Requirements Evolution and Reuse Using the Systems Engineering Process Activities (SEPA)

K. S. Barber, T. Graser,
P. Grisham, S. Jernigan
The Laboratory for Intelligent Processes and Systems
Electrical and Computer Engineering
The University of Texas at Austin
Austin, TX 78712
http://www.lips.utexas.edu
barber@mail.utexas.edu
phone: (512) 471-6152
fax: (512) 471-3652

Abstract

As more organizations attempt to reuse previous development efforts and incorporate legacy systems, typical software development activities have transitioned from unique ground-up coding efforts to the integration of new code, legacy code, and COTS implementations. This transition has brought on a whole new set of development issues, including resolving mismatches between integrated components and tracing legacy and COTS components to requirements. This paper presents the Systems Engineering Process Activities (SEPA) methodology, developed to address these and other issues in current software development practices. SEPA aids the reuse and integration process by focusing on requirements integration and evolution, while maintaining traceability to requirements gathered from domain experts and end users. The SEPA methodology supports the development process in three main areas: *(i)* requirements analysis prior to design; *(ii)* separation of domain-based and application-based (i.e. implementation-specific) requirements; and *(iii)* requirements analysis for component-based development. The paper also presents a spectrum of research tools that are currently being developed to address these main features of the SEPA methodology followed by an example illustrating the application of SEPA in the incident response domain to facilitate requirements management and foster requirements reuse.

Technical Report
TR99-UT-LIPS-SEPA-05

August 15, 1999

Requirements Evolution and Reuse Using the Systems Engineering Process Activities (SEPA)

K. Suzanne Barber, Thomas J. Graser,

Paul S. Grisham, Stephen R. Jernigan

The Laboratory for Intelligent Processes and Systems
The University of Texas at Austin
Austin, TX 78712
voice: 512-471-6152 fax: 512-471-3316
barber@mail.utexas.edu

Abstract

As more organizations attempt to reuse previous development efforts and incorporate legacy systems, typical software development activities have transitioned from unique ground-up coding efforts to the integration of new code, legacy code, and COTS implementations. This transition has brought on a whole new set of development issues, including resolving mismatches between integrated components and tracing legacy and COTS components to requirements. This paper presents the Systems Engineering Process Activities (SEPA) methodology, developed to address these and other issues in current software development practices. SEPA aids the reuse and integration process by focusing on requirements integration and evolution, while maintaining traceability to requirements gathered from domain experts and end users. The SEPA methodology supports the development process in three main areas: *(i)* requirements analysis prior to design; *(ii)* separation of domain-based and application-based (i.e. implementation-specific) requirements; and *(iii)* requirements analysis for component-based development. The paper also presents a spectrum of research tools that are currently being developed to address these main features of the SEPA methodology followed by an example illustrating the application of SEPA in the incident response domain to facilitate requirements management and foster requirements reuse.[12]

---

[1] This research was supported in part by the Texas Higher Education Coordinating Board Advanced Technology Program (ATP #003658452) and the Defense Advanced Research Projects Agency (DARPA).

[2] The authors would like to acknowledge that contributions to the research presented in this paper were made by former lab members Brian McGiverin and Srini Ramaswamy.

1. Introduction

Despite the introduction of software development methodologies and CASE tools to help alleviate the "Software Crisis," the development and maintenance of software is still too expensive. In large part, the software crisis can be attributed to poorly defined and managed requirements and insufficient reuse opportunities. When Alford reviewed several Air Force projects, he found that "in nearly every software project which fails to meet performance and cost goals, requirements inadequacies play a major and expensive role in project failure" (Alford & Lawson, 1979). In addition, requirements inadequacies reduce opportunities for software reuse, effectively increasing system development costs. Even if requirements for a system have been adequately defined, reuse of the system components is threatened when 1) the components are not associated with specific requirements and 2) the requirements for a component are not evolved along side it during the maintenance phase.

Maintenance alone accounts for nearly 70% of software development cost (Schach, 1993). The process of gathering, managing, and analyzing requirements has a direct affect on the cost of maintenance. The need for maintenance can be chiefly attributed to (i) the satisfaction of evolving user requirements and (ii) changes needed in a software implementation to address requirements not addressed in the original delivery. The root of the latter lies in poor requirements gathering and analysis and an poor translations from requirements to system designs. Furthermore, inadequate verification and validation often fail to detect delivered systems that fall short of their intended purpose. In most domains, the evolution of user requirements is inevitable. Changes in requirements can occur even before the initial release of the system. Needs continue to change after system development as the benefits of new technologies are recognized and domain processes evolve.

Since maintenance and system modification are inevitable, the software community has focused on achieving greater reuse of software through the use of software components. According to Caldieri, Gianluigi, and Basili, reuse saves time and resources and reduces the risk of building in defects. Effective reuse from previous software development efforts has the potential for increasing software productivity and quality an order of magnitude (Caldieri, Gianluigi, & Basili, 1991). This implies that software developers are becoming design and integration specialists by “building” complex software systems with the help of commercial-off-the-shelf (COTS) software products. However, the cost of installing and customizing COTS software, developing supplemental in-house software components, and integrating such software systems with existing systems can be enormous. A new thrust, termed Component-Based Software Engineering (CBSE), aims to mitigate this cost and addresses the unique requirements of COTS-based development. CBSE has been largely affected by the distributed nature and prevalence of the World Wide Web, the acceptance of OO techniques, and the move toward client/server environments (Brown, 1996).

With the advent of component-based software development, the need for integration typically falls into one of the following scenarios:

1. *COTS integration from multiple developers.* The development environment has moved beyond sole source component solutions. While industry standard protocols have somewhat simplified integration, they have not completely eliminated the difficulty faced when choosing among potential technologies. The complexity of integration is aggravated when different components address fundamentally different requirements, address similar requirements differently, run on different platforms, or involve multiple protocols common in today's distributed systems.

2. *COTS integration with new customized portions.* Systems are often a hybrid of COTS portions and customized portions, where business logic is added to custom components that take advantage of basic services (e.g. word processing, data storage) provided by COTS components.

3. *Integration of legacy, COTS, and new components.* Many businesses have a large investment in existing, legacy systems that have a proven record of enabling essential business processes. It is often cost prohibitive to rebuild these systems, yet new functionality must be incorporated.

Although the use of existing, well-tested components reduces the need for reinventing the wheel, it adds to an already difficult integration effort. While systems built from scratch have the advantage of being designed for integration in the system, the disparate components integrated in today's systems are not all designed with integration in mind. Customers demand *both*, the features these components provide and the assurance they will integrate seamlessly. Furthermore, given the ever-increasing complexity and scale demanded of today's systems, the move toward plug-compatible standards has little chance of keeping up with the difficulties in integration resulting from new technologies and complex requirements. The difficulty of integrating disparate components has been a significant barrier to realizing code reuse on a large scale (Garlan, Allen, & Ockerbloom, 1995).

Considering the complexity of COTS integrations, each new system can often become as much a new invention as if it had been created from scratch. Programmers often assume that writing their own code is easier than integrating someone else' s code. Typical issues aggravating the integration effort include the following:

- Current methodologies *lack the ability to reuse requirements* in determining if intended end-user needs for a new system are similar to those of a previously developed system or portions of new requirements are related to portions/components of previously developed systems.

- Current methodologies *do not emphasize the reuse of artifacts other than code (analysis or design)* from previous implementations.

- It is a *rare opportunity when new systems can be developed entirely from COTS software components* designed for seamless integration. The addition of "glue code" is typically necessary to aid integration. While the use of COTS components may reduce programming effort, this additional level of effort represents value added which must be factored into the total cost in development.

Integration carries with it new challenges and cost considerations, including evaluating the degree to which selected components will satisfy requirements and determining the level of effort required to modify selected components for successful integration. By focusing on code and software component reuse alone, large scale reuse may be unachievable. To increase the opportunities for reuse, developers should consider reuse of artifacts throughout the analysis, design, and implementation phases -- the opportunities for reuse are vast and should not be limited to code alone. The U.S. Department of Defense suggests that coding should only consume about 10% to 15% of total software development time, so an emphasis on analysis and design reuse is more profitable. While code reuse typically occurs only at lower-level system design artifacts, analysis and design reuse often results in whole collections of related artifacts being reused (Department of Defense, 1996). Methodologies can further increase odds for reuse by identifying potential reuse at all system levels (e.g. entire system, subsystem, code function) and among all participating elements (e.g. new development, legacy systems, and COTS components).

The selection of any existing component for reuse or the development of a system component should have an appropriate justification based on requirements, cost, and schedule. The cost and complexity associated with system development and integration reemphasize the need for gathering and analyzing client requirements. Although most current methodologies recognize the need for this activity, requirement analysis is still treated as a *separate process that somehow occurs* before “actual” software development. Current object-oriented (OO) approaches provide little guidance for requirements elicitation and refinement prior to developing an object model and thus poorly satisfy the needs of component-based development.

Reuse on the requirements level provides a number of benefits, including the following:

1. *motivation for selection of components:* Requirements gathered and analyzed guide the selection of optimal components for reuse. When requirements are transferred between development efforts, the justification for selecting components in a new development effort based on reused requirements brings with it the prior justification associated with the originally selected components.

2. *context for reuse decisions:* Requirements tie directly back to information gathered from domain experts and system users. Requirements are thus set in the context of domain processes or specific implementation needs.

3. *parametric constraints:* Requirements come in many forms, including specific parametric constraints (i.e. the system delivered must run at speed x) as well as

general "wish list" statements (e.g. the system's interface should be user friendly) and domain tasks and processes. Parametric constraints allow a first cut, static evaluation to narrow the field of available components.

Although the leveraging of legacy systems comprises a large part of the demand for reuse, the reusability of old systems is often difficult to evaluate by direct inspection. Perhaps the rework effort can be determined through code evaluation and system documentation. However, it can be difficult to determine the relative adherence of the old implementation to newly gathered requirements. When originally under development, requirements were likely gathered and analyzed for satisfaction by the old implementation. Given that a requirements effort has already occurred, an alternative is to determine which of the old requirements can transfer to satisfy new requirements rather than take on a fresh evaluation of the old system.

When requirements are established, they reflect the needs of the domain, organization, and specific implementation. However, these needs continue to evolve. Therefore, a methodology and its associated tools should support a "requirements evolution" process. Given that requirements guide the implementation process, it should be possible to trace an implementation back to requirements. As requirements evolve, tool support can allow the corresponding traces to implementations to be carried along, providing guidance as to how the old implementation may satisfy new requirements. Further, the set of new requirements which are not addressed by old requirements provide an indication as to the degree of rework required (expressed in requirements) and guide additional development.

Three general approaches to requirement's representation are applied today. The first approach uses a long list of natural language "shall," "should," and "will" statements. Tool support may include the ability to hyperlink particular requirements to later project artifacts or the ability to gather requirements into a checklist. IEEE recommended practice for Software Requirements Specifications (Std 830-1993) can be met by this representation but the required maintenance can be significant for larger projects. Many popular commercial case tools use this representation (e.g., RTM (Chipware, 1999), SLATE (Technologies, 1999), and DOORS (QSS, 1998)). Palmer notes the difficulties associated with maintaining traceability when apply these tools (Palmer, 1997). Specifically, he notes that the elements to be traced must be manually identified and the links between traceable elements must be manually established. Furthermore, requirements in this representation can be hard to validate for completeness and consistency. A second approach is to formal specify the requirements in hopes that a theorem proving system can *prove* the correctness of the specification (Heitmeyer, Kirby, & Labaw, 1997). However, the difficulties associated with creating correct formal statements that cover all of the requirements typically limit these approaches to research organizations. A final approach is to use semi-formal graphical models (e.g., E-R diagrams, use-case diagrams, process traces) for representing requirements. The semi-formal models can be readily created by knowledge engineers and verified by domain experts. However, since any one graphical notation can't show all aspects of a system, multiple notations (models) are required and practitioners must be concerned with

overlap, inconsistencies, and omissions among the models. CASE tools, such as the ones presented in this paper, can provide some level of validation and assist in the maintenance of traceability information.

Figure 1 depicts the requirements evolution scenario and the traceability to implementations under a comprehensive development methodology. *Old Requirements* captured, analyzed, and verified are linked to an existing implementation, *Old Implementation*. As *New Requirements* are captured, analyzed, and verified, they are mapped to *Old Requirements*, thereby indirectly referencing components from the *Old Implementation*. The mapping from *New Requirements* to *Old Requirements* may not cover the complete set of *New Requirements*. In fact, *Old Requirements* may not even be relevant in the *New Requirements*. Moreover, the *Old Implementation* may only partially address the *New Requirements*, resulting in the need for evolving (modifying) the *Old Implementation*. The remaining *New Requirements* spawn development of new components (*New Implementation*), which when integrated with components from the *Old Implementation*, result in a *Delivered Product.* While this approach appears intuitive, the difficulty lies in matching requirements to implementations. To effectively render this mapping requires implementations be expressed in a "meta-specificiation" composed of the same language as the requirements themselves. Furthermore, this specification should operate at an abstraction level that allows it to be independent of implementation specifics (e.g. Component A requires an integer value while Component B provides a float value), yet be expressive enough to capture a broad range of requirements (e.g. component must perform domain Task X on Operating System Y with Throughput Z). Defining this meta-specification at the appropriate level of abstract along with careful management of *Traceability Links* from requirements to implementations will encourage the reuse of whole collections of related artifacts from analysis and design.

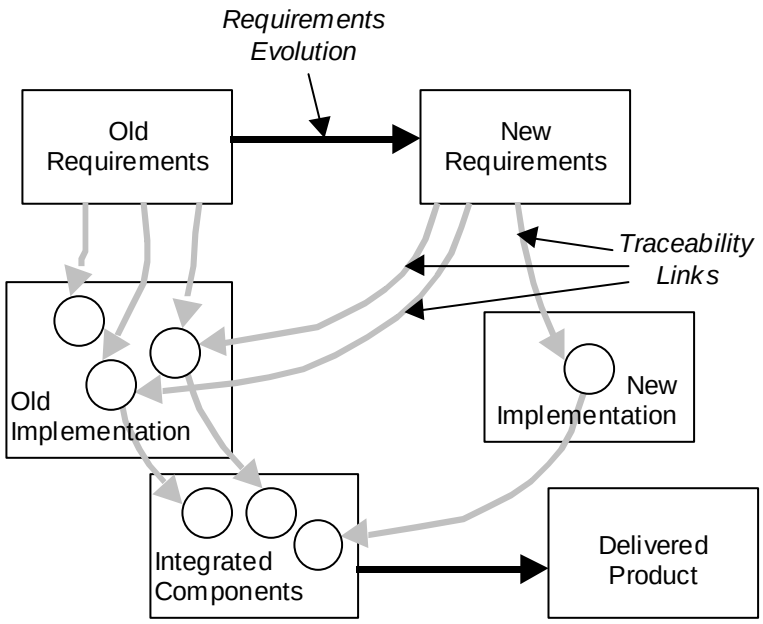


Figure 1: Requirements Evolution and Traceability

Given the requirements evolution process outlined above, the development cycle becomes an issue of integration on two levels:

1. mapping, evolving, and integrating requirements previously gathered, analyzed, and verified into new requirements and

2. integrating technologies inferred by the traceability links from integrated requirements.

Figure 2 illustrates this two-level integration problem. Requirements are derived from an array of sources from a previous development effort, available media (e.g., system documentation), multiple domain experts, and multiple system users. Requirements elicited from multiple sources often overlap and possibly contradict. Yet understanding requirements involves incorporating multiple viewpoints, involving the capture, analysis, and resolution of many ideas, perspectives, and relationships at varying levels of detail (Kotonya & Sommerville, 1997). Once integrated, requirements guide the selection of appropriate COTS components, legacy systems, and newly developed implementations. Multiple system configurations may satisfy the same set of requirements, requiring further tradeoff studies for evaluation. Resulting selections are then integrated into a cohesive solution.

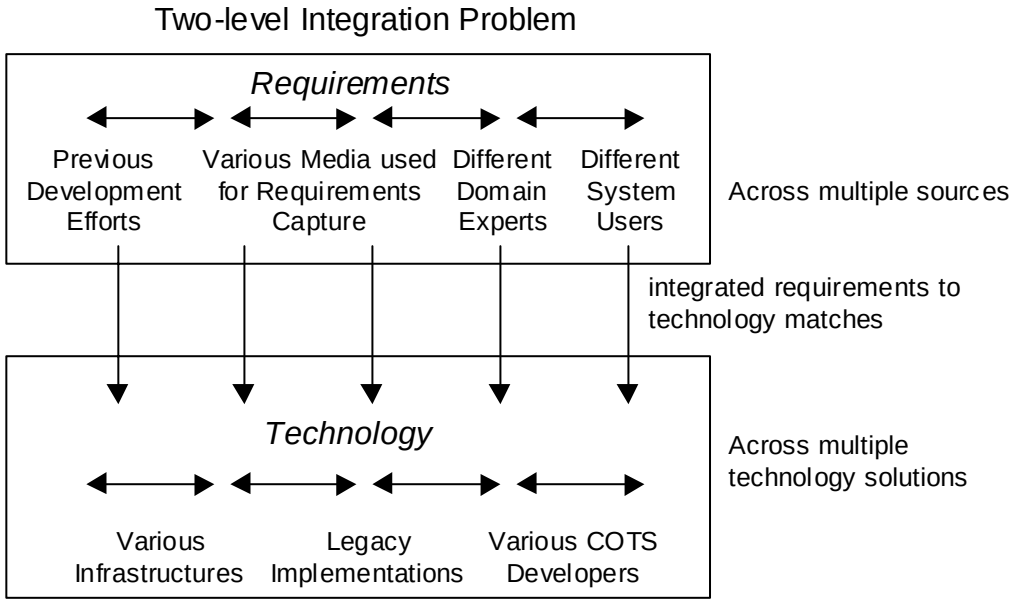


Figure 2: Two-level Integration Problem

Along with identifying the need to develop, maintain, and integrate component-based and legacy systems, researchers have also recognized the contribution of domain analysis (Gomma, 1995; Perito-Diaz, 1990) and knowledge engineering (Alonso, 1996) in complex software systems development. The Domain-Specific Software Architectures (DSSA) engineering process was introduced in 1991 to promote a clear distinction between domain and implementation requirements[3] and provide a procedure to guide developers from knowledge acquisition to design (Tracz, 1991; Tracz, 1993; Tracz, 1995). A case study using a domain specific approach to the reuse of requirements is presented in (Lam, 1997). DSSA-ADAGE is a joint industry/university research effort which was initiated to demonstrate the benefits of large-scale component-based software reuse within the avionics domain by creating a process-oriented, software composition environment that uses constraint-based reasoning to assist the user in application generation and verification[4] (Tracz, 1996). The DSSA process consists of domain analysis, domain modeling, architecture specification, component documentation, architecture specialization and component generation stages. In (Alonso, 1996), the authors clearly identify and establish the need for modern-day methodologies to integrate knowledge engineering into the software engineering lifecycle.

---

[3] Implementation requirements are specific to a particular system instantiation, while domain requirements are valid for all present and future instantitations. This distinction will be discussed further in Section 3.2.5.

[4] ADAGE stands for Avionics Domain Application Generation Environment.

The Systems Engineering Process Activities (SEPA), as described in this paper, builds upon both the DSSA methodology and popular object-oriented (OO) approaches by focusing on knowledge engineering activities critical for effective domain analysis. Similar to DSSA, SEPA highlights the need to distinguish between domain requirements (e.g. the primary task of a Payroll Dept. is to issue payroll warrants) and application requirements (e.g. all payroll warrants must be processed within 8 hours), while also promoting the identification and specification of appropriate "domain components" and corresponding "technology solutions." Domain components represent an object-oriented partitioning of domain tasks (e.g. issue payroll warrants) across responsible parties (e.g. "Payroll Issuer") without designating a particular implementation. To realize an actual system implementation, the system designer must select appropriate technology solutions based on application requirements to satisfy chosen domain components based on domain requirements. A technology solution need not be a hardware or software implementation; the most appropriate solution may be an individual playing the role of a domain component (e.g. Fred Smith is best selection as a "Payroll Issuer" because he is able to process all payroll in less than 8 hours).

