SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; **00**:1–2 *Prepared using* speauth.cls *[Version: 2002/09/23 v2.2]*

The composition paradox in software architecture

Christoph Stoermer*[1], Chris Verhoef[2]

[1] *Robert Bosch Corporation, Research and Technology Center, USA*
[2] *Free University of Amsterdam, NL*

SUMMARY

One of the most prominent activities in software architecture design is the partitioning of systems into parts, resulting in architectures decomposed into elements and their relations. This decomposition is important to manage increasingly complex systems. However, complex designs largely driven by decomposition lead to elements with many complex and varied interconnections. The resulting highly coupled structures are harmful for systems that have to be composed from parts with isolated interfaces. Decomposition and composition are complementary activities in software architecture design and equally important to balance cohesion and coupling. Consequently, explicit focus is needed to compose elements into a working system as well as to decompose systems into smaller elements and relations to manage their complexity. Software architectures only designed with decomposition in mind lead to monolithic systems and thus achieve the opposite of what was originally intended. We named this nonintuitive side-effect of realizing systems *the composition paradox*. Addressing this composition paradox will lead software architects to decisions that will strike a better balance between partitioning and integrating both monolithic and componentized structures. We explain the composition paradox along the design and assembly of a security system.

KEY WORDS: asset constraint, composition, composition paradox, decomposition, prediction, quality attributes, software architecture, views

1. Introduction

Who needs software architecture? is a prominent question that sometimes comes across in organizations. The question was discussed in a column by Dave McComb [22]. His conclusion is that software architectures are only needed

*Correspondence to: Robert Bosch Corporation, Research and Technology Center, 2 NorthShore Center #320, Pittsburgh, PA 15212, USA
Contract/grant sponsor: This research has been partially sponsored by the Robert Bosch Corporation and the Dutch Ministry of Economic Affairs via contract SENTER-TSIT3018 *CALCE: Computer-aided Life Cycle Enabling of Software Assets*

where there are relatively complex systems, where the complexity is interfering with productivity or the ability to change and respond, or where major changes to the infrastructure are being contemplated, that companies should really consider undertaking architectural projects.

So, software architectures are necessary for complex systems to ensure return of investment. Along the same line as this argument one of the earlier assertions by Shaw and Garlan assign complexity a prominent role in the motivation for software architecture [26]:

As the size and complexity of software systems increases, the design and specification of overall system structure or software architecture emerges as a central concern.

Today, we know that structure is foremost driven by quality attributes [1], such as variability in product lines sold in worldwide markets [2], safety in fly-by-wire systems [17], or fault-tolerance in embedded systems [13]. McComb's conclusion, as cited above, also addresses the significance of quality attributes. He refers to the ability of complex systems to respond to changes. Because quality attribute requirements drive the software architecture design, they are the source for many early design decisions. In an IEEE Software column, Martin Fowler wonders why people would feel the need to get some things right early in the project [12]. This is Fowler's answer including a definition of software architecture (that was independently also proposed in [16])

The answer, of course, is because they perceive those things as hard to change. So you might end up defining architecture as things that people perceive as hard to change.

Although Fowlers proposal does not provide a recipe to design a software architecture and also provides only one aspect among many other architecture definitions that are documented, for example, in [25], it highlights the importance of early design decisions that will be hard to change during later development steps. Even though we focus in this paper on software architectures, his assertion is valid for other architectures as well, including system, hardware and civil architectures.

Design decisions are foremost driven by quality attribute concerns. Their sources are primarily found in the business practices of an organization. This includes a variety of technical and organizational practices, such as marketing, product planning, technology scouting, service, and maintenance. Software architects are responsible for creating software architectures that comply with these business requirements. The resulting architecture solution implies human creativity to build and validate that the solution is realized within time and budget as illustrated in Figure 1. The human input depicted in Figure 1 expresses the ingenious activity to create and validate a software architecture that incorporates design elements to facilitate change where expected, to fix structure where needed, and to be flexible where suspected.

