COMPUTERS*, volume 38, pages 73–143. Academic Press, 1994.

[3] B. Chandrasekaran and Hermann Kaindl. Representing functional requirements and user-system interactions. Conference Submission, 1996.

[4] David Garlan and Dewayne E. Perry. Introduction to the special issue on software architecture. *IEEE Transactions on Software Engineering*, 21(4), April 1995.

[5] David Garlan and Mary Shaw. *PERSPECTIVES ON SOFTWARE ARCHITECTURE*. Prentice-Hall, 1996.

[6] John Hartman and B. Chandrasekaran. Functional representation and understanding of software: Technology and application. In *5TH ANNUAL DUAL-USE TECHNOLOGIES and APPLICATIONS CONFERENCE*. Mohawk IEEE and Rome Lab, May 1995. http://www.cis.ohio-state.edu/ hartman/publications.html.

[7] John Hartman and B. Chandrasekaran. Functional representation of executable software architectures. Technical report, Laboratory For Artificial Intelligence Research, The Ohio State University, Dept. of Computer and Information Science, December 1, 1995. http://www.cis.ohio-state.edu/ hartman/publications.html.

[8] W. Lewis Johnson and Ali Erdem. Interactive explanation of software systems. In *Proceedings KBSE'95 – The Tenth Knowledge-Based Software Engineering Conference*. IEEE Computer Society Press, November 12-15, 1995. Boston, Mass.

[9] John R. Josephson. Technical note on formalizing functional representation. In J. Hodges, editor, *AAAI-94 WORKSHOP on REPRESENTING and REASONING ABOUT DEVICE FUNCTION*. AAAI, 1994. Seattle, Washington.

[10] John J. Kenney. *EXECUTABLE FORMAL MODELS of DISTRIBUTED TRANSACTION SYSTEMS BASED ON EVENT PROCESSING*. PhD thesis, Stanford University, June 1995.

[11] Paul Kogut and Paul Clements. The software architecture renaissance. *CrossTalk*, 7(11):20–23, November 1994.

[12] David C. Luckham, John J. Kenney, Larry M. Augustin, James Vera, Doug Bryan, and Walter Mann. Specification and analysis of system architecture using Rapide. *IEEE Transactions on Software Engineering*, 21(4):336–355, April 1995.

[13] David C. Luckham and James Vera. An event-based architecture definition language. *IEEE Transactions on Software Engineering*, 21(9):717–734, September 1995.