Object-oriented development provides widely acknowledged benefits such as the reuse of existing code, extensibility, and simplified maintenance. Object-oriented analysis (OOA) focuses on “what” must be done while object-oriented design (OOD) focuses on “how” it is done. The importance of this distinction is that “what” an object does is less dynamic over time than “how” the object does it. SEPA borrows and extends this “what vs. how” phenomenon to defining domain components and identifying applicable technology solutions. Domain requirements are quite stable and relatively resilient to technological changes, while application requirements are more dynamic and exhibit a greater degree of vulnerability to changes in technology. In addition, SEPA emphasizes the iterative refinement process typically espoused by different lifecycle approaches.

SEPA extends DSSA by formalizing the analysis and design methodology and providing tool support throughout the process. For example, the SEPA tools aid the process of gathering and representing requirements knowledge and accommodate the coexistence of contrasting (and often conflicting) requirement perspectives. These contrasting perspectives are subsequently unified and used to derive domain-based components and specific application requirements. SEPA is further discussed in Sections 3 and 4.

For completeness purposes and to help underline the arguments presented in the subsequent sections, Section 2 provides a brief overview of some features which are strongly desired in a methodology intending to support requirements evolution and reuse. With this background, Section 3 introduces the SEPA methodology and positions its distinguishing characteristics among the features outlined in Section 2. The SEPA tools are introduced in Section 4, followed by an example in Section 5 illustrating the

application of SEPA in the incident response domain.[5] Section 5 is concluded with a discussion highlighting selected SEPA contributions to requirements evolution and reuse evident in the example. Section 5.7 then concludes the paper.

2. Methodology Features Necessary to Support Requirements Evolution and Reuse

It is understood that different application domains will likely require appropriate fine-tuning of methodologies to obtain the right "fit" (Glass, 1996) and that methodology modifications are inevitable. It is also known that this is not an uncommon occurrence in real-world projects (Glass, 1996; Hardy, 1995; Oskarrson, 1996; Vlasbom, Rijsenbriji, & Glastra, 1995). However, it is essential that a methodology exists to collaborate development, insure evaluation, and provide a support mechanism (e.g., documentation, various graphical models for representation, guidance on what concepts to represent) for complex, component-based software systems development. Therefore, despite distinguishing discrete activities in the software lifecycle (such as requirements gathering, specification, design, etc.), a methodology must allow consecutive or concurrent activities to flow smoothly together. Each of the lifecycle activities does not exist in a vacuum; interdependencies do exist (e.g., dependencies caused by the deliverables of one activity used as the inputs of the following activity). A methodology needs to aid developers by providing a smooth transition as requirements are acquired, modeled, and refined. It should also assist the identification of new implementations with respect to currently available technologies and ensure implementation selections address new and existing requirements. A methodology should not force all activities of the development lifecycle to be advanced in unison. Since there is no guarantee all requirements can be gathered, analyzed, and verified in a single effort, the methodology must support the incremental incorporation of new requirements with those already represented. Furthermore, the methodology should support the *inevitable rework cycles* by upholding previous decisions where they are still applicable. Thus, while it is possible for a methodology and its supporting tools to address only part of the lifecycle, it *should* unquestionably direct the team of system engineers throughout the *entire* lifecycle process.

A report by Grady (Grady, 1992) suggests that 50%-60% of the defects in a software product are introduced during design, not implementation. While such statistical information is oft repeated, frequent failure reports of adequately staffed and well-funded software development efforts have served to emphasize the need for methodology support for formal requirements gathering and analysis.

In this section, we briefly discuss some essential features of good software engineering methodologies to adequately support requirements evolution and reuse, with an emphasis on activities during requirements gathering and analysis. Callouts depicted in Figure 3

[5] Detailed discussions of the operations of SEPA tools are beyond the scope of this paper.

indicate where methodology features address the requirements evolution, traceability, and integration issues discussed in Section 1.

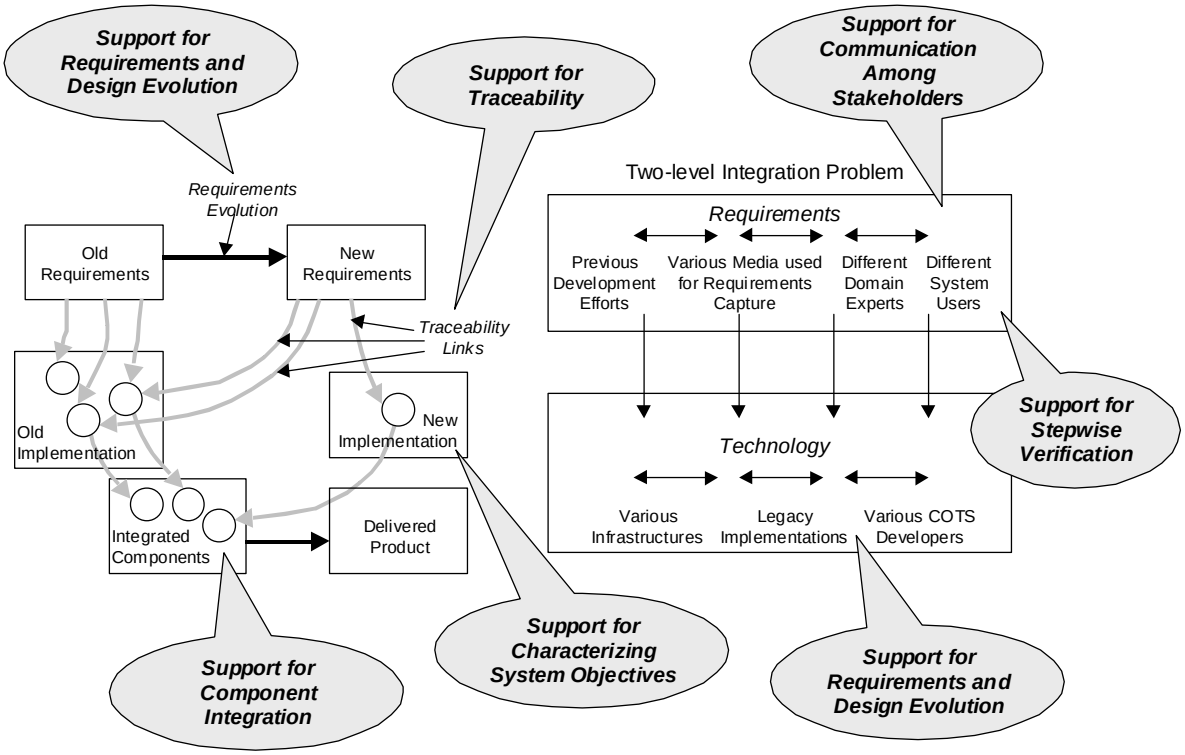


Figure 3: Methodology Features Necessary for Requirements Evolution and Reuse

2.1. Support for Requirements and Design Evolution

Cybulski suggests that requirements be viewed as an evolving negotiation among the stakeholders (Cybulski, 1995). This negotiation results in the evolution of the client's "conceptual" requirements for the project. One factor often overlooked in many methodologies is the continuous evolution of these "conceptual" requirements and effective tools to support system engineers in capturing these changes. Another factor that forces the design to evolve is the continual development of new technologies. Methodologies that anticipate and support these evolution in requirements / technology can avoid costly maintenance resulting from bringing the delivered technology product inline with the client's perceptions.

2.2. Support for Communication Among Stakeholders

The success of a development effort strongly depends on the ability of participants (i.e. end users, developers, integrators) with varying backgrounds to express their views and communicate effectively. In fact, the importance of communication between the system stakeholders cannot be understated (Cybulski, 1995). A Savant Institute study (cited in

(Christel & Kang, 1992)) found that 56% of the errors in an installed system are the result of poor communication. It is also stated that these errors were the most expensive to correct, consuming up to 82% of the total development time. Thus, a methodology must support the expression of diverse viewpoints, while providing adequate mechanisms for reasoning with and coalescing requirements from, such views.

2.3. Support for Traceability

As stated in the previous paragraphs, the reality of complex system design dictates that it is highly unlikely that all relevant information will be gathered and modeled correctly in a single iteration. Therefore, it is necessary that new information and requirements changes be identified and traced throughout the process. Methodologies that maintain links from design features back to the requirements they address, along with information about the original source (e.g., domain expert, end user) who specified those requirements, can identify requirements problems early. This will help promote conflict resolution by recording appropriate "rationale at issue" points for each conflict. Traceability links between the components in a design and the requirements each component fulfills provide a source for more complete explanations of design choices. A methodology must therefore sufficiently support this progressive refinement and change of requirements, while preserving the traceability of these requirements from and to design artifacts.

2.4. Support for Component Integration

A component-based implementation is an integration of components designed to operate as a cohesive whole. This implies that any integration issues encountered during implementation are resolved. Resolution typically takes place during implementation between implemented components rather than between the requirements and specifications that describe components. An ideal methodology should support the representation of these requirements *prior* to implementation to allow the designer to evaluate implementation alternatives, such as decomposing functionality or rearranging functionality among components.

More often than not, problems encountered during system integration have roots in originally stated requirements. A methodology should not only provide a mechanism for representing integration needs (dependencies between components), but also provide a mechanism for tracing them to the original requirements. For example, system integration issues are often based on data or event dependencies between components; one component may require data or events from another component. By capturing such requirements and dependencies earlier in the lifecycle, the designer may understand (and possibly correct) integration problems prior to realizing the implementation.

2.5. Support for Characterizing System Objectives

To encourage long term reuse, a methodology should provide for the creation of a system architecture comprised of implementation-independent components satisfying

domain requirements (e.g. domain data owned by a component, domain services provided by a component, integration rules between components originating from data/event dependencies). These domain-based components are highly reusable because (1) they are relatively stable over time and (2) they can be satisfied by a variety of specific implementations. A fundamental objective in building this type of system architecture is to accommodate long-term architectural characteristics prioritized by the system stakeholders, where priorities are defined in accordance with the overall goals of the development effort. These overall goals are based on concerns such as: dynamics of the domain, turnover of domain experts, availability of technology solutions, and the likelihood of changes in application requirements.

2.6. Support for Stepwise Verification

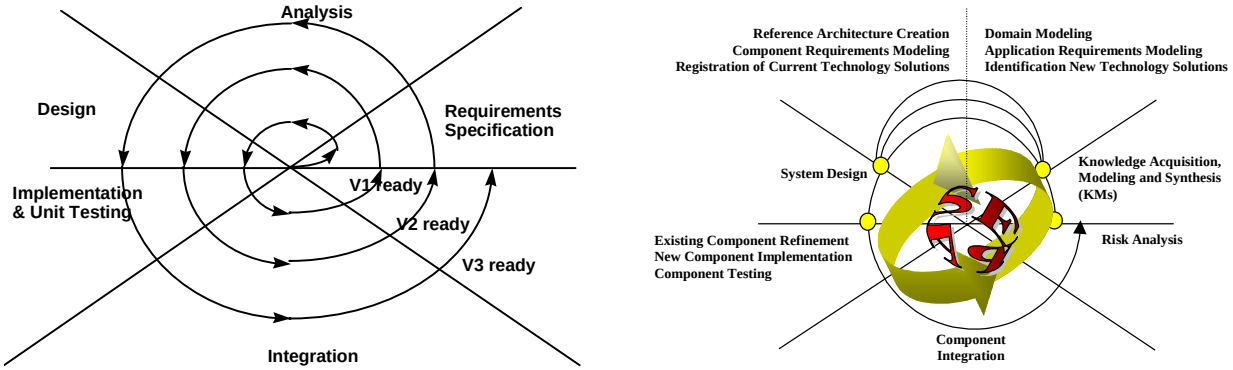
Verification considerations are most often confined to checking an implementation against the requirements specification document. However, verification should be supported throughout the software lifecycle. Formal verification, in terms of proving the correctness of the system deliverables, is neither appropriate, nor feasible during requirements gathering. Therefore verification, as it pertains to this discussion, refers to the process of feeding intermediate deliverables back to domain experts and end users (the source of the requirements the deliverable intended to address) to *verify* the correctness of those deliverables. Kramer asserts that "requirements may be misunderstood because they are so complex that the client and practitioner have difficulty focusing on one aspect at a time and perceiving interactions between requirements, or because the specified system is impossible to visualize from the resulting specification" (Kramer, Ng, Potts, & Whitehead, 1988).

In the requirements gathering and modeling activities, verification is often impeded by the introduction of multiple domain knowledge (i.e. requirements) sources as well as the size, structure, consistency and correctness of the requirements. To combat this, most methodologies rely on a number of intuitive, graphical notations. These notations allow the client to verify the requirements captured with greater ease. The application of a variety of available notations helps the designer express specific aspects (e.g. data, events, timing) of the requirements. Verification of the requirements in these notations resolves fundamental problems early in the design process. However, the use of heterogeneous, overlapping representations complicates the process of checking for consistency and completeness when different notations represent similar concepts. This inconsistency problem is exaggerated by the necessity that requirements must originate from a variety of sources including both domain documents and stakeholders[6]. Given the amount of information, it is difficult to identify the inconsistencies, much less verify that they were resolved.

[6] "Information gathered from only one group, or only one level, is likely to be biased by the level of abstraction from which those people conceptualize the problem domain, their planning horizon, level of expertise, personal preconceptions, goals, and responsibilities" (Christel & Kang, 1992).

Methodologies with appropriate tool support can aid in tracking and verifying the details (e.g. performance concerns, architectural preferences, target system environment) of a large project. This tracking and verifying should include the automatic verification of component designs and characteristics of the system architecture against the necessary properties as captured by the requirements.

3. The SEPA Methodology



a: Spiral Model **b: Modified Spiral**

Figure 4: Lifecycle Illustration

In this section, activities that comprise the SEPA methodology are discussed. SEPA is designed to leverage and extrapolate the advantages offered by several other design methodologies, while emphasizing support for the following areas: *(i)* requirements analysis prior to design; *(ii)* separation of domain and application (i.e. implementation-specific) requirements; and *(iii)* requirements analysis for component-based development. To better distinguish the SEPA methodology, these three important and distinguishing characteristics of SEPA are addressed in the context of the required methodology features presented in Section 2.

3.1. Overview of SEPA Activities

To understand the significance of SEPA activities, the SEPA structure is presented using Boehms' spiral model in Figure 4. Figure 4a illustrates Boehms' spiral lifecycle model for software development and Figure 4b shows a modified spiral model highlighting activities emphasized by SEPA. However, SEPA activities are more aptly described by the SEPA funnel structure as shown in Figure 5. It represents a spectrum of user inputs/requirements that are continuously gathered, narrowed, refined, and structured into a component-based system design specification. User inputs require refinement for a number of reasons, including the need to: *(i)* merge inputs from multiple sources, *(ii)* discard irrelevant information, and *(iii)* distinguish between inputs relating to system implementation requirements and those relating to general domain knowledge.

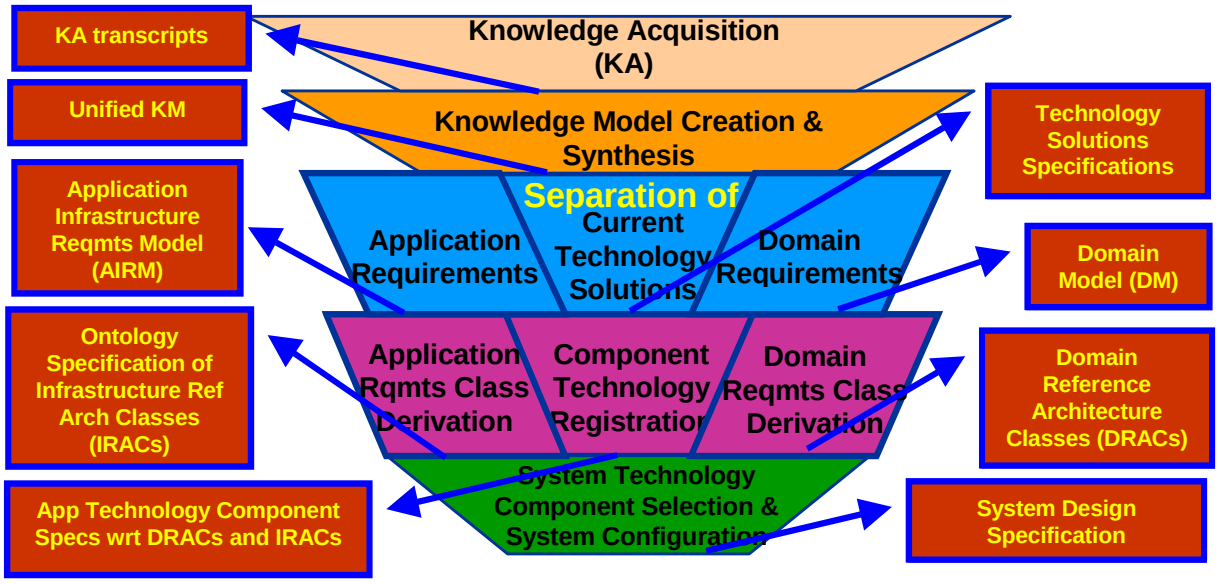


Figure 5: The SEPA Funnel

The SEPA methodology emphasizes the separation of user requirements for a particular application from the knowledge applicable to the general domain. Whether the knowledge models are elicited simultaneously and then separated, or they are elicited independently, an application cannot be created without gathering information about the domain as a whole, along with the requirements of the specific application. During knowledge modeling, Knowledge Engineers (KEs) employ knowledge models (KMs), such as message sequence charts, task descriptions, etc., to graphically depict and document knowledge acquired from domain experts and promote verification and validation feedback cycles. A single KA session may result in several new KMs. During the KA process, KEs are typically presented with two distinct types of information from the domain expert: domain-specific and application-specific information. In an ideal situation, the KE would have separate KA sessions with the domain expert for each of these types of information. However, domain experts do not typically have this abstracted view of their work and may find it difficult to provide such information. Therefore, the preferred approach is to elicit information from domain experts in the context of scenarios of operation (McGraw & Harbison, 1997), whether it relates to the entire domain, current project, or past projects. This information is captured in KMs that necessarily contain both domain-specific and application-specific information. The translation from the KMs to the Domain Model (DM) only preserves the domain-specific information. Similarly the Application Requirements Model (ARM) preserves any application specific information. The DM is a unified homogenous model. Since the representation for the DM may contain more information than can be viewed at any one time, it results in multiple views. These views are often graphical and usually show abstractions of the entities in the model rather than the complete details of the entities. During the KM and DM stages, the KE repetitively refines and structures these views.

The SEPA Reference Architecture (RA) must be completely domain-specific and highly flexible for building similar systems in the future. This flexibility is achieved by describing components in terms of "what" they do. The following criteria have been identified for classification and representation of components: *(i)* Model of assigned characteristic attributes and services. *(ii)* Abstraction of component behavior. *(iii)* Constraints between the domain components resulting from data/service dependencies and subsystem compositions. The Reference Architecture (RA) therefore gives a repository of domain component classes, which are reusable in a "family" of domain applications. One or more actual objects in an implementation may realize a single component. As a result, these component definitions can outlive technology solutions available during the analysis and design activities.

Implementation requirements are originally captured in KMs, which are then translated into the Application Requirements Model (ARM). A System Design Specification is constructed by selecting available technology solutions mapped to RA components. Knowledge about application infrastructure and relationships to RA component classes guide decisions in selecting "how" a domain service in the RA can be satisfied by technology solutions in a particular application. The solutions may be chosen based on any number of design trade-off concerns (e.g. cost, availability, ease of implementation, etc.).