The resulting architecture of the creation process is mainly driven by the necessity to decompose a system into parts to divide and conquer complexity. We argue in this paper for a complementary emphasis on the composition of a system from its parts and the validation of that composition in order to address integration, distribution, and customization needs in today's networked and componentized systems. A complementary view will allow for better software architectures with decisions that on the one hand balance cohesion and coupling, and on the other hand balance decomposition and composition. Potential conflicts of decomposition

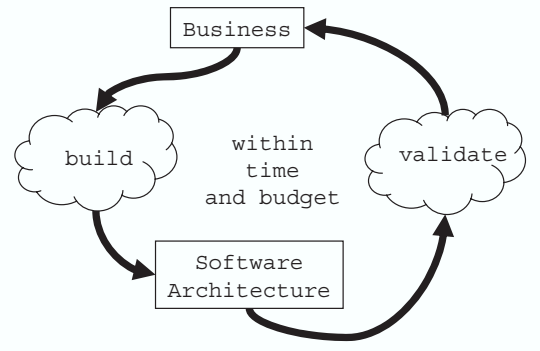

Figure 1. Human input for building and validation of software architectures.

and composition views have to be resolved in additional views, that link the composition and decomposition elements in order to preserve the conceptual integrity of the architecture [3].

Composition also affects the verification process that investigates the architectural decisions captured in the software architecture. The arguments in this paper emphasize the validation of system composition from many parts equally with the validation of the decomposition of a system into smaller structures. A validation of the break-down structures that only takes a system decomposition into account will miss the significance of the composition validation when systems are installed in many customer configurations that are often unforeseeable.

Relating decomposition and composition will broaden the usage of quality attribute models. For example, a performance model affects the decomposition by carefully separating time-critical from non-time critical paths. A schedulability analysis [20] will then verify the decomposition model, which often results in design improvements. The composition perspective uses the performance model in a similar way by verifying the system performance in a distributed component network. The system installer or configuration service will use the model to verify or predict a plausible system configuration.

Tying decomposition and composition will result in:

(1) Integration of component models in architectures. Component models provide rules, for example, about component lifecycles and interactions. They provide constraints for further decomposition to ensure composition and deployment.
(2) Common quality attribute models. Quality attribute models provide a framework to reason about design decisions and alternatives [1]. They also predict qualities of systems composed of components.
(3) Early software architecture evaluation processes that will also focus on composition. Evaluations have to consider if systems are correctly decomposed into parts as well as whether they can be correctly composed from their parts.

Frequently, composition becomes a concern after the delivery of initial products. Systems that do not address composition factors in their software architecture designs because of time-to-market pressure for initial customers or markets may have problems with future deployments. Although initial customer releases are successful, the development organization has to invest in expensive modifications for follow-up deliverables and to satisfy return-on-investment goals. The bottom line is that shortcuts in the software architecture design will most certainly imply expensive modifications. Therefore, complexity is primarily not a matter of size, but rather the provision of a solution in time and budget for systems that are used in many contexts. Both, systems and contexts, evolve over time, which has to be facilitated by the underlying architecture.

The example system presented in this paper was investigated during a case study in the embedded security systems domain. This domain is a fertile ground to show that composition will affect architectural choices significantly. Systems originally designed for stand-alone products have to be integrated into larger networks in residential and commercial markets. Technical changes are typically constrained by time and budget limitations. The example system is used in this paper to demonstrate the importance of composition in the software architecture design and evaluation process. Other architecture concerns, such as the communication architecture and the management of time critical events from different sources, are not addressed in this paper to keep the focus on the composition paradox. Of course, those other aspects of design should also be done, but these are more established than the aspect of this paper: putting composition explicitly on the agenda. Our example is particularly suited since:

(1) Security systems are widely used
(2) The composition after production of the parts is a prominent concern
(3) Large security systems are rather complex, such as an integrated airport security system or a security system in a petrochemical process factory.

The reason for our participation was initially triggered due to the fact that the organization required an evaluation of the fitness of an existing fire panel software architecture for a new combined panel with fire, intrusion, and access functionality. We performed the evaluation based on use case maps [4]. The abstraction level of use case maps turned out to be very useful to increase software architecture understanding, to clarify terminology and domain concepts, and to expose limitations and opportunities of the existing architecture.

The remainder of the paper is organized as follows. Section 2 introduces the example system including an overview of the embedded security system domain. The primary decomposition of a security control panel is outlined in Section 3. Section 4 introduces an impact evaluation process due to constraints imposed by the organization. Section 5 introduces the composition paradox. The different foci of decomposition and composition are discussed in Section 6. Section 7 describes a concrete example configuration and discusses the role of the component model in the example system. In Section 8 we provide an overview of the related work. Finally, we summarize our lessons learned in Section 9.