3.2. Distinguishing Characteristics

During requirements gathering for CBSE, it is essential to understand the interaction of the system to the environment, organizational issues, interface standards and guidelines. The SEPA methodology encourages commitment to requirements gathering and subsequent analysis. A user-centered KA effort and subsequent modeling of the elicited information dominate this stage.

In this section, the distinguishing SEPA characteristics are discussed in the context of how they support the "good" methodological features presented in Section 2. Three characteristics are emphasized in this discussion:

(i) requirements analysis prior to design,

(ii) the separation of domain and application requirements, and

(iii) requirements analysis for component-based development.

3.2.1. Support for Requirements and Design Evolution

The analysis process usually involves gathering requirements from a diverse group of system users and administrators. The SEPA methodology encourages a number of KA sessions with domain experts that are representative of all the contributing perspectives. Following the KA sessions, the information that was elicited is captured in KMs in any one of a number of graphical formats (e.g., process traces, collaboration diagrams, task

decompositions, etc.). These KMs naturally reflect the perspective, level of abstraction, and terminology of the domain expert from whom the information was acquired. To be useful to developers, the models need to be synthesized together to form a homogeneous representation of the system requirements.

Unlike prototyping methods, SEPA does not lend itself to committing to implementations early in the lifecycle. However, SEPA does not preclude the use of rapid prototyping techniques when the domain is well understood. SEPA, along with its supporting tools, helps in the identification of inconsistencies and incompleteness in the KMs and synthesizes the KMs into a complete, consistent unification. During the integration of KMs, the KE is asked to make several decisions, such as the relevance of certain details, the proper levels of abstraction, and what to do about inconsistent facts. The user, applying certain default heuristics (e.g., for the resolution of tasks defined with pre / post-condition mismatches, detection of differences in terminology, etc.), guides the integration process. User corrections are cached for future application and documentation of the evolutionary process. Given the amount of decisions to be made to completely integrate the entire set of KMs, it is unlikely to complete the synthesis of the DMs and ARMs in one step. SEPA suggests the creation of models that include increasingly larger subsets of the KMs. Each intermediary model must be consistent. If the subset of the KMs that an intermediate model includes is chosen carefully, it can represent the domain / application from the perspective of a category of domain experts[7]. The union of all of the intermediary models (and all of their KMs) will be a global portrayal of the domain, as a unified KM.

By focusing on and developing the analysis process, the SEPA methodology has been able to explicitly support requirements evolution. The SEPA methodology tool support more formally documents this evolution process and eases management of requirements artifacts and associated rationale during large and long-term projects(Rolland, 1994). The following subsections discuss how the SEPA methodology addresses requirements traceability, integration, and verification.

3.2.2. Support Communication Among Stakeholders

System stakeholders have varying notions on the purpose of requirements gathering and these differences may lead to communication problems. Often, the requirements gathering process is seen by the client as a negotiation process, wherein the last words have lasting ramifications throughout the life of the product, including its design and development stages. On the other hand, the developer would like for the requirements to be fully developed and fixed, and views any new requirements as the clients' change of "mind". However, these inevitable changes are often *corrections* that bring the written

[7] Note that SEPA does not force / promote a specific ordering process. This can be specific to the domain in which the process is being adopted. For example, an intermediary model may include all of the KMs resulting from KA sessions with system administrators.

requirements in-line with the conceptual requirements of the client. Another cause for the breakdown in communication is the presentation form of the requirements. Developers would ideally desire a complete formal specification document. Clients are often not trained in formal language specifications and would often prefer to leave some of the implementation details at the discretion of the developers. Natural language documents are most frequently used to specify requirements; however natural language is often too ambiguous[8], or, in an attempt to make a precise statement, it is made incomprehensible to the common reader[9]. The issues identified above are part of the "requirements gap" that separates the developer and the client. SEPA focuses on early analysis and definition of activity deliverables to directly bridge this gap. Specifically, this results in a better understanding of the conceptual requirements of the user.

3.2.3. Support for Traceability

SEPA provides traceability from the DMs back to the domain experts that stated the requirements. SEPA ensures that model synthesis, which occurs during DM creation, automatically preserves and extends traceability links into new design artifacts. Later, these traceability links can be leveraged to determine the necessary rework to be performed to implement a requirements change. In the near future, SEPA will allow the automatic propagation of slight changes in the KMs to the DMs.

3.2.4. Support for Component Integration

The promise of lower development costs and greater productivity, has resulted in the software engineering community placing considerable emphasis on component-based development and software architectures. Software architectures allow systems to be built from reusable components, to evolve quickly, and to be analyzed reliably (Clements, 1996).

A SEPA deliverable, the Reference Architecture, is comprised of a collection of domain-based component classes derived from information in the Domain Model. The Reference Architecture (RA) provides a key link between analysis and implementation, reflecting domain information found in the Domain Model and providing developers with a template for identifying and developing new technologies for particular implementations. In the process of creating the RA, the functional, procedural Domain Model is transitioned to an object-oriented Reference Architecture composed of technology independent domain components applicable to a "family" of applications in the domain. While it is feasible to partition RA components along either functional or object-oriented boundaries, an object-oriented approach promotes qualities such as

---

[8] There are at least three different, reasonable interpretations for the following requirement *"All the accounts contain the same security control field"*.

[9] An informal survey of 23 computer engineering graduate students resulted in less than half who correctly identified a formal, natural language specification for the minimum function.

reusability, faster development, flexibility, scalability, maintainability, and better correspondence to the domain being modeled (Graham, 1995). More importantly, by following an object-oriented approach, the delineation of components based on “what” services must be performed allows for flexibility and accommodates changes in “how” a task is performed when components are instantiated. Changes in the technology which perform a specific service will have minimal affect on the Reference Architecture if *how* the technology functions is not represented in the Reference Architecture.

An RA component class is assigned responsibilities based on domain tasks; the data/services required and provided by the component are declared according to the requirements of those tasks. Component behavior is then defined such that the component is able to satisfy all services provided. To accommodate this information, SEPA Reference Architecture components are represented by three categories of information (Graser, 1996): The Declarative Model (D-M) defines the attributes and services contained within and offered by a component. The Behavioral Model (B-M) defines the states of a component, the transitions between those states, and the events which affect transitions. The Integration Model (I-M) defines the constraints and dependencies between components, capturing integration requirements, potential subsystem compositions and user integration preferences. The D-M, B-M, and I-M combine to form a component definition based on "what" the component does, thus its definition is able to outlive various technology solutions as they become available.

Since RA component definitions should reflect responsibilities extracted from DM tasks, components must either be traceable to DM elements (e.g. tying an RA component to a DM task resource and/or concept) or be based on rationale introduced by the architect during construction of the RA (e.g. creating an abstract component from which other components inherit common data). DM information, in turn, is traceable to KMs (those KMs traceable to KA reports) and rationale introduced by the KE during DM development. This type of traceability ensures complete coverage of DM tasks and verification of the RA components. Furthermore, if changes are identified during system architecting or design, traceability can aid in determining what information in the DM may be affected, and whether new or modified component definitions require additional KA.

During the creation of the System Design Specification in the System Design stage, SEPA considers domain-based requirements modeled in the RA in conjunction with application-based requirements from the CARM. By considering what a system should be capable of delivering to end-users and how to deliver those capabilities in the context of technology and business constraints, the designer may be able to eliminate integration issues prior to implementation. Integration issues may take the form of implementation dependencies (e.g. dependency on COTS components) or domain dependencies between RA components (e.g. requiring domain data/service from another component). For example, information in the RA Integration Model ensures each component is provided all necessary data and services to perform selected tasks, while information in the CARM provides selection criteria for candidate technologies solutions to instantiate selected

components. If resolution of these is postponed until implementation, the correction may be more costly due to implementation commitments already made.

While some researchers advocate a phased approach to architectural design, experience has shown that architecture development is highly iterative and requires some prototyping, testing, measurement and analysis. The Software Engineering Institute suggests architects are influenced by factors such as (i) Requirements of the system (including required quality attributes). (ii) Requirements imposed by the organization (perhaps implicitly). (iii) Experience of the architect – the results of previous decisions, successful or not, will affect whether the architect reuses those strategies (Clements & Northrop, 1996). Furthermore, each stakeholder of a software system – customer, user, project manager, coder, tester, etc. – is concerned with different aspects of the system for which the architecture is an important factor. The evaluation of an architecture is usually focused not on its runtime aspects; but on qualities such as maintainability, including portability, reusability, adaptability, and extensibility (Clements & Northrop, 1996).

A fundamental objective of the SEPA RA development activity is to produce an optimal system architecture that exhibits qualities prioritized by the developer and system client, where priorities are set in accordance with the overall goals of the development effort (e.g., flexibility, extensibility, ease of installation, etc.). The overriding criteria, which characterize a "good" architecture often, vary between domains. As stated earlier, the developers select these criteria based on such concerns as: dynamics of the domain, turnover of domain experts, availability of technology solutions, and likelihood of change of application requirements. Given these concerns, criteria for a good architecture may include extensibility of the RA; comprehensibility of the KMs, DM, and RA; ability to adapt a system design specification for new technology offerings; and traceability of application requirements. SEPA systematically applies object-oriented (OO) heuristics and metrics to incrementally build a reference architecture reflecting developer goals.

An ideal design methodology will be flexible enough to incorporate rapid changes in technology by allowing engineers to take advantage of the best available technology. To achieve this flexibility, the SEPA methodology aids analysis without the biases and restrictions of premature implementation choices. Later, during System Design, matches are made between an appropriate technology solution to respective component requirements, associated application requirements, and subsystem constraints. As in any methodology, these matches are most valid when the requirements have been fully specified.

3.2.5. Support for Characterizing System Objectives

The SEPA methodology emphasizes the separation of requirements associated with a particular application from the requirements applicable to the domain in general. Recognizing the relative permanence of domain requirements with respect to system development, this emphasis on separation was made in an effort to facilitate reuse of domain analysis and design artifacts. That is, SEPA focuses the bulk of its analysis and design efforts on thoroughly modeling the set of abstract components in the domain.

Then, by applying the application requirements to that model, the desired application can be built. Moreover, the model can continue to be reused as a design template for building future applications for the same domain.

Figure 6 illustrates the separation of these two concerns. These are *(i)* Gathering and modeling domain information. *(ii)* Gathering and modeling specific application requirements. In practice, however, only after sufficient knowledge is acquired, KA reports generated, and KMs created to document the various views of information acquired, does the KE begin to separate application requirements from domain requirements. By making this distinction, the SEPA methodology is able to address the constant evolution of system requirements by encouraging the following:

1) As KEs gather and identify more domain information, the domain model becomes richer and more complete, gradually approaching an ideal DM. Due to the model's independence from application requirements and available technology solutions, it can outlive current development efforts.

2) Throughout the lifecycle, application requirements inevitably and continually change. These changes may be the result of the client and the developer reaching better mutual understanding of the issues involved, or they may be due to the introduction of new technology, thereby influencing design decisions. Since the emphasis is to separate application requirements, the impact of making modifications to the application requirements (regardless of the current status of the development effort) can be minimized.

While resolving integration issues is a difficult problem, making the distinction between application and domain requirements eases the resulting effects of the problem. Although this distinction may not contribute to the complete detection of all integration problems, it does help to guide their resolution when detected. Since such problems are

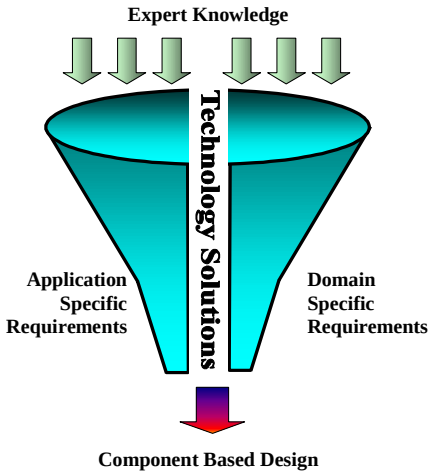


Figure 6: SEPA Separation of Application and Domain Requirements

often the result of conflicting requirements, it is critical to understand the rationale behind those requirements. While an application requirement may need to be renegotiated, a domain requirement involves a more fundamental challenge – i.e. perhaps the domain was misunderstood or this domain requirement will steer the development effort in a particular direction. Thus, the distinction between application and domain requirements greatly assists the development team with the resolution of integration issues, by identifying the cause and severity of these conflicts.

The SEPA methodology's tenet of separating application and domain requirements also contributes to verification. While verifying a complete model of a complex domain is difficult, the separation of requirements allows different experts to verify the portions that they best understand. On the other hand, if these requirements are all captured together, the question arises as to who verifies that necessary domain services were implemented. Furthermore, it must be determined if the same person is qualified to verify whether each of those domain services was implemented in accordance with the specific the application requirements requested. However, when these requirements are represented independently, a domain expert can verify compliance to domain services while an end-user can verify satisfaction of application requirements.

3.2.6. Support for Stepwise Verification

In SEPA, the ability to perform early verification of the derived KMs and DMs guides the development of the analysis process as well as the selection of artifact representations (multiple, intuitive, graphical notations). During KA, transcripts on KA sessions are verified with the domain experts to allow correction of mistakes, or to further elaborate on unclear points. These transcripts are used in the creation of KMs that are submitted to the domain expert for verification. After model integration, it can become unclear as to which domain expert should verify a model. For instance, a domain expert that only understands one of the contributing perspectives cannot verify models that capture multiple perspectives. Therefore, these integrated DMs are verified through the maintenance of strong ties back to contributing models that have been previously verified. In situations where model integration requires the intervention of the KE, "rationale at issue" points (Christel & Kang, 1992) are captured so that the source of any requirement can be determined.

SEPA tools provide a number of graphical views (task hierarchy diagrams, concept decomposition, Venn diagrams, etc.) to aid the knowledge engineer in modeling the domain. Several tools have already shown that graphical models can be used as an executable specification and to verify the conformance of current implementations (e.g. ObjectGeode (Verilog, 1997), Telelogics (Telelogics, 1997), Statemate (Harel, 1990)). Unlike other popular graphical notations, such as the UML (Rational, 1998b), the SEPA representations are more tolerant to incorrect syntax and incompleteness that necessarily occur during early knowledge modeling. By allowing these errors to occur and providing support for detecting and fixing them later, the KE is encouraged to begin building KMs earlier. This decreased amount of time elapsed between the KA session and the

subsequent modeling of information from that session provides stronger traceability and promotes easier verification and client validation.

4. SEPA Prototype Tools

Tool support cannot fully replace decisions and contributions provided by system clients, users, system architects, integrators, and developers. However, it can guide KEs, system architects, and developers through the development lifecycle while retaining traceability, documenting rationale for decisions, identifying inconsistencies, and applying metrics for evaluation. An important contribution of effective tool support is managing the large quantity of information generated during the development lifecycle.

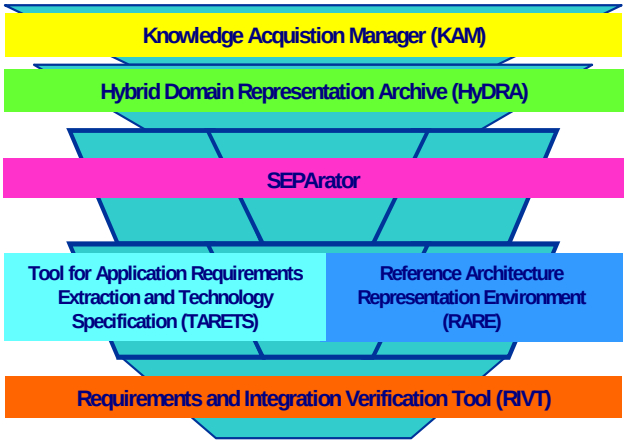


Figure 7: SEPA Tools

In this section, the suite of tools being developed to support the SEPA methodology is presented. The responsibilities for the various SEPA activities, as captured by the SEPA funnel structure (introduced in Figure 5), have been allocated to the tool suite. Figure 7 illustrates the SEPA funnel overlaid with the tools that support the respective SEPA activities.

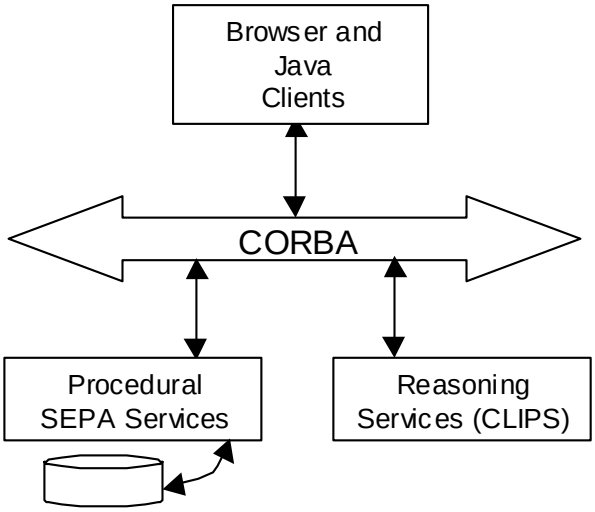
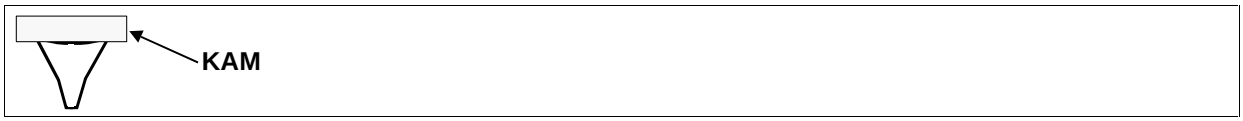


Figure 8: SEPA Tool Architecture

Figure 8 depicts the general architecture for the SEPA tools. The decisions made in the design of this architecture have taken into consideration a number of long-term implementation goals. Some of the most compelling reasons include the following. *(i) Leveraging existing standards*: avoid "reinventing the wheel" and use existing proven technologies wherever appropriate. *(ii) Scalability*: facilitate the development of additional SEPA tools and ensure that all tools in the suite are able to handle large, complex projects. *(iii) Portability*: provide the ability to run on a variety of client platforms. *(iv) Accessibility*: provide Internet accessibility to support large, geographically distributed development teams. *(v) Flexibility*: provide the ability to integrate disparate processing options (e.g. LISP, C++, COTS tools) to broaden the implementation options which can be considered for each SEPA process. *(vi) Integration*: facilitate information sharing among SEPA tools and integration with third party client tools.

Based on these issues, a CORBA-based architecture was chosen for the implementation. The CORBA backbone provides the flexibility to select the appropriate technology for each system component. For example, a web-based Java client was selected to realize client side portability and accessibility. SEPA's server side components leverage COTS database solutions for object persistence as well as CLIPS components for advanced reasoning.

The remainder of this section discusses the developmental status of these research tools and illustrates some of their features that support the SEPA methodology.



4.1. Knowledge Acquisition Manager (KAM)

The Knowledge Acquisition Manager (KAM) is a web based tool providing project management and document management functions (e.g. versioning, access control, change logs, etc.) for the Knowledge Acquisition (KA) process. KAM's knowledge acquisition project management facilities allow the knowledge engineers to (1) document KA plans, (2) specify and maintain participant (e.g. domain experts, end users, knowledge engineers) contact info and background profiles, (3) document intended KA session objectives, (4) document elicitation scenarios, and (5) upload session reports created in popular word processing or drawing programs which document the knowledge acquired during session and (6) reference shelf hard copy documents acquired during the KA session. Uploaded documents or referenced hard copy documents can be searched by content, organization, and perspective. When a new session is scheduled or a new session is documented and a report is uploaded, KAM can notify interested project participants and receive feedback requesting the knowledge engineer acquire or clarify knowledge about a particular area (e.g. "please clarify the contents of an incident report") or inquire about knowledge captured in a specific report such may assist the knowledge engineer in determining required follow-on sessions with the expert.

4.2. Hybrid Domain Representation Archive (HyDRA)

The Hybrid Domain Representation Archive (HyDRA) provides intelligent reasoning functions that guide the user during the creation of knowledge models (KMs) and unification of the KMs (Barber & Jernigan, 1999). Similar research efforts include (Sommerville, Sawyer, & Viller, 1998),(Pohl, 1996),(Leite & Freeman, 1991), and (Finkelstein, Gabbay, Hunter, Kramer, & Nuseibeh, 1994). HyDRA's objectives are to:

1. Aid the Knowledge Engineer (KE) by providing tool support for KA and modeling.

2. Automate the transition from unstructured, incomplete requirements to formal, complete, and consistent requirements.

3. Maintain traceability through the necessary merging of requirements from varying viewpoints.

The individual knowledge models need to be merged into a consistent, global model of the domain that represents the combined viewpoints of all domain experts. Currently, HyDRA provides a semi-formal and semi-automated mechanism through the iterative process of model integration and requirements refinement. This integration process identifies inconsistencies and incompleteness in the KMs.

As stated in Section 3, during the integration process, the user iteratively applies default heuristic rules. Corrections to the default rules are cached for future application and documentation of the integration process. Traceability is preserved across the integration of knowledge models and assists in the definition and validation of requirements from multiple knowledge sources.

Individual knowledge models are combined in incrementally larger and larger models. While HyDRA's primary deliverable is the one model that results from the combination of all of the knowledge models, the intermediate models are also of value. For instance, if an intermediate model is the result of all the knowledge acquisition sessions conducted with a particular expert viewpoint (e.g. physician, nurse, etc.), then that model represents the domain from that viewpoint. While these models were initially a byproduct of the synthesis process, they represent new artifacts, viewpoint-based models, that are only available through the automated synthesis. Viewpoint models are especially useful when developing components, such as GUIs, that are specific to a single type of domain user.

The primary deliverable produced by HyDRA is a Unified KM (UKM) synthesized across the viewpoints of multiple domain experts. Although the goal of a particular KA session may be to focus on the elicitation of one type of knowledge over another, a typical KA session may gather a broad spectrum of information. To reduce the risk of losing relevant information, the KE typically avoids forcing the DE to filter or abstract information. As such, the information gathered may contain domain knowledge, specific application requirements, and/or information about specific technologies currently in use (e.g., legacy systems). In future versions of HyDRA, the traceability links will be used to verify modifications to the UKM against the source knowledge models.

4.3. SEPArator

The SEPArator assists the Knowledge Engineer in separating the domain requirements from the application requirements in the UKM. After synthesis, the UKM will reflect

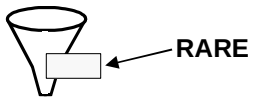
concepts from many different perspectives and may be too complex for a KE to navigate and evaluate without assistance. SEPArator navigates the UKM and applies a set of heuristics to classify requirements modeled in the UKM.

The domain requirements are the set of functional or data requirements shared by the entire domain. The basic rule for identifying domain requirements is to locate tasks and data identified by the Domain Experts. This task information is gathered together into the Domain Model (DM). The DM is used by the system architect to create the domain Reference Architecture, so it must contain enough information about the domain tasks to generate a set of end-user services for the system.

Application requirements reflect the details of a system implementation or the requirements of a particular client. In general, application requirements will be performance constraints on domain entities (tasks, roles, resources, etc.) or constraints on implementation choices. These requirements are gathered together into the Application Requirements Model (ARM), and are used during technology registration and system design to guide the creation of a system that satisfies the client's requirements.

Performance constraints are quantitative requirements on the domain entities irrespective of implementation strategy, e.g., speed constraints, numerical accuracy, etc. Implementation or delivery constraints specify user requirements on the implementation of the system or on the user interface with the system. For instance, a delivery constraint might be that the payroll system must be developed for a specific hardware platform and the system must interact with a particular legacy application. Another type of implementation constraint refers to user interface requirements. For instance, a particular application may be required to support federal accessibility guidelines, or adhere to the interface defined in a prototype model.

During Knowledge Acquisition, Domain Experts may begin to give the Knowledge Engineer information about specific technology solutions, such as COTS tools or legacy applications. A technology specification template is available to the Knowledge Engineer to assist in gathering application information during this phase. This structure of the software specification template is based on the IEEE 830-1993 Recommended Practice for Software Requirements Specification and extensive interaction with software developers and integrators. SEPArator extracts the application specification information from these models and makes it available to the technology registration tool in TARETS.



4.4. Reference Architecture Representation Environment (RARE)

The Reference Architecture Representation Environment (RARE) semi-automatically guides in the transition from the Domain Model (DM) produced by the SEPArator to a Reference Architecture (RA) by systematically applying object-oriented (OO) heuristics, software architecture heuristics, and quality metrics. The desirable qualities of object-orientation, such as extensibility, reusability, comprehensibility, and maintainability, come to fruition in an architecture as the result of prudent choices during the architecting process.

Relying on a complete, consistent Domain Model (DM) provided by HyDRA and SEPArator, the RARE process focuses on the allocation of domain tasks to responsible object-oriented component classes, Domain Reference Architecture Classes (DRACs). During the derivation process, RARE records the rationale supporting the architect's decisions. The resulting collection of DRACs comprise a domain-based, object-oriented Reference Architecture (RA) traceable to stakeholder requirements and offering the following benefits:

- *Provides a blueprint for developers:* The architecture provides a framework for development and is partitioned into classes based on object-oriented principles, providing well-established advantages such as design extensibility and reusability. Since the architecture is founded in models capturing domain tasks rather than system requirements, reusability is further enhanced because architecture services are independent of specific implementations.

- *Aids in domain understanding:* For developers trying to gain an understanding of the domain, architecture classes highlight domain tasks, performer roles and responsibilities, and the relationships between performer roles.

- *Identifies rules of composition among architecture classes:* Domain tasks often depend on each other for execution (e.g. it is necessary to receive data X before producing event Y). Based on domain task dependencies, the RA represents corresponding service and data dependency constraints between DRACs. When developers build applications intended to satisfy specific DRAC services, rules of composition utilize these constraints to highlight necessary implementation interfaces.

The DRAC representation is comprised of the following elements: (i) Declarative Model (D-M) - data owned and services offered; (ii) Behavioral Model (B-M) - an abstraction of component behavior represented as a state chart; and (iii) Integration Model

(I-M) - constraints between DRACs resulting from data/service dependencies and subsystem compositions (rules of composition). Although the derivation of components closely follows object-oriented principles, RA components are not necessarily realized one-to-one by implementation objects. RA components are purposely specified on a domain level to remain as independent from technology as possible. For example, three class definitions in a C++ implementation may cooperate to satisfy a single RA domain component; this represents just one of many possible implementations schemes.

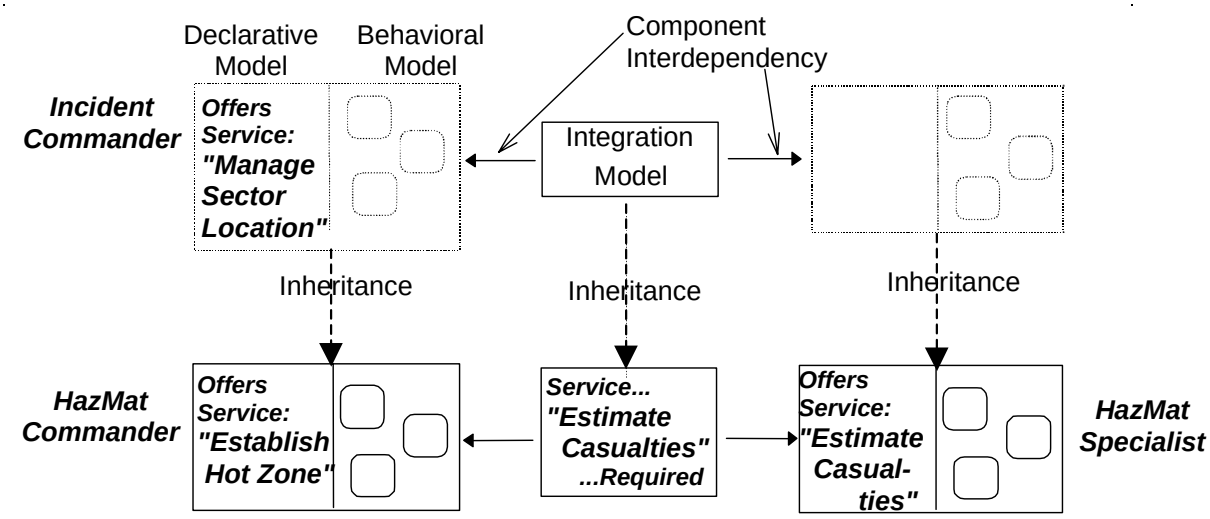


Figure 9: RARE Component Class Representation

The RARE DRAC representation is illustrated in Figure 9, annotated with example component classes and services from the incident response domain in bold italics. The *HazMat Commander*, responsible for managing responders during a chemical or biological release incident, inherits the *"Manage Sector Location"* service from the *Incident Commander* component. In addition, the *HazMat Commander* offers the *"Establish Hot Zone"* service, which is dependent on casualty estimate data provided by the *"Estimate Casualties"* service in the *HazMat Specialist* component. This dependency is represented in the combined Integration Models of the *HazMat Commander* and *HazMat Specialist* DRACs

RA derivation is an iterative process, where successive iterations represent increasing coverage of domain information and greater refinement of the RA based on user-established goals. Typical architecture goals include extensibility, comprehensibility, and maintainability, often reflecting the benefits associated with object-oriented approaches.

RARE guides the architect towards these goals through the application of architecture heuristics, architecture metrics, and heuristic strategies. These are described as follows:

- *Architecture Heuristic:* A "rule of thumb" compiled from expert experience on past projects which assists the architect in making rational decisions in defining RA components. One well-known object-oriented heuristic recommends reducing coupling among components to encourage reuse (Riel, 1997).

- *Architecture Metric:* A measurement of a particular characteristic of an RA which provides an indication whether the architect adhered to a given heuristic. Continuing with the previous example, the DRAC inheritance hierarchy and/or number of messages passed among DRACs would provide some evidence as to the degree of coupling in the RA (Whitmire, 1997).

- *Heuristic Strategy:* A step-by-step procedure (sequence of actions) used to apply a given heuristic. Following the "reduce coupling" example, a strategy might explicitly state, "move service S1 from DRAC D1 to DRAC D2" to eliminate the need to exchange data between DRACs D1 and D2.

4.5. Tool for Application Requirements Extraction and Technology Specification (TARETS)

The primary goal of TARETS is to:

1. create specifications of technology solutions (e.g. legacy systems, COTS applications, application systems or components under development, or planned systems or components) referred to as “technology registration” documenting WHAT functionality and data the application delivers and HOW the application is implemented.

2. model constraints for the delivery of those technology solutions. These delivery constraints specify user, organization and site specific implementation and installation constraints (e.g. user interface preferences for a domain-related requirement to display casualty data as well as operating system, memory, hardware platforms installation restrictions).

As the system designers and integrators seek determine which application components or systems can be deployed in a particular system configuration for a specific user installation site, they must determine:

- ability of applications to satisfy domain-related functional and data requirements (specified in the Reference Architecture, see Section 4.4)
- state of application (legacy custom system, COTS application, under development, planned)
- ability of application to met delivery expectations. Thus, the implementation specifics about an application as well as the expectations of installation sites must also be known.

TARETS holds information about an application with regard to all three aspects of an application and the delivery expectations of potential user sites.

The process of registering a technology solution is initiated by the identification of a new end-user application (application delivering services identified in the Reference Architecture). TARETS can initiate registration by recognizing a reference to an application in the AIRM (typically a resource required for domain related task), or a user can introduce a new application to TARETS through the registration tool. The next step will be to register the technology solution against the RA specifying WHAT functionality and data the application delivers. If the application appears in the AIRM, some preliminary information about the services offered may already be provided to promote automated registration. If not, the user must select which RA services are offered and which data or events are managed by the application. An application is *registered to a DRAC* if it offers any service or manages any data or events in the specification of that DRAC.

Once the end-user application has been identified, it must be classified according to a taxonomy of technology solutions. TARETS features an ontology of technology solutions to help facilitate the registration with regard to HOW the application is implemented. Each class of technology solutions has its own infrastructure requirements. Knowledge about the generic infrastructure requirements of technology solutions is represented in the ontology as Infrastructure Reference Architecture Classes (IRACS). For instance, the class of software instances known as “operating systems” require a piece of equipment called a “computing platform” and often have specific requirements on “instruction set” and “input/output devices.” Figure 10 shows a subset of the technology solutions ontology. Using the information from the ontology, TARETS can interrogate the user for more information about these infrastructure requirements. The requirements captured in the ontology may not always be valid for the specific application, but they provide a means of guiding the registration process.

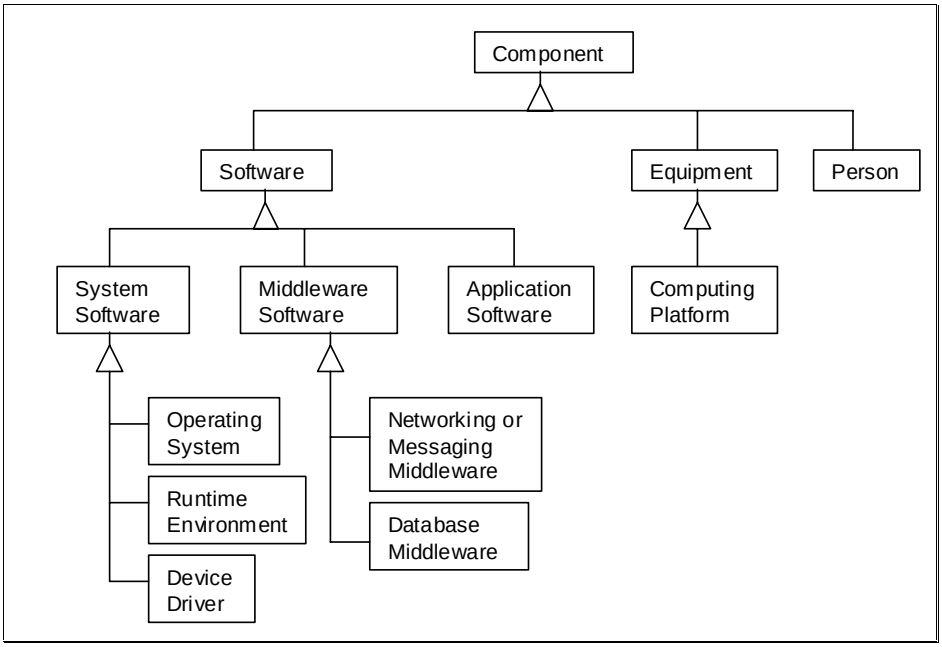


Figure 10: Subset of TARETS IRAC Ontology

Finally, information about additional application-specific constraints needs to be captured in the technology specification. For instance, if there is a constraint that a task X must be performed once every fifteen seconds, then the user must be asked a series of questions designed to obtain the application's ability to satisfy that constraint. For cases of notional applications, the user is offered the opportunity to declare that the application satisfies the constraint. In that case, the property becomes a part of the design specification of that application.

Delivery expectations with regards to specific users, organizations or installation sites are modeled as additional constraints related to services and data contained in the Reference Architecture (e.g. performance constraints on DRAC services or UI display of DRAC data) and installation infrastructure constraints specified and modeled using the IRAC representation.

Specifying the application capabilities and delivery expectations using the same representation constructs allows automated evaluations of compatibility.

4.6. Requirements Integration and Verification Tool (RIVT)

RIVT assists users (e.g. system architects, designers, and integrators) in (1) identifying and evaluating application technologies against domain, application and system infrastructure requirements and (2) evaluating the viability of integrated component application configurations. The resulting System Design Specification (SDS) contains a set of application component solutions, system infrastructure components (e.g. COTS tools, middleware, and hardware), and the integration dependencies between respective components and between components and the infrastructure. With the help of RIVT, the user may identify multiple system specifications that adequately satisfy requirements. These specifications provide guidance for further design and implementation decisions and encourage trade-off analysis and evaluation.

As application components are selected from the Technology Solutions Repository (TSR) and integrated into a prospective system design, rules are fired to ensure that their addition to the design will not violate application and infrastructure constraints in the CARM or DRAC interdependency requirements in the RA. The log of rules fired during a design session yields an additional product of RIVT, a configuration design rationale.

In addition to supporting system integration activities, RIVT also provides a reporting facility allowing users to browse the CARM, TSR, and RA, highlighting the relationships between application and infrastructure requirements, application component solutions, and domain functional and data requirements. With accessibility to the complete set of requirements represented and applications registered, the RIVT reporting facility is capable of supporting a number of analysis activities: (1) application impact change analysis, (2) investigation of possibilities for requirements reuse, (3) verification of configuration support for selected usage scenarios, and (4) evaluation of the viability of proposed configurations.

5. SEPA Requirements Management and Reuse Example

SEPA is being applied in a number of domains. Among them is a Defense Advance Research Project Agency (DARPA) initiative to develop information systems supporting first responders (fire department, police, EMS) during emergency situations (e.g. fire, natural disaster) and chemical/biological (chem/bio) warfare attacks that may occur at large public events (e.g. rock concerts, political gatherings). In today's emergency response environment, most incident types are initially managed by "first responders". The process by which the incident is managed is locally defined and evolves during the course of the incident, based on pre-defined Standard Operating Procedures and local

resource availability. The system under development will have functionality including responder task assignment and location tracking, casualty assessment, chem/bio agent analysis, and post-incident reporting and analysis. A key mandate of the resulting system is that it be flexible enough to support a wide variety of incident response facilities and organizations. This mandate will demand a significant amount of application reuse and system customizability. The requirements of each facility must be evaluated in the context of the system and incident response domain and satisfied by associated applications, making reuse at the requirements level the clear approach to multi-site implementation.

5.1. Knowledge Acquisition

The first step in gathering requirements is to determine the viewpoints or perspectives that must be considered when designing the system (Sommerville & Sawyer, 1997). Sommerville suggests that these viewpoints be gathered into a stakeholder viewpoint hierarchy. SEPA's Knowledge Acquisition Manager (KAM) tool actually uses two orthogonal hierarchies. The first hierarchy is used to describe the domain perspectives (with leaf nodes such as HazMat Incident Commander or Fire Administrator). The second hierarchy is used to describe the organizational perspectives (with leaf nodes such as City Y Fire Department or City X Police Department). While the KAM tool uses both perspectives simultaneously, this example will only use the shaded portion of the simple domain viewpoint shown in Figure 11.