2. Example System

Building security systems comprise fire, access, and intrusion systems. These systems typically consist of sensor networks, control panels, and management as illustrated in Figure 2:

- Sensor networks can contain a wide variety of different sensor and actuator types, such as ultrasound glass-breaking sensors, optical smoke detectors, heat detectors, CO, CO_2, CH_4, and NH_3 gas detectors, infrared and radar motion detectors, water detectors, door contacts and magnets, sounders, video camera arrays, and automatic call units. An industry trend is to move away from today's wired networks towards wireless systems with significant reductions of cable installation and maintenance costs. To compose larger networks, modern wireless protocols support multi-hop communication, where sensor information can be transmitted via parent/child relations [29]. A further reason for wireless security networks is monumental buildings, museums, castles, churches, and cathedrals where wiring is often not possible.
- Gateways. Sensor networks can be partitioned into sensor clusters with a gateway as the cluster head. A mid-size airport can have a sensor network with 20,000 smoke detectors, thus requiring partitioning of the sensor network into clusters to manage its operation. Modern Gateways route information in wired security networks, such as the Local Security Network (LSN) [21], and wireless sensor networks with low-power consumption protocols, such as ZigBee[†] [5]. Gateways also accommodate for sensor network limitations, such as buffering of information to support low-power consumption protocols with limited Radio Frequency (RF) time.
- Control panels are embedded systems that receive and process signals from sensor networks. They determine whether a signal results in an alarm or other activity. Control panels can also contain sensor groupings, for example an area with sensors controlling a part of an office block. Control panels can be networked either to compose a larger system or to achieve other goals such as redundancy.
- Management systems provide functionality to configure control panels with a customer configuration, or provide video surveillance interfaces for security guards. Management systems are typically PC based with an off-the-shelf hardware and software infrastructure.

Security systems are deployed in residential markets with only a few sensors at the low end as well as in commercial markets with many thousands of sensors at the high end. In most markets installation and service is done via contractors that didn't develop or manufacture the equipment. They have to compose security systems from sensor networks, gateways, and control panels depending on the customer's desired degree of protection, building topology and materials, existing legacy installations, processes, and cost constraints.

Security systems are regulated in many markets. For example, fire control panels in Europe are connected to networks with fire sensors and actuators only and are never sold

[†]The ZigBee alliance is an association of companies working together to enable reliable, cost-effective, low-power, wirelessly networked, monitoring and control products based on an open global standard.

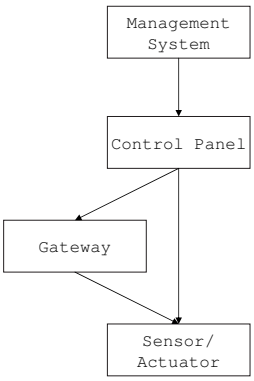

Figure 2. Associations between Security Network Elements.

as combinations with intrusion and access functionality. This restriction does not exist in the American and Asian markets, where combinations of fire with access and intrusion functionality are possible.

The case study was performed largely on the architecture design of a Control Panel, which contains the control software to support a variety of security configurations assembled from components. The software on a Control Panel currently ranges from approx. 120K lines of code (LOC) for smaller systems to 550K LOC for larger systems. A Control Panel also captures most security system concerns of an organization. Some of the technical and business drivers are:

- Combination panels. Panels can be fire, access, intrusion panels, and their combinations.
- Scalability. Systems can consist of thousands of sensors and actuators with high-end and low-end computing platforms in residential and commercial markets.
- Low installation and service cost. Easy installation and service for equipment that requires local presence of expensive personnel.
- Reduction of panel development cost. The organization expects significant development cost reduction due to the design of combination panels. This approach should replace the traditional development of individual fire, access, and intrusion panels.

Over the course of the case study we had a consulting role with particular emphasis on the following objectives:

(1) Moderate the architecture evaluation and design process
(2) Introduce useful best practices in architecture design and evaluation
(3) Explore component configurations for different customers
(4) Provide recommendations to the technical management

Figure 3. Decomposition driven by combination panels.

Our involvement in the case study was initially triggered due to the fact that the organization required an evaluation of the fitness of an existing fire panel architecture for a new combined panel that should comprise fire, intrusion, and access functionality.