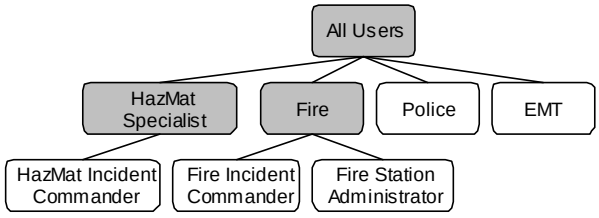


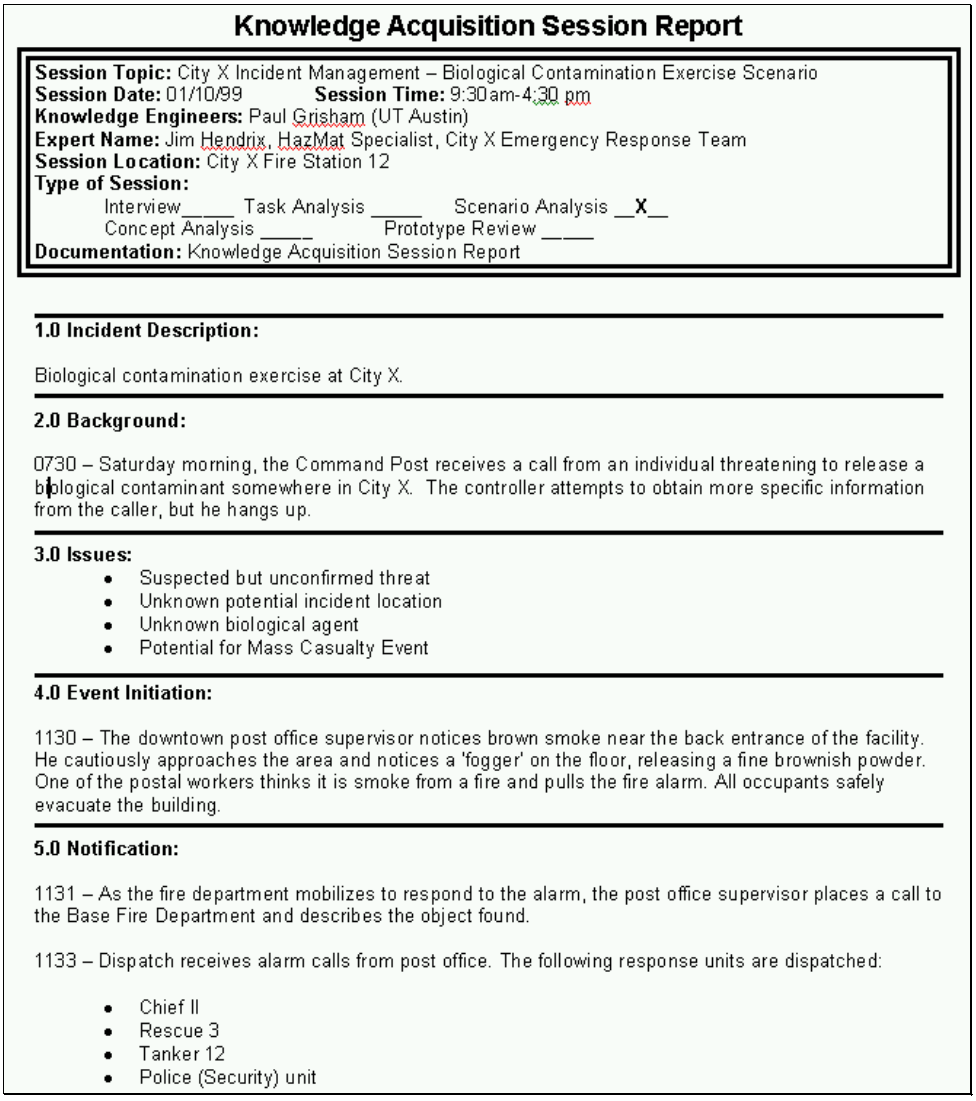
Figure 11: Example domain viewpoint hierarchy

Domain experts for this project were selected so that a variety of expert viewpoints were represented, which for our example includes fire personnel and hazardous material specialists. The KAM tool was used to maintain contact information on project participants and allowed these domain experts to be assigned to leaf nodes of two hierarchies based on their domain perspective and organization perspective. KAM was also used to schedule several KA Sessions with these experts and can record session information such as session goals, times, dates, locations, participants, and KA approaches. Although KEs used a variety of approaches to elicit domain and system requirements, scenario analysis was the primary approach used to acquire of domain task, performer, and timing information (Harbison, 1997). Other KA Sessions focused on acquiring specific implementation requirements through prototype review, yielding information about preferred look-and-feel as well as installation specifics such as required operating system. The SEPA example which follows is rooted in the information acquired from the sessions listed in Table 1.

Table 1: KA Sessions for Incident Response Example

Session	Expert / Viewpoints	Location	KA Approach	Information Gathered
1	Jim Hendrix • Hazardous Materials Specialist • HazMat Incident Commander	City X, City Response Center	Scenario Analysis	Tasks, performers, and resources for chem/bio response
2	Sam Cook • Fireman • Fire Incident Commander • Fire Station Administrator	City Y, Fire Station 1	Scenario Analysis	Tasks, performers, and resources for fire response
3	Bob Smith • Fireman • Fire Incident Commander	City Y, Fire Station 2	Prototype Review	User Interface (UI) requirements and specification installation needs for fire response

To document each session, Knowledge Engineers (KEs) created one or more KA Session Reports, accompanied by supporting diagrams, videos, and supplemental documents. KA Session Reports represent knowledge from domain and organization perspectives associated with the domain experts involved in each session. Excerpts from the session reports corresponding to the sessions in Table 1 are depicted in Figure 12 – Figure 14.



Knowledge Acquisition Session Report

Session Topic: City X Incident Management – Biological Contamination Exercise Scenario
Session Date: 01/10/99 **Session Time:** 9:30am-4:30 pm
Knowledge Engineers: Paul Grisham (UT Austin)
Expert Name: Jim Hendrix, HazMat Specialist, City X Emergency Response Team
Session Location: City X Fire Station 12
Type of Session:
Interview____ Task Analysis _____ Scenario Analysis __**X**__
Concept Analysis _____ Prototype Review ____
Documentation: Knowledge Acquisition Session Report

1.0 Incident Description:

Biological contamination exercise at City X.

2.0 Background:

0730 – Saturday morning, the Command Post receives a call from an individual threatening to release a biological contaminant somewhere in City X. The controller attempts to obtain more specific information from the caller, but he hangs up.

3.0 Issues:

- Suspected but unconfirmed threat
- Unknown potential incident location
- Unknown biological agent
- Potential for Mass Casualty Event

4.0 Event Initiation:

1130 – The downtown post office supervisor notices brown smoke near the back entrance of the facility. He cautiously approaches the area and notices a 'fogger' on the floor, releasing a fine brownish powder. One of the postal workers thinks it is smoke from a fire and pulls the fire alarm. All occupants safely evacuate the building.

5.0 Notification:

1131 – As the fire department mobilizes to respond to the alarm, the post office supervisor places a call to the Base Fire Department and describes the object found.

1133 – Dispatch receives alarm calls from post office. The following response units are dispatched:

- Chief II
- Rescue 3
- Tanker 12
- Police (Security) unit

Figure 12: Incident Response KA Session 1 Report

Knowledge Acquisition Session Report

Session Topic: City Y Incident Management - Dumpster Fire Scenario
Session Date: 01/05/99 **Session Time:** 9:30am-11:30 am
Knowledge Engineers: Tom Graser (UT Austin)
Expert Name: Sam Cook, Fire Chief, City Y Fire Department
Session Location: City Y Fire Station 2
Type of Session:
Interview____ Task Analysis _____ Scenario Analysis __X__
Concept Analysis ____ Prototype Review ____
Documentation: Knowledge Acquisition Session Report

Incident Description:
Dumpster Fire

Incident Setting:
Downtown area near the university main campus. High traffic freeways are nearby and side streets are congested with pedestrians. The secondary roads surrounding the accident are also congested as residents attempt to bypass the freeways to reach downtown businesses.

Features of the Scenario:
- Local dispatch has 911 *enhanced* capability
- Dumpster is part of an adjoining structure

1.0 Notification
A resident of a university dorm calls 911, reporting a fire in a nearby trash dumpster. The dumpster is part of an adjoining building that is believed to be unoccupied. The dispatcher queries the caller regarding the exact location of the dumpster and any known details regarding the fire. The following information is conveyed:

Call in: 2:45 PM
Estimated time of event: 2:40 PM
Address: 9th Street and Main Street
Problem: Fire and smoke coming from a trash dumpster
Known Injuries: Unknown
Caller: John Smith, 555-1234

2.0 Mobilization
Dispatcher confirms caller location via the Computer Aided Tracking (CAT) system. Once the incident location is confirmed, a request for response is sent (via radio) to the fire and police departments serving the area. The following resources are dispatched to the scene based on the information provided by the caller (shifts and associated units identified to dispatcher periodically):
- 1 Fire Engine
- 1 Fire Truck (pumper)
- 1 Police Patrol Unit (Local)

Figure 13: Incident Response KA Session 2 Report

Knowledge Acquisition Session Report

Session Topic: Responder Tracking for Incident Response
Session Date: 01/01/99 **Session Time:** 9:30am-11:30 am
Knowledge Engineers: Steve Jernigan (UT Austin)
Expert Name: Bob Smith, Fire Chief, City Y Fire Department
Session Location: City Y Fire Station 1
Type of Session:
Interview_____ Task Analysis _____ Scenario Analysis _____
Concept Analysis _____ Prototype Review _____
Documentation: Knowledge Acquisition Session Report

General Topic Area:
Prototype implementation for responder tracking.

Prototype Screen:

Prototype Description:
Map-based approach for assigning responders to various incident hotspots and tracking their location as an incident progresses.

Specific Implementation Requirements:
- Operating System: Windows NT v4.0 SP4

Figure 14: Incident Response KA Session 3 Report

Knowledge Engineers use KAM to (1) identify domain experts, (2) define domain and organization viewpoints, (3) define an overall KA session plan, and (4) store the products from each session. Figure 15 shows the KA session plan in KAM referring to the sessions in Table 1.

The documents maintained in KAM are accessible by all project personnel through the web. Project participants can perform content-based searches with filters based on the domain and organization perspectives. Session reports, diagrams, videos, and supplemental documents provide a foundation for requirements traceability.

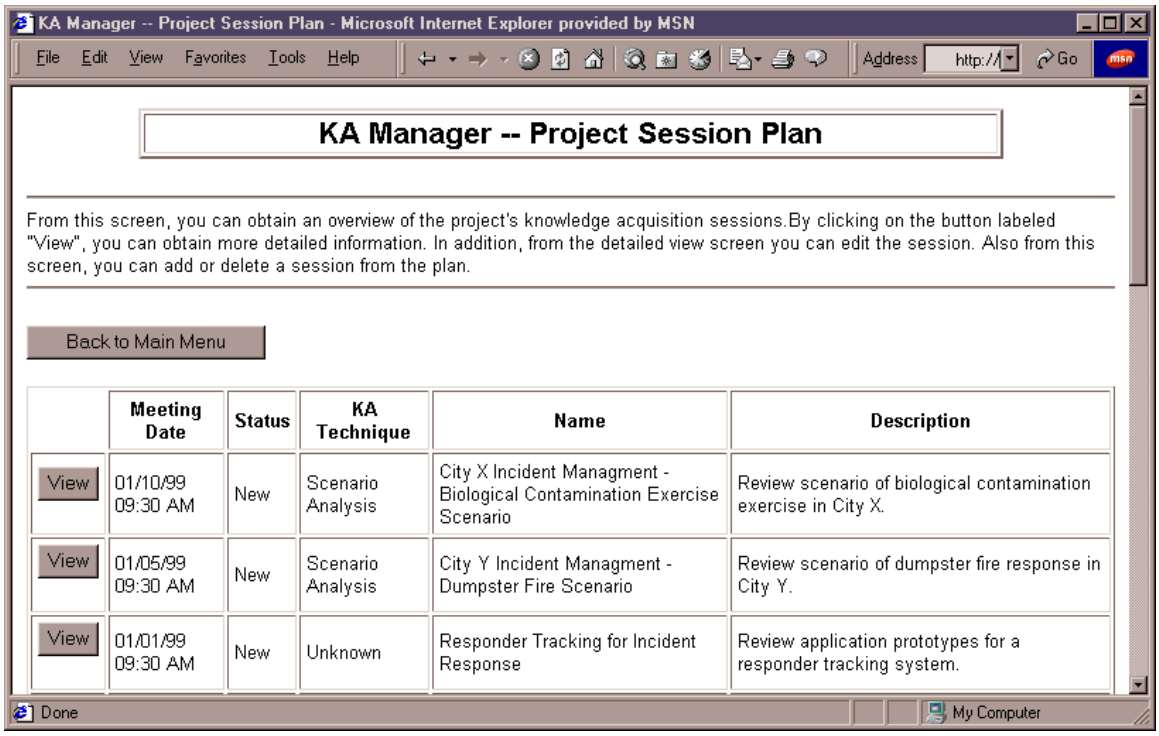


Figure 15: KA Session Plan for Incident Response Example

5.2. Knowledge Modeling

Information found in KA session reports and related documents is not structured, making effective reasoning by a computer difficult. To transition KA artifacts to a computational representation, the KE interprets the artifacts and creates structured graphical and textual Knowledge Models (KMs) in HyDRA. While the ultimate goal is to use the modeling capabilities of one or more of the commercially popular CASE tools, integration difficulties and perceived shortfalls in traceability and change management motivated the implementation of a few modeling tools within HyDRA. Currently, the KE has the choice of a number of models, including data flow diagrams, task decomposition diagrams, task templates, Venn diagrams, and entity-relationship diagrams (concept maps). Appropriate knowledge models are selected based on the type of knowledge acquired. For example, task decomposition diagrams provide an overall view of domain tasks and subtasks, while a task template contains specifics about the data, timing, and performance requirements for a specific task.

To facilitate the validation of KA artifacts with domain experts, each KA session yields a standalone collection of knowledge models called a Model Space (MS). If information from multiple experts was combined into a single knowledge model, the model would reflect a hybrid of the viewpoints from those experts, and no single expert would be able to validate and sign off on their contribution. Furthermore, the rationale used by the KE in merging the information from multiple experts would not be captured, compromising traceability.

The resulting knowledge models generated from the sessions in Table 1 are summarized in Table 2 below. Figure 16 - Figure 18 show screen shots of three sample knowledge models. Figure 19 - Figure 24 highlight the important details of the remaining example knowledge models in a condensed format.

Table 2: Knowledge Models from Incident Response Example

Knowledge Model	**Source KA Session**	**Data Represented**
Venn Diagram (Figure 18)	1	Attributes composing the "weather" concept.
Task Decomposition (Figure 19)	1	Task decomposition for chem/bio incident response.
Task Templates (Figure 20)	1	Selected task details for chem/bio incident response tasks, including pre/post conditions, performer, and input/output data.
Task Decomposition (Figure 21)	2	Task decomposition for small fire incident response.
Task Templates (Figure 22)	2	Selected task details for small fire incident response tasks, including pre/post conditions, performer, and input/output data.
Task Performance Constraint Template (Figure 23)	3	Prototype presented to expert is associated with respective domain tasks as suggested implementation approach.
System Constraint Template (Figure 24)	3	Overriding system implementation constraints (e.g. operating system).

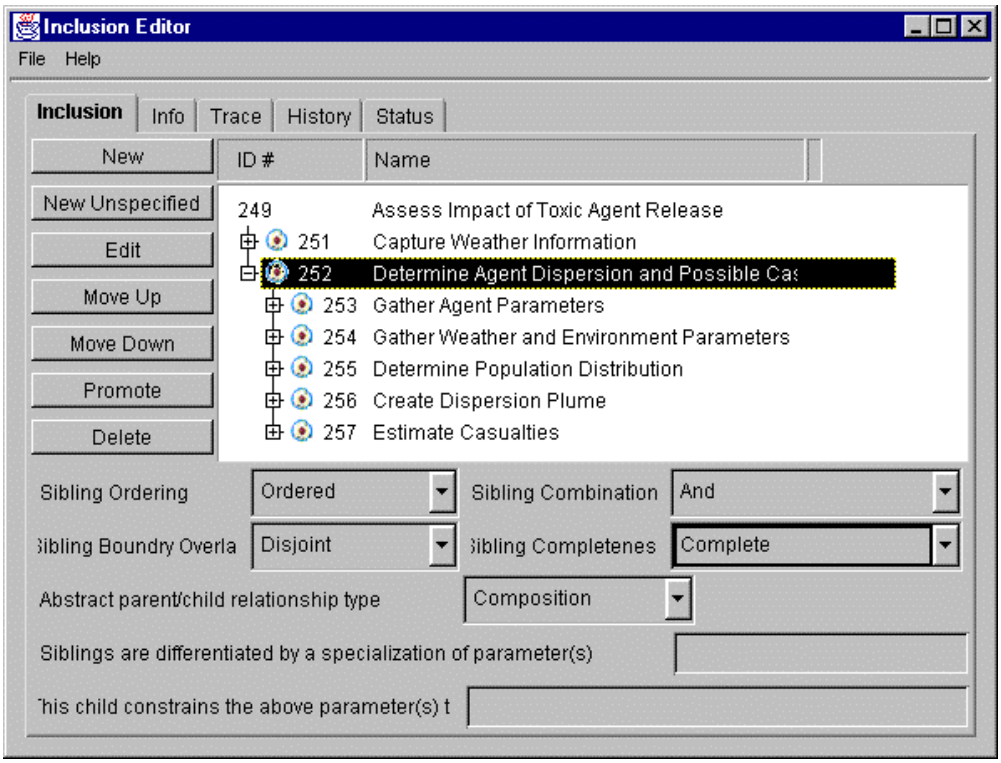


Figure 16: Incident Response Task Decomposition Example in HyDRA

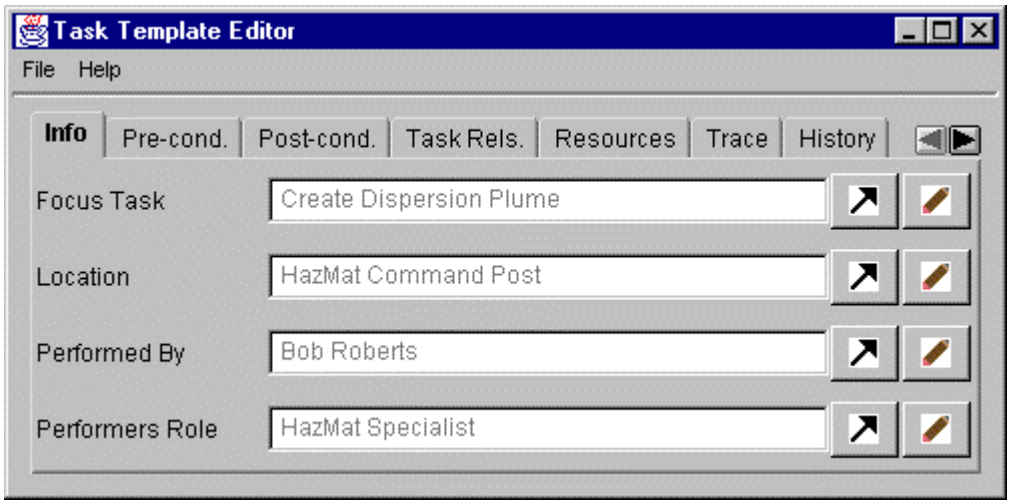


Figure 17: Incident Response Task Template Example in HyDRA

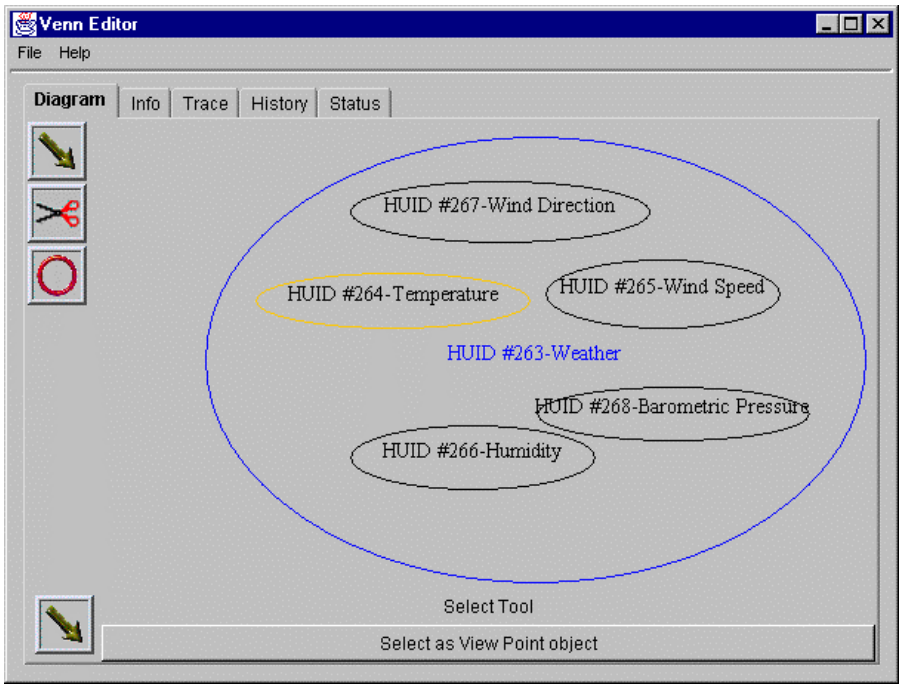


Figure 18: Incident Response Venn Diagram from KA Session 1 in HyDRA

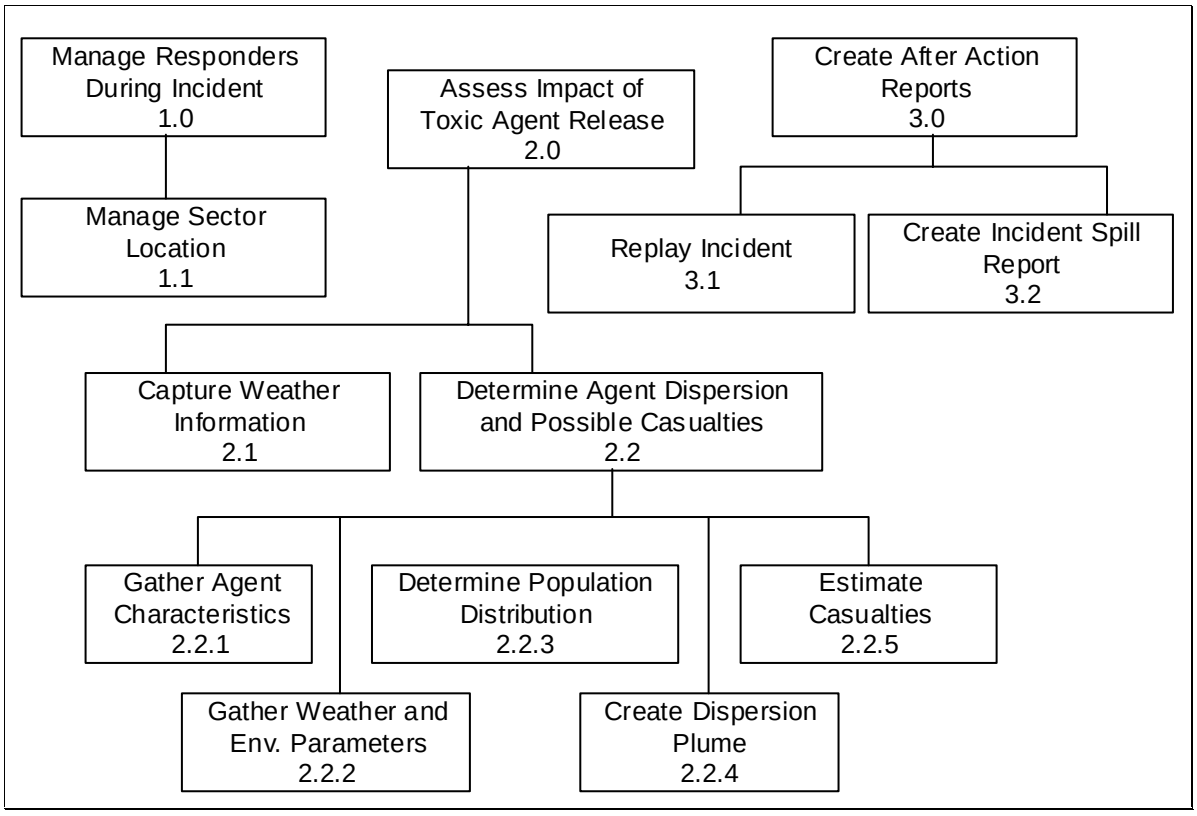


Figure 19: Incident Response Task Decomposition from KA Session 1

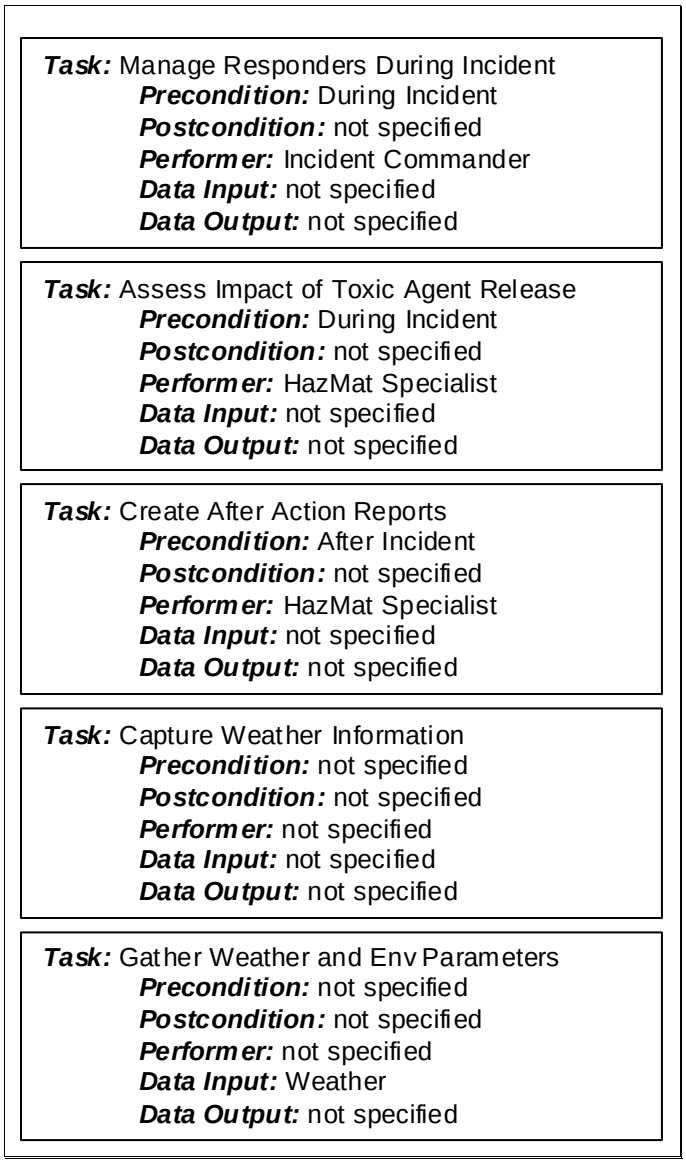


Figure 20: Incident Response Task Templates from KA Session 1

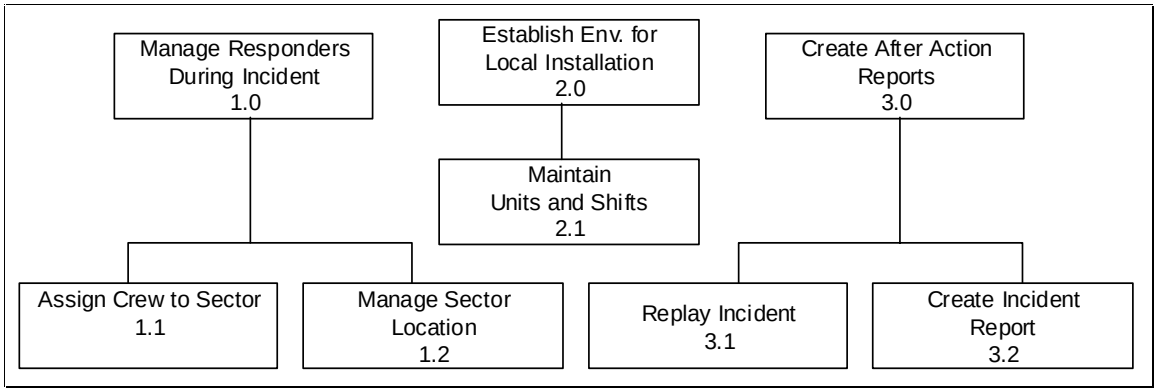


Figure 21: Incident Response Task Decomposition from KA Session 2

Task: Establish Env. for Local Installation
Precondition: Before Incident
Postcondition: not specified
Performer: Fire Chief
Data Input: not specified
Data Output: not specified

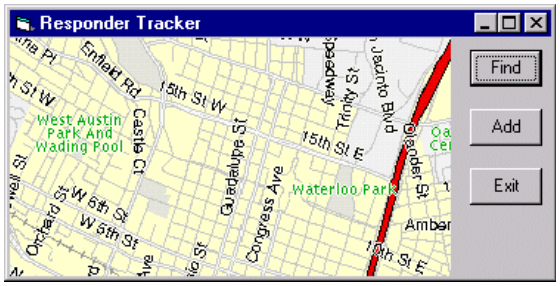
Task: Manage Responders During Incident
Precondition: During Incident
Postcondition: not specified
Performer: Incident Commander
Data Input: not specified
Data Output: not specified

Task: Create After Action Reports
Precondition: After Incident
Postcondition: not specified
Performer: Fire Chief
Data Input: not specified
Data Output: not specified

Figure 22: Incident Response Task Templates from KA Session 2

Task: Manage Responders During Incident

Interface Prototype:



Prototype Description: Map-based approach for assigning responders to various incident hotspots and tracking their location as an incident progresses.

Figure 23: Incident Response Task Performance Constraint Template from KA Session 2

Implementation ID: City Y Fire Dept.

Constraints:

Operating System: Windows NT

Figure 24: Incident Response System Constraint Template from KA Session 2

5.3. Knowledge Modeling Synthesis

To achieve a single, unified picture of requirements, HYDRA assists the KE in synthesizing requirements gathered from multiple experts, following a hierarchy of viewpoints defined for a domain. In this example, all models from "Hazardous Materials Specialist" experts would be merged into a "Hazardous Materials Specialist" model space (step 1 in Figure 25), and all models from "Incident Commander" experts would be merged into an "Incident Commander" model space (step 2 in Figure 25). These models space would subsequently be merged into a unified model space, the Unified Knowledge Model (UKM) (step 3 in Figure 25).

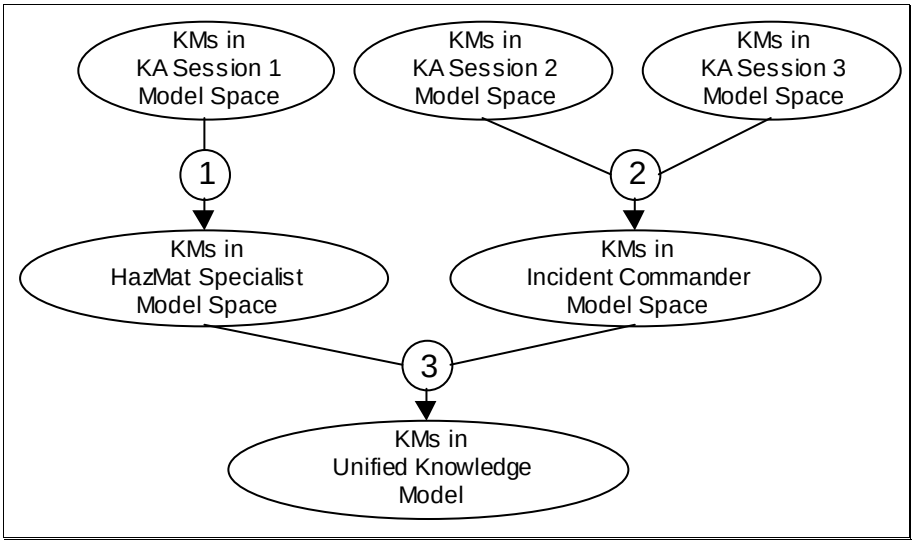


Figure 25: Incident Response Knowledge Synthesis Process in HyDRA

As synthesis proceeds, HyDRA detects conflicts between the knowledge models and presents the KE with possible resolutions. For example, when the ordering of two tasks vary between two models, the KE must determine which usage to retain or to mark the task unordered. As each decision is made, HyDRA ensures the KE's rationale is captured. Traceability is retained to record how model elements were changed or merged to produce the resulting element. Details regarding the synthesis operation can be found in (Barber & Jernigan, 1999).

Conflicts detected in this example along with their selected resolutions are:

1. *Task with almost identical children:* Task 3.0 in Figure 19 and task 3.0 in Figure 21 have the same name and share a common child. However, the other child (3.2) is different. During the merge, HyDRA detects this situation and asks the KE if it should (i) keep all three children, (ii) ignore one of the dissimilar children, or (iii) merge the two dissimilar children into one task. In this case, the KE recognized that these are the same task and asked HyDRA to merge the two dissimilar tasks into one with the name "Create Incident Report". Alternatively, the difference between the children of task 1.0, "Manage Responders During Incident", of the two task decompositions will not create a problem if the task decomposition from the first KA session, the session with only one child task, is marked with an open world semantics or incomplete flag.

2. *Type - usage conflict:* The Venn diagram for "Weather" indicated the concepts in the diagram were "states" (not shown in panel exposed in Figure 18). However, the "Weather" concept was used in a task template knowledge model as a "resource".

HyDRA detects this usage as being in consistent with the type provided by the Venn diagram. Since, the knowledge models were validated by the domain experts before the merge, HyDRA cannot change them and preserve the validation. However, it can ask if the user would like to ignore the conflict or ignore either the usage of "Weather" in the task template or the type information from the Venn diagram. Finally, the user has the choice of changing a source knowledge model, revalidating it with the domain expert, and re-performing the merge process. In the example, the KE choose to change the Venn Diagram so that it indicated that the concepts where "information", a non-volitional, intangible, consumable kind of resource.

3. *"Weather" not created:* As part of the last steps of a merge, HyDRA attempts to perform some completeness checks. In this case it finds that "Weather", as a consumable resource, is not ever created before being used. In response, the KE adds "Weather" as a Data Output on the "Capture Weather Information" task and re-merges.

The knowledge models must be translated into a common representation during the merging process. HyDRA uses a semantic net for this representation. Concepts become nodes in the net. Figure 26 shows a portion of semantic net resulting from the "Incident Commander" model space merge (step 2 in Figure 25). The concepts are connected to each other via relations. Concepts and relations can inherit from an ontology of domain unspecific concepts (e.g., mental task, non-consumable resource, post-condition) which constrain their use in other relations (i.e., a concept that plays the role of a "performer" must be an "agent").

The concepts shown in the Figure 26 are those most relevant to the "Create Dispersion Plume" task. The type information and links to concepts in the ontology have been hidden to make the figure more readable. The concepts in the upper right quadrant of the picture result from the task decomposition shown in Figure 19. The "Weather" decomposition from the Venn Diagram in Figure 18 can be see in the lower left. Just above the "Weather" concepts is a decomposition of a state ("state853") that occurs when the task is being executed. The decomposition denotes that two resources must be available; the "Weather" resource and a "HazMat Specialist" who also happens to be the task performer. The bottom right of the figure shows that the task has a postcondition that the "Plume Characterization" resource is available and that the task occurs at the "HazMat Command Post."

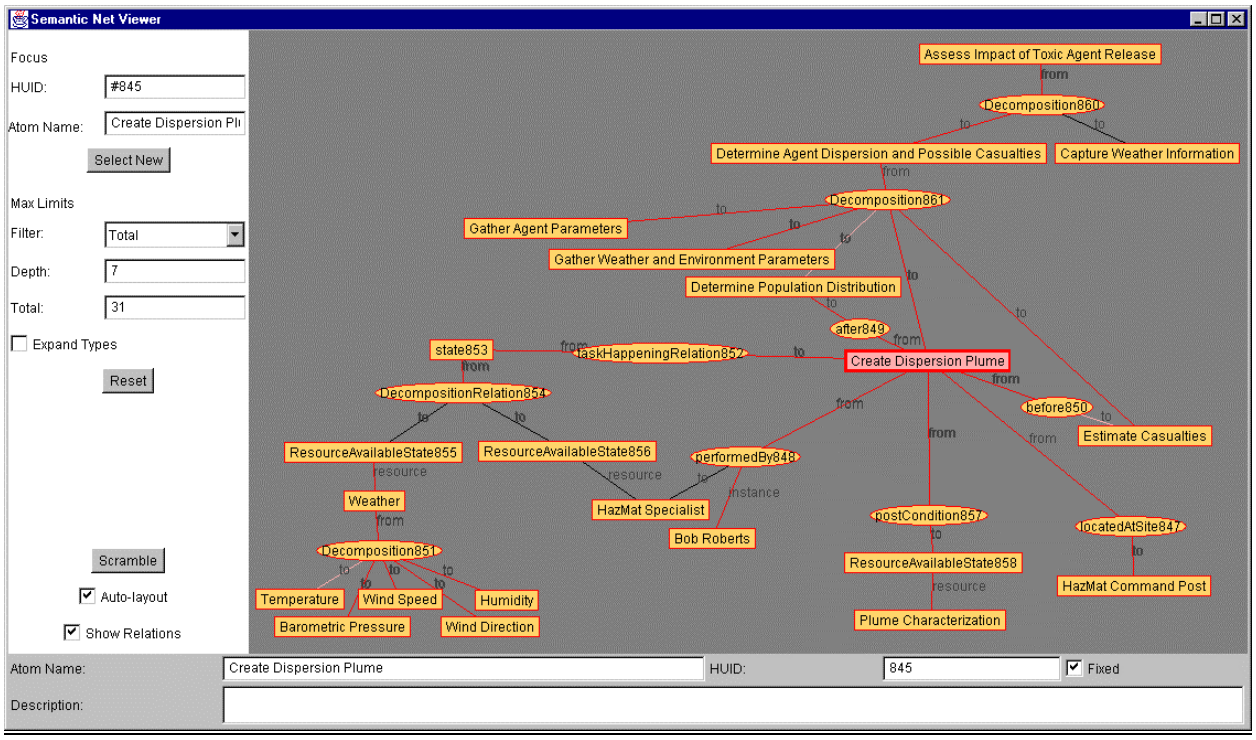


Figure 26: Incident Response Unified Knowledge Model in HyDRA

Following the synthesis operations depicted in Figure 25, the incident response UKM would include concepts from the following knowledge models in its semantic net. The merged task decomposition information in the UKM is illustrated in Figure 27 for clarity.

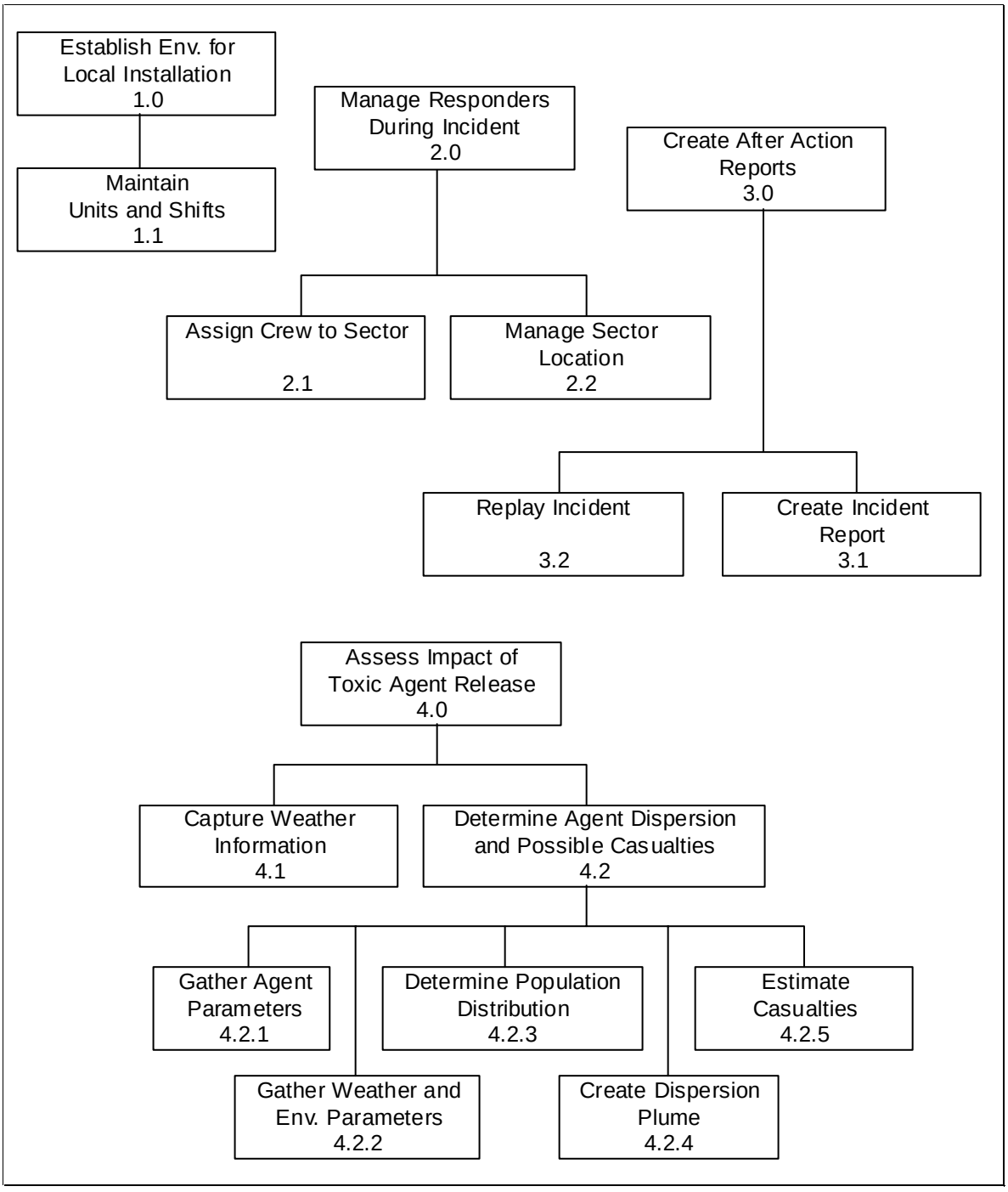


Figure 27: Merged Incident Response Task Decomposition from UKM

5.4. Separation of Domain and Application Requirements

Once the UKM has been constructed, the SEPArator must separate domain and application requirements. The domain content in the UKM, originally represented as concept models (such as the Venn Diagram) and task-based models, is assembled to form the Domain Model (DM). The content represented by the task performance constraint models and the system constraint models are used during technology registration. If any of the DEs had made specific reference to a particular technology solution (such as a software application), specification templates for these entities would also be sent to the technology registrar. In this example only domain requirements were elicited from KA sessions 1 and 2. Application requirements were elicited from KA session 3.

5.5. Reference Architecture Derivation and Application Registration

The RARE tool assists in the translation from the function-based Domain Model (DM) to an object-oriented domain Reference Architecture (RA). This translation involves the creation of Domain Reference Architecture Classes (DRACs) and the allocation of tasks from the DM to specific DRACs. The RA serves as a blueprint for developers, where each DRAC is a specification containing domain data owned, domain services provided, data and event exchange with other DRACs, and subsystem participation (see Section 4.4). In addition, the DRAC hierarchy and associations in the RA help guide developers in finding the appropriate DRAC when registering their applications.

As class derivation and task mapping take place, rationale is captured and traceability to DM tasks and other elements is established. A complete RA accommodates all leaf tasks (having no subtasks) in the DM. New domain information (i.e. from further KA) would not be introduced during this process without going through previous SEPA phases. Domain tasks can be allocated among classes in a number of combinations. As with traditional object-oriented development, the identification of classes is more of an art than a science. To drive the allocation process, SEPA's Reference Architecture Representation Environment (RARE) follows a set of high-level goals prioritized by the architect. Goals may have conflicting implications. For instance, the architect may select reusability to be the highest priority, but the derivation heuristics associated with other goals may conflict with those associated with reusability. In addition to the overriding goal increase reusability, four other goals are at play for this project:

1. *Aligning with performers in the domain:* Many domain tasks will remain un-automated, thus the collections of tasks should closely follow the tasks assigned to domain performers (e.g. Incident Commander).

2. *Aligning with existing COTS applications:* To maximize reuse of existing COTS applications capable of performing domain tasks, tasks should be grouped based on those tasks associated with existing applications.

3. *Increasing installation customizability:* To increase customizability for each installation, it is recommended to reduce the number of services offered by each

DRAC and thus increase the total number of DRACs. In general, a greater number of smaller DRACs can be combined in more arrangements to support specific installation requirements.

4. *Aligning based on when tasks are typically performed during incident response:* Responders only concern themselves with a certain set of tasks during any particular time period of an incident. For example, tasks such as assigning crews to shifts and defining resources (e.g. fire trucks) are done outside of any particular incident. Tasks in different time periods naturally have fewer coupling constraints than tasks in the same time period. Therefore these large grained time periods provide guidance for the assignment of tasks to DRACs so that inter-DRAC coupling and the number of DRACs the responder must interact with during any given period are reduced.

The process of associating a new, existing, or proposed (notional) application with DRAC data and functionality is referred to as "registration." Since a primary goal of this development effort is to reuse COTS tools when possible, registration involves both COTS applications and newly developed applications. The applications under consideration for this example are described as follows:

- FDManager v2.0 – A COTS application that provides a complete set of services to support small and midrange incidents. These services include resource allocation and responder assignment and tracking.

- IncidentReporter v2.0 – A newly developed application that provides analysis and government-mandated reporting after an incident has completed.

- PlumeAnalyzer v1.0 – A COTS application used to predict casualties from a chem/bio agent release based on agent characteristics, population distribution, weather, and other environmental conditions.

- ResponderLocator v1.0 – A newly developed map-based application used to assign tasks and locations to responders and monitor their location.

- WeatherMonitor v0.7 – A newly developed application that provides an interface for collecting weather data and disseminating this data to multiple responders.

The following lists the RA DRACs derived from the incident management DM and identifies the applications registered to DRAC services. Rationale is provided with each DRAC to describe the basis for DRAC creation.

DRAC 1: **Administrator**

Rationale for creation:

- *Aligning based on when tasks are typically performed during incident response:* Collect all tasks performed outside of an incident.

DRAC Service	Registered Applications
Maintain Units and Shifts	*FDManager v2.0*

DRAC 2: **Incident Manager**

Rationale for creation:

- *Aligning with performers in the domain:* The domain role of Incident Commander is significant.

DRAC Service	Registered Applications
Assign Crew to Sector	*FDManager v2.0* *ResponderLocator v1.0*
Manage Sector Location	*FDManager v2.0* *ResponderLocator v1.0*

DRAC 3: **Weather Manager**

Rationale for creation:

- *Increasing installation customizability:* Weather data should be independently managed from other types of data so data handlers can be combined in different ways for different installations.

DRAC Service	Registered Applications
Capture Weather Information	*WeatherMonitor v0.7*

DRAC 4: **After Incident Reporter**

Rationale:

- *Aligning based on when tasks are typically performed during incident response:* Collect all services occurring after completion of an incident.

DRAC Service	Registered Applications
Create Incident Spill Report	*IncidentReporter v2.0*
Replay Incident	*IncidentReporter v2.0*

DRAC 5: **HazMat Manager**

Rationale:

- *Aligning with performers in the domain:* Services follow those tasks performed by HazMat Specialist in UKM.
- *Aligning with existing COTS applications:* Registered COTS application "PlumeAnalyzer" is designed to provide these types of services.

DRAC Service	Registered Applications
Gather Agent Parameters	*Human*
Gather Weather and Env. Parameters	*Human*
Determine Population Distribution	*Human*
Create Dispersion Plume	*PlumeAnalyzer v1.0*
Estimate Casualties	*PlumeAnalyzer v1.0*

Associating an application to respective DRAC services is only part of the registration picture. Later, during system design, the system integrator performs a brokering activity based on “how” an application performs its functions as well as “what” domain tasks the application provides. Thus, the application must also be characterized by specific implementation features and infrastructure requirements.

Taking into account both the DRAC and IRAC registrations, Figure 28 depicts both the "what" and "how" registration for the *FDManager v2.0* and *ResponderLocator v1.0* applications listed above. An application is registered against both DRACs and IRACs. Registering against a DRAC specifies "what" the application does; registering against an IRAC specifies "how" it does it. The IRAC ontology (see Section 4.5) represents domain-independent knowledge about generic infrastructure requirements.

TARETS assists the developer in traversing the IRAC ontology to ensure an application is registered in a manner that describes it as completely as possible.

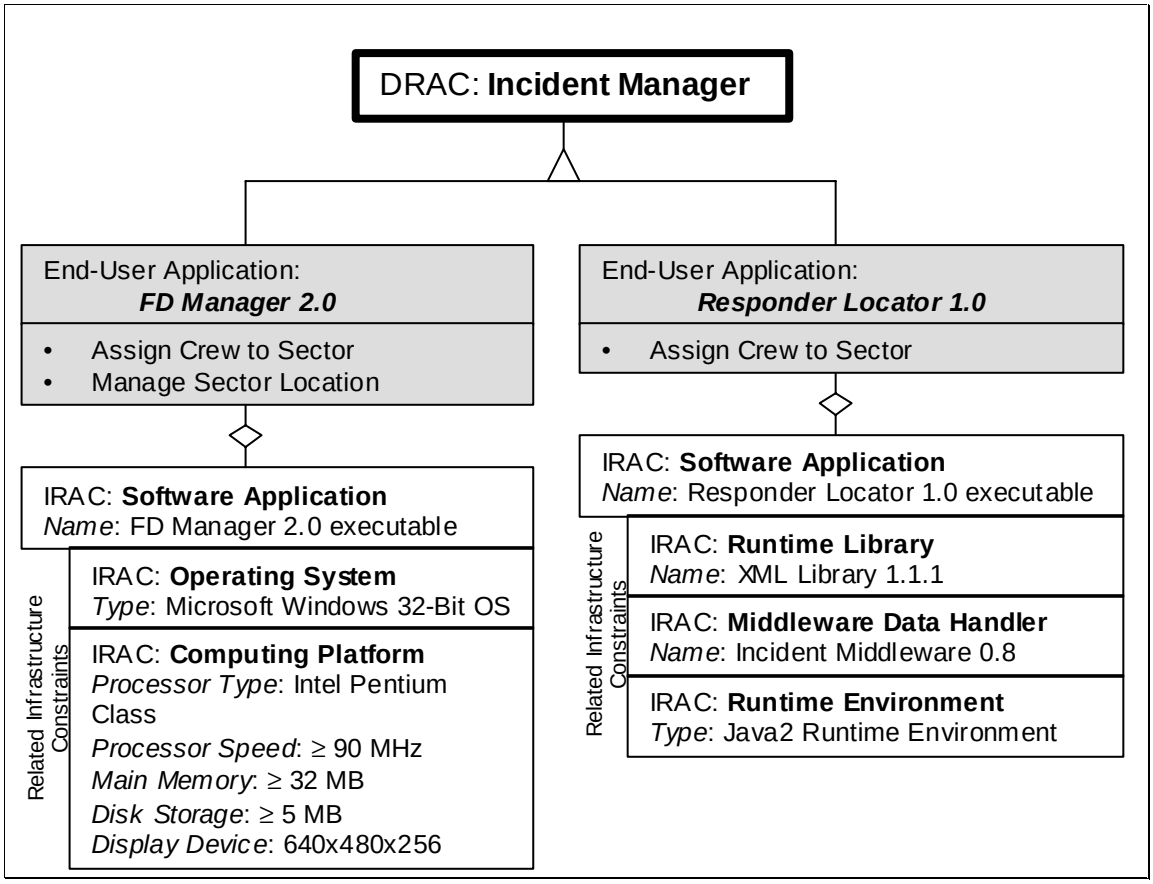


Figure 28: Incident Response Application Registration Example

5.6. System Design

With an understanding of the domain and infrastructure requirements for a particular installation, the system designer performs the activity of system design by selecting specific applications that together solve the client's installation requirements.

As evident from the sample KA session reports above, the incident management application is intended for two very different installations: City X and City Y. To accommodate the requirements of each installation from a single set of registered

applications, SEPA's domain and implementation requirements representations are intended to aid the designer in identifying candidates for reuse.

Beginning with a blank slate, system design starts with a specification of "What domain tasks should be automated at this installation?" SEPA tools help the designer to answer this question independently of implementation concerns by perusing the domain model and/or reference architecture to identify candidate tasks. Browsing the reference architecture is often preferred since services are likely organized into DRACs that correspond to familiar domain roles (e.g. Incident Commander role, HazMat Specialist role).

For City X, the required domain functions based on KA (Figure 16) are:

- Manage Sector Location (synonym for the originally requested "Manage Division Location")
- Create Dispersion Plume
- Estimate Casualties
- Create Incident Spill Report
- Gather Agent Parameters
- Gather Weather and Env. Parameters
- Determine Population Distribution
- Capture Weather Information

Once the set of DRACs that must be addressed are identified, the Requirements Integration and Verification Tool (RIVT) can be used to identify registered applications capable of performing the selected domain tasks.

Based on the domain services selected for City X, candidate applications suggested are:

- Manage Sector Location – *FDManager v2.0* OR *ResponderLocator v1.0*
- Create Dispersion Plume – *PlumeAnalyzer v1.0*
- Estimate Casualties – *PlumeAnalyzer v1.0*
- Create Incident Spill Report – *IncidentReporter v2.0*
- Gather Agent Parameters – *Manual Task*

- Gather Weather and Env. Parameters – *Manual Task*
- Determine Population Distribution – *Manual Task*
- Capture Weather Information – *WeatherMonitor 0.7*

For City X, neither task constraints nor system constraints have been specified. Thus the designer may select either *FDManager v2.0* or *ResponderLocator v1.0* to provide "Manage Sector Location."

The designer narrows the set of candidate applications by imposing implementation requirements. These requirements (constraints) are expressed in the same language as the application registered infrastructure, the IRAC ontology. Having satisfied both domain and implementation requirements for each application, the designer continues the brokering process by attempting to integrate applications to ensure compatibility based on their registration specifications.

Each of the registered applications is associated with one or more IRACs to describe the infrastructure required for each application:

- *FDManager v2.0*
  - Processor Type: Intel
  - Processor Speed: ≥ 90 MHz
  - Disk-space: ≥ 5 MB
  - Memory: ≥ 32 MB
  - Display Size: 640x480x256
  - requires-OS: { Windows NT Workstation v4.0 SP4 | Windows 95 | Windows 98 }

- *ResponderLocator v1.0*
  - requires-OS: { Windows NT Workstation v4.0 SP4 | Windows 95 | Windows 98 }
  - depends-on-application-framework: *IncidentMiddleware v0.8*
  - uses-system-library: *XMLLibrary v1.1.1*
  - executes-in-runtime-environment: *JavaRuntimeEnvironment v1.2*

- *PlumeAnalyzer v1.0*
    - depends-on-related-software: { DSWE Plume Calculator | DITR Plume Calculator }
    - requires-OS: { Windows NT Workstation v4.0 SP4 | Windows 95 | Windows 98 }

- *IncidentReporter v2.0*
    - requires-OS: { Windows NT Workstation v4.0 SP4 | Windows 95 | Windows 98 }
    - depends-on-application-framework: *IncidentMiddleware v0.8*

- *WeatherMonitor 0.7*
    - requires-OS: { Windows NT Workstation v4.0 SP4 | Windows 95 | Windows 98 }
    - executes-in-runtime-environment: *JavaRuntimeEnvironment v1.2*

The integration process also highlights the need for supporting applications, such as databases, word processors, and web servers. Related infrastructure applications may, in turn, require additional applications:

- *IncidentMiddleware v0.8* (required by *IncidentReporter v2.0*)
    - uses-repository *IncidentRepository 2.0*

In contrast to the City X installation, options for the City Y installation are constrained by the requested map-based user interface and the required operating system (Windows NT). Applications used in the City X installation are reused for the City Y installation

based on their registration to required tasks, providing they satisfy the stated infrastructure requirements. These requirements help narrow the selection, resulting in the following application choices.

- Manage Units and Shifts – *FDManager v2.0*
- Assign Crew to Sector – *ResponderLocator v1.0* (task is constrained by requirement for map-based application)
- Manage Sector Location – *ResponderLocator v1.0*
- Replay Incident –IncidentReporter v2.0

In addition to supporting the brokering process, RIVT provides a query facility against the information contained in IRAC, DRAC, and registration representations to support impact and reuse analysis.

5.6.1. Impact analysis for new application development

In a domain as complex as Incident Management, the domain scope modeled will likely need to grow to satisfy a larger customer base. Through KA, new domain tasks are added in context with existing functionality. Relationships to existing tasks are determined during knowledge model merging and filtered down to the RA as changes in data and event exchange between DRAC services.

When new applications are under consideration, an initial analysis of the Reference Architecture and currently registered applications can provide information regarding the degree to which a proposed application will affect or be affected by other applications. Functionality to be provided by the new application is identified in the RA. If the domain task has not yet been represented in the RA, additional KA yields an expanded DM and results in new RA services. Likely interaction among applications is evident through data and event exchange between the DRACs that provide the functionality under consideration. These interactions can identify (i) other applications already registered which may require modification to correctly interface with the new application and (ii) functionality which has not yet been automated but must now be automated if the functionality under consideration is to be included in the system design.

5.6.2. Reuse of existing applications through requirements

The SEPA process and tools enable reuse of requirements through requirements modeled in a computational representation. Implementation and domain requirements represented in SEPA can be used in combination to determine the likelihood of reuse.

As in the brokering examples described in the prior section, the first step in satisfying the needs of an installation site is determining which domain tasks are to be supported. Through application registration against the RA, these domain tasks reference applications that become candidates for reuse. To identify domain functionality of interest, the end user can take a number of approaches in posing questions to the RA.

- For users focused on particular domain data elements that would be affected

  Elements that represent information resources in the DM are "owned" by DRACs in the RA. RIVT can be used to determine what DRAC services utilize those data elements and what applications are registered to those services.

- For users familiar with high-level domain tasks that are to be automated

  High-level tasks in DM identify lower level tasks which are satisfied by DRAC services in the RA. RIVT can be used to determine what applications are registered to those services.

- For users intending to automate the tasks for a particular domain role (performer)

  Tasks performed by a domain role are identified in the DM and those tasks are satisfied by services in the DRACs. Furthermore, DRACs often closely align with performer roles, thus collecting the services associated with a role. RIVT can be used to determine what applications are registered to those services.

Once a selection of candidate applications is identified for reuse, infrastructure requirements associated with registered applications provide additional information for determining level of reuse. Questions that can be answered by the SEPA IRAC representation include:

- What types of resources does the application require (e.g. hard disk, memory, peripherals)?

- Does the application run in the chosen runtime environment (e.g. Java, Visual Basic for Applications)?

- Does the application run under the chosen operating system?

- Will an application conflict with other candidate applications if integrated in a single installation (e.g. the sum of memory required by two applications exceeds that offered by the destination server)?

- Consider the pool of available application developers, is the application developed using skills developers possess and technologies they are familiar with (e.g. Java, Visual Basic)?

Figure 29 depicts a selected query screen in RIVT designed to answer these types of questions. This screen provides the developer, end-user, or integrator with a high-level picture of the relationships between RA services (domain tasks), DRACs, registered applications, and installation sites. In the example shown, the service "Manage Sector Location" has been selected. The DRAC offering this service is highlighted, *Incident Manager*. In the third column, applications registered to the Incident Manager are highlighted, *FDManager v2.0* and *ResponderLocator v1.0*. The installation sites requiring this service are highlighted, including both City X and City Y.

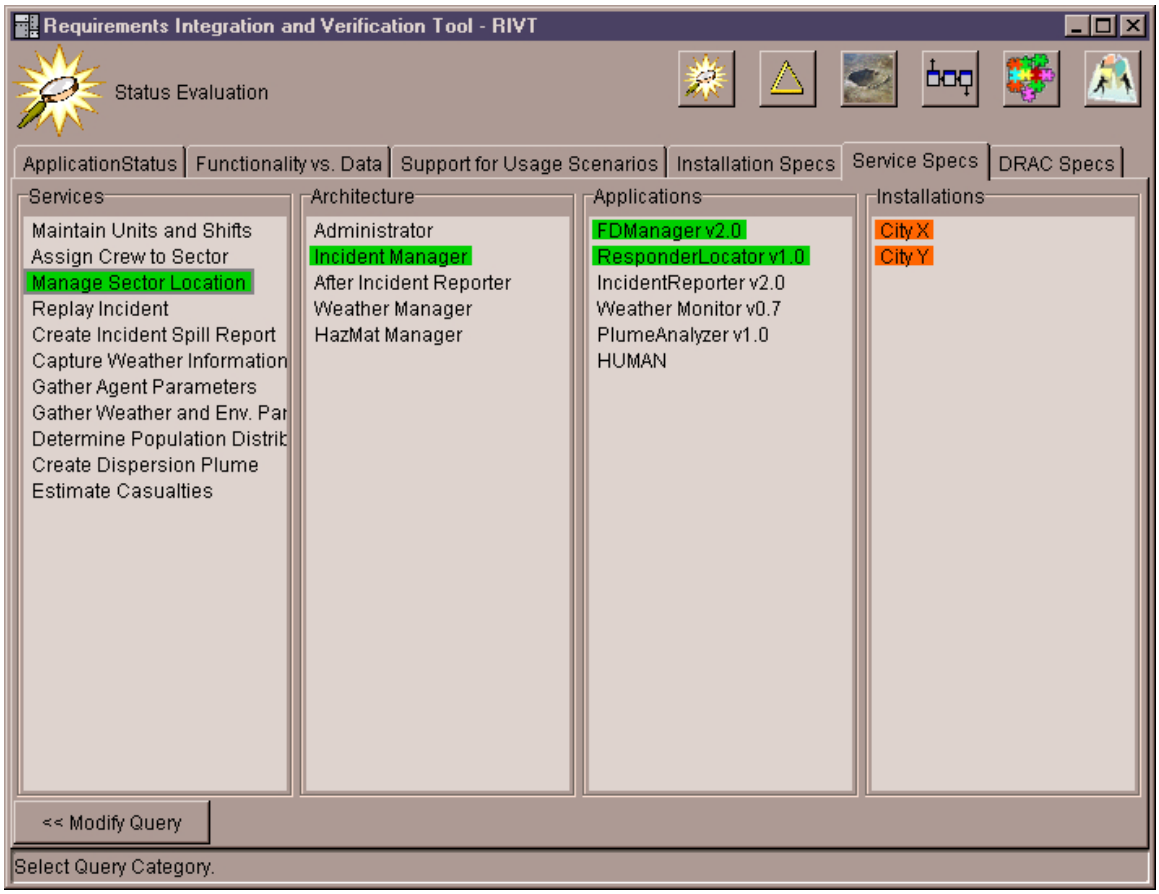


Figure 29: Example RIVT Query Interface for Requirements Reuse Analysis

5.7. SEPA Contributions to Requirements Evolution and Reuse

This section highlights selected SEPA contributions to requirements evolution and reuse attributable to its emphasis on requirements analysis prior to design, separation of domain and application requirements, and requirements analysis for component-based development.

- ***Traceability:*** SEPA's emphasis on traceability throughout the requirements analysis and refinement process allows SEPA to support queries against any artifacts in the process (e.g. KMs, DM, RA, registered application) and relate those artifacts to other artifacts in the process. For example, the domain expert may be more familiar with high-level domain tasks than with specific application functions. A query to identify applications supporting specific domain tasks would best be posed to the domain model. These tasks, in turn, are traceable to DRAC services that are satisfied by registered applications.

- ***Support for Multiple Viewpoints:*** Derivation of the Reference Architecture and all subsequent system development depend on a single, unified representation of the domain. On the other hand, experts representing individual domain viewpoints want to be assured either that their interests are being captured in the resulting model or that there is adequate justification for why their interests are not being represented. HyDRA captures the original results from each KA session in a separate model space (a collection of knowledge models). As artifacts from KA sessions are synthesized into a unified model, traceability is retained and the rationale for each KE decision during synthesis is recorded. One benefit afforded by this approach is that even after the unified model is created, an expert can continue to browse domain tasks in their own terminology and referencing unified tasks via traceability links. Furthermore, the HyDRA synthesis process follows the viewpoint hierarchy, *first* unifying requirements from experts holding the *same* viewpoint. This retains separate model spaces representing the functional and data requirements associated with each viewpoint.

- ***Computational Requirements Representation:*** Typical requirements management approaches rely on tracing text fragments to design and development artifacts (Chipware, 1999; QSS, 1998; Technologies, 1999). This flat "bucket of thou shalt" representation supports only text string searches and provides little ability for structured requirements types and complex relationships among various requirements types (Palmer, 1997). The end result is a greater difficulty associating applications features to originally stated requirements. For example, suppose a text requirement stated "The application supporting the billing process shall run on a Windows NT machine with no more than 128MB memory." A flat representation leads to some difficulty:

  - This statement actually suggests two different requirements: "Windows NT" and "less than 128 MB."

  - To determine if "128 MB memory" is satisfied requires a numeric representation for comparison.

  - Both these requirements act as constraints on the "billing process" task. Any application registered to the "billing process" task should satisfy both the

"Windows NT" and "less than 128 MB" requirements.

- ***Separation between Domain and Infrastructure Requirements:*** To improve chances for reuse, software methodologies often suggest that "what" a system must do should be modeled independently of "how" the system should be implemented. The presumption is that "what" is done in a domain changes far less often than "how" it is accomplished, especially given frequent changes in technology (Tracz, 1991; Tracz, 1993; Tracz, 1995). Despite this advice, other requirements management tools do not explicitly recognize this separation.

- ***Comprehensive Process Support for Requirements Refinement and Management:*** Numerous requirements management tools are available for requirements analysis and refinement, taking different approaches to requirements representation (Chipware, 1999; QSS, 1998; Technologies, 1999). An even greater number of tools support object-oriented development after requirements have been elicited, validated, and merged (SES, 1999; Verilog, 1997). Selected tool suites attempt to bridge requirements identification with subsequent object-oriented analysis and development (Rational, 1998a). SEPA provides an integrated suite that connects all artifacts throughout requirements evolution via a common traceability representation. Without this integration, the development process cannot be seamless, and the transition from requirements capture, to requirements synthesis, to class derivation, to system design (often performed by different people) becomes excessively difficult. Among the consequences of this “break” in the process, some requirements may be ignored while others may not be maintained in the long-term after implementation or as the domain scope broadens.

- ***Infrastructure Requirements Represented in a Domain-Independent Ontology:*** The separation of domain and infrastructure requirements is key to SEPA's approach to maximum reuse. Further encouraging long-term reuse is SEPA's domain-independent IRAC ontology for modeling infrastructure requirements. While there are many approaches for characterizing infrastructure requirements, the IRAC ontology provides consistency between projects and beyond initial implementation. Over time, additional applications may be introduced that can be registered to existing domain services in the RA. These new applications are registered utilizing the same IRAC ontology as the originally registered applications, thus providing a consistent language for comparing application features. The consistent representation also allows applications to be selected based on a common feature, independent of domain functions (e.g. find all applications that run on Windows NT).

As illustrated in the City Y installation example, the separate application requirements representation simplifies the application selection process. Registered applications can be selected based on their ability to perform domain tasks independently of their specific implementation features or required infrastructure. Often multiple applications are registered as being capable of providing the same DRAC service. Specific application features or resource requirements can be used to narrow the set and the common representation allows applications to be compared in a trade-off analysis.

- ***Close Alignment of RA DRACs to COTS Applications:*** SEPA's RARE tool guides the derivation of DRACs from DM functional and data information based on a set of goals prioritized by the architect. Among the goals significant to COTS reuse is the emphasis on grouping domain tasks into DRACs based on close alignment with those services existing COTS applications provide. This goal improves the likelihood a COTS application will be registered completely to a DRAC (all services supported) and increases opportunities to select COTS applications in whole based on desired DRAC functionality.

6. Conclusions

The Systems Engineering Processing Activities (SEPA) being developed at the University of Texas at Austin in the Laboratory for Intelligent Processes and Systems (LIPS) seeks to improve the systems engineering process by providing a comprehensive development methodology and a suite of supporting tools. SEPA focuses on support for Component-based Software Engineering (CBSE). *While available support for implementation of component-based systems is strong, SEPA emphasizes the early requirements gathering and analysis activities demanded by the CBSE process.*

Distinguishing SEPA features include emphasis on the following.

1. *Support for requirements analysis prior to design.* Recognizing that many modeling methodologies do not adequately support early analysis efforts, SEPA emphasizes earlier activities to provide a sound foundation for component derivation. Furthermore, attention paid to these early analysis activities facilitates maintaining traceability and verification of deliverables. SEPA supports the incremental gathering of requirements and the multiple perspectives acquired from different domain experts typically found in large development efforts.

2. *Separation of domain and application requirements.* Domain-based requirements focus on "what" a component does while application requirements emphasize "how" a component must perform for a particular system implementation. The importance of this distinction is that "what" a component does in a domain is often less dynamic over time than "how" the component does since technologies change over time and the "how" may be strongly dependent on the particular technology solution selected.

3. *Support for requirements analysis for component-based development.* The analysis of domain requirements in SEPA yields a Reference Architecture (RA) of domain-based components responsible for services required to support domain tasks. In addition to declarative and behavioral information, the SEPA domain-based RA representation includes integration rules describing constraints and dependencies between components. The RA produced by the SEPA process represents domain requirements and is independent of specific implementations. This allows it to be reused in a "family" of applications in the domain.

To guide the developer in applying the unique features of the SEPA methodology, a suite of tools is being developed to support each phase in the SEPA process:

- *Knowledge Acquisition Manager:* Provides project management and document management functions to support the Knowledge Acquisition (KA) process.

- *Hybrid Domain Representation Archive:* Aids in representing and synthesizing the information from Knowledge Acquisition reports into a single, functional Unified Knowledge Model. The synthesis process merges domain information from multiple experts while preserving traceability to KA.

- *SEPArator:* Separates the information present in the SEPA Unified Knowledge Model into a Domain Model containing domain-based requirements, a Technology Solutions Repository capturing legacy systems, and an Application Requirements Model containing requirements for specific system implementations.

- *Reference Architecture Representation Environment (RARE):* Guides the developer in transitioning from a functional Domain Model to an object-oriented Reference Architecture of domain-based components. As a domain-based model specifying "what" components must do, the RA remains relatively stable over time and provides a template for a "family" of application systems.

- *Tool for Application Requirements Extraction and Technology Specification (TARETS):* Complements RARE by modeling application requirements and technology solutions gathered from KA and represented in the Application Requirements Model. Models are linked to Reference Architecture components for subsequent use in generating a system design specification satisfying domain and application requirements.

- *Requirements Integration and Verification Tool (RIVT):* Aids in the system design function by combining application requirements and technology solutions modeled in TARETS with domain requirements represented as components in the Reference Architecture to satisfy all requirements for a given application and provide a rationale for a given system design configuration.

7. References

Alford, M., & Lawson, J. (1979). *Software Requirements Engineering Methodology (Development)* (RADC-TR-79-168): U.S. Air Force Rome Air Development Center.

Alonso, F., Juristo, N., Mate, J. L., Pazos, J. (1996). Software Engineering and Knowledge Engineering: Towards a Common Life Cycle. *Journal of Systems and Software, 33*, 65-79.

Barber, K. S., & Jernigan, S. R. (1999, June 28-July 1). *Changes in the model creation process to ensure traceability and reuse.* Paper presented at the International Conference on Artificial Intelligence, Las Vegas, Nevada.

Brown, A. W. (1996). Foundations for Component-Based Software Engineering. In A. W. Brown (Ed.), *Component-based Software Engineering* (pp. vii-x). Los Alamitos, California: IEEE Computer Society Press.

Caldieri, Gianluigi, & Basili, V. R. (1991, February 1991). Identifying and Qualifying Reusable Software Components. *IEEE*.

Chipware, I. (1999). *icCONCEPT RTM- Requirements & Traceability Management Tool*, [Web site]. Marconi Systems Technology, Inc. Available: http://www.mstus.com/.

Christel, M. G., & Kang, K. C. (1992). *Issues in Requirements Elicitation* (Technical CMU/SEI-92-TR-12). Pittsburg, Pennsylvania: Carnegie Mellon University.

Clements, P. C. (1996). From Subroutines to Subsystems: Component based Software Engineering. In A. W. Brown (Ed.), *Component based Software Engineering* (pp. 3-6): IEEE Computer Society Press.

Clements, P. C., & Northrop, L. N. (1996). Software Architecture: An Executive Overview. In A. W. Brown (Ed.), *Component-based Software Engineering* (pp. 55-68). Los Alimitos, California: IEEE Computer Society Press.

Cybulski, J. L. (1995). Reusing Software Specifications by Analyzing Informal Requirements Texts. .

Department of Defense, U. S. (1996). *Guidelines for Successful Acquisition and Management of Software-Intensive Systems* : Department of the Air Force, Software Technology Support Center.

Finkelstein, A. C. W., Gabbay, D., Hunter, A., Kramer, J., & Nuseibeh, B. (1994). Inconsistency Handling in Multiperspective Specifications. *IEEE Transactions on Software Engineering, 20*(8), 569-577.

Garlan, D., Allen, R., & Ockerbloom, J. (1995, April 1995). *Architectural Mismatch or, Why it's hard to build systems out of existing parts.* Paper presented at the 17th International Conference on Software Engineering, Seattle, Washington.

Glass, R. L. (1996). Methodologies: Bend to Fit? *Journal of Systems and Software, 35*, 93-94.

Gomma, H. (1995). Reusable Software Requirements and Architectures for Families of Systems. *Journal of Systems and Software, 28*, 189-202.

Grady, R. (1992). *Practical Software Metrics for Projects Management and Process Improvement*. Englewood Cliffs: Prentice-Hall.

Graham, I. (1995). *Migrating to object technology*. Wokingham, England ; Reading, Mass.: Addison-Wesley Pub. Co.

Graser, T. J. (1996). *Reference Architecture Representation Environment (RARE) - A Reference Architecture Archive Promoting Component Reuse and Model Interaction.* Unpublished Masters, The University of Texas at Austin, Austin.

Harbison, K. (1997). Scenario-based Engineering Process . http://caesar.uta.edu/caesar/process.html: Center for Advanced Egineering Systems and Automated Research, The University of Texas at Arlington.

Hardy, C. J., Barrie, T. J., Edwards, H. M. (1995). The Use, Limitations and Customization of Structured Systems Development Methods in the United Kingdom. *Information and Software Technology*(September).

Harel, D. (1990). STATEMATE: A working environment for the development of Complex Reactive Systems. In T. DeMarco & T. Lister (Eds.), *Software State-of-the-Art: Selected Papers* (pp. 322-338). New York: Dorset House.

Heitmeyer, C., Kirby, J., Jr., & Labaw, B. (1997). *Tools for formal specification, verification, and validation of requirements.* Paper presented at the 12th Annual Conference on Computer Assurance (COMPASS '97), Gaithersburg, MD.

Kotonya, G., & Sommerville, I. (1997). Requirements engineering with viewpoints. In R. H. Thayer & M. Dorfman (Eds.), *Software Requirements Engineering* (pp. 150-163). Los Alamitos, CA: IEEE Computer Society Press.

Kramer, J., Ng, K., Potts, C., & Whitehead, K. (1988). Tool Support for Requirements Analysis. *Software Engineering Journal, 3*(3), 86-96.

Lam, W. (1997). Achieving Requirements Reuse: A Domain Specific Approach for Avionics. *Journal of Systems and Software, 38*, 197-209.

Leite, J. C. S. d. P., & Freeman, P. A. (1991). Requirements Validation Through Viewpoint Resolution. *IEEE Transactions on Software Engineering, 17*(12), 1253-1269.

McGraw, K., & Harbison, K. (1997). *User-centered Requirements*. Mahwah: Lawrence Erlbaum Associates, Publishers.

Oskarrson, O., Glass, R. L. (1996). *An ISO 9000 Approach to Building Quality Software*. New Jersey: Prentice-Hall.

Palmer, J. D. (1997). Traceability. In R. H. Thayer & M. Dorfman (Eds.), *Software Requirements Engineering* (pp. 364-374). Los Alamitos, CA: IEEE Computer Society Press.

Perito-Diaz, R. (1990). Domain Analysis: An Introduction. *ACM SIGSOFT Software Engineering Notes, 15*(2), 47-54.

Pohl, K. (1996, April 15-18, 1996). *PRO-ART: Enabling Requirements Pre-Traceability.* Paper presented at the Second International Conference on Requirements Engineering, Colorado Springs, Colorado.

QSS. (1998). DOORS 4.0 . http://www.qss.co.uk/DOORS4/: Quality Systems and Software.

Rational. (1998a). Software Development, Component-Based, Programming Tools: Rational . http://www.rational.com/: Rational.

Rational. (1998b). Unified Modeling Language . http://www.rational.com/uml: Rational Software Corporation.

Riel, A. J. (1997). *Object-Oriented Design Heuristics*. Reading, MA: Addison-Wesley.

Rolland, C. (1994, April 18-22). *Modeling the Evolution of Artifacts.* Paper presented at the The First International Conference on Requirements Engineering, Colorado Springs, Colorado.

Schach, S. R. (1993). *Software Engineering*. (2nd ed.). Boston: Aksen Associates Incorporated Publishers.

SES. (1999). SES/objectbench: object-oriented analysis, simulation and code-generation. .

Sommerville, I., & Sawyer, P. (1997). *Requirements engineering : a good practice guide*. New York: Wiley.

Sommerville, I., Sawyer, P., & Viller, S. (1998, April 6-10, 1998). *Viewpoints for requirements elicitation: a practical approach.* Paper presented at the Third International Conference on Requirements Engineering, Colorado Springs, Colorado.

Technologies, T. (1999). *Slate REquire*. Available: http://www.tdtech.com/.

Telelogics. (1997). ATM via SDL . www.telelogic.com: Telelogic.

Tracz, W. (1991, Novenber 18-20, 1991). *An Outline for a Domain-Specific Software Architecture Engineering Process.* Paper presented at the Fourth Annual Workshop on Software Reuse, Reston, VA.

Tracz, W., Coglianese, L., Young, P. (1993). A Domain Specific Software Engineering Process Outline. *ACM SIGSOFT Software Engineering Notes, 18*(2), 40-49.

Tracz, W. (1995). DSSA (Domain Specific Software Architecture) Pedagogical Example. *ACM SIGSOFT Software Engineering Notes, 20*(3), 49-62.

Tracz, W. (1996). *Domain-Specific Software Architectures, Frequently Asked Questions* : Loral Federal Systems Company.

Verilog. (1997). Verilog ObjectGEODE . http://www.verilogusa.com/home.htm.

Vlasbom, G., Rijsenbriji, D., & Glastra, M. (1995). Flexibilization of the Methodology of Systems Development. *Information and Software Technology*(November).

Whitmire, S. A. (1997). *Object-Oriented Design Measurement*. New York, NY: John Wiley & Sons, Inc.