3. Decomposition

Architecture decomposition is the activity of partitioning a system into subparts. This divide-and-conquer activity is important to identify technical and organizational break-down structures, not only in large systems but also in smaller systems. Sources for the decomposition are requirements documents as well as—often undocumented—business, organizational, and technical contexts. In particular primary decompositions relate the major software architecture elements and their relations to drivers that express technical and business goals that organizations want to achieve with a system. Resulting decomposition structures are foremost used to communicate the broad picture of the architecture to stakeholders.

In the following, we describe a series of decomposition steps that relate the technical and business drivers from the previous section to their impact on the primary decomposition. The result of each step is illustrated with a figure.

3.1. Combination panels

This driver requires the identification of fire, access, and intrusion elements in the system and either excludes or includes them for a particular installation. In addition, the elements can represent feature sets depending on the market and customer. We depicted the decomposition in Figure 3, a box identifies an architectural element and the text inside the box stands for the element's name.

3.2. Scalability

Panel hardware exists in different hardware versions with low-end and high-end computing platforms, and panel-specific hardware equipment for fire, access, and intrusion. The major driver to introduce the Infrastructure element, depicted in Figure 4, is portability of the application across panel versions and operating systems. The Infrastructure also provides portability to software development platforms for test purposes.

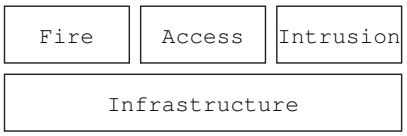

Figure 4. Decomposition driven by scalability.

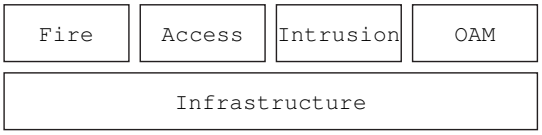

Figure 5. Decomposition driven by low installation and service.

3.3. Low installation and service

This driver is supported by the introduction of an OAM (Operation Administration Maintenance) element as illustrated in Figure 5. The OAM element provides, for example, a uniform interface to Fire, Access, and Intrusion elements or allows exchange of protocols depending on the company that installs, services, or operates the system.

3.4. Reduction of panel development cost

This driver led the organization to establish a reuse effort. Reuse efforts identify commonality and variability in fire, access, and intrusion functionality. The Common element in Figure 6 could include

- the logic to determine if a sensor signal results in an alarm;
- publish/subscribe mechanisms to inform Access about a fire alarm and consequently unlocking doors of occupied rooms;

At this point, the architecture addresses the primary concerns articulated in the technical and business drivers from the previous section. The decomposition provides a coherent set of architectural elements that are traceable to important architectural drivers. We will refer back to the elements identified in the decomposition in the discussion about composition (Sections 5–7).

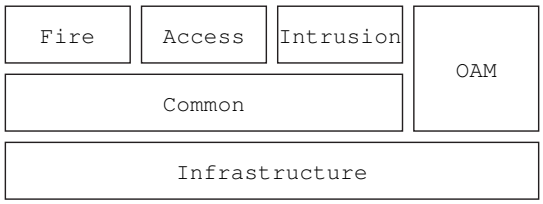

Figure 6. Decomposition driven by reduction of panel development cost.

4. Impact Evaluation

Many new system developments are constrained in their degree of design freedom. Sources for constraints are technical or organizational in nature and have to be carefully weighed with respect to the objectives that the system has to fulfill.

Constraints can impact architectural decisions and processes significantly. In our case study, the management of the organization favored an existing product on which the architecture of the new system should be founded. One of the reasons for this selection was the amount of effort already invested for the design and development of the existing product. This so-called *asset constraint* exists in many organizations to prevent the destruction of knowledge and capital investments.

A pragmatic solution to tackle an asset constraint is to perform an impact evaluation on the existing control panel architecture. In our case study this implied that a key question to be answered was whether the existing architecture could be used as a basis for the new system. The evaluation process, that we jointly designed with the developers of the organization, contained the following steps:

(1) Select representative features from the requirements document of the newly envisioned system.
(2) Create concrete use case scenarios from the selected features.
(3) Map the scenarios onto the high-level design elements of the architecture of the existing system.
(4) Repeat the scenario mapping from Step 3 on a more detailed decomposition level.
(5) Analyze the mapping to come to a conclusion about whether the existing architecture is a solid base for the newly envisioned system.

4.1. Features and Scenarios

Steps 1 and 2 were performed by analyzing the requirements document for the new system. The feature selection was based on architecturally important features and storyboards [18] demonstrating the usage of features in a representative context. Storyboards, also referred to as user interface flow diagrams, describe the user experience in the interaction with the system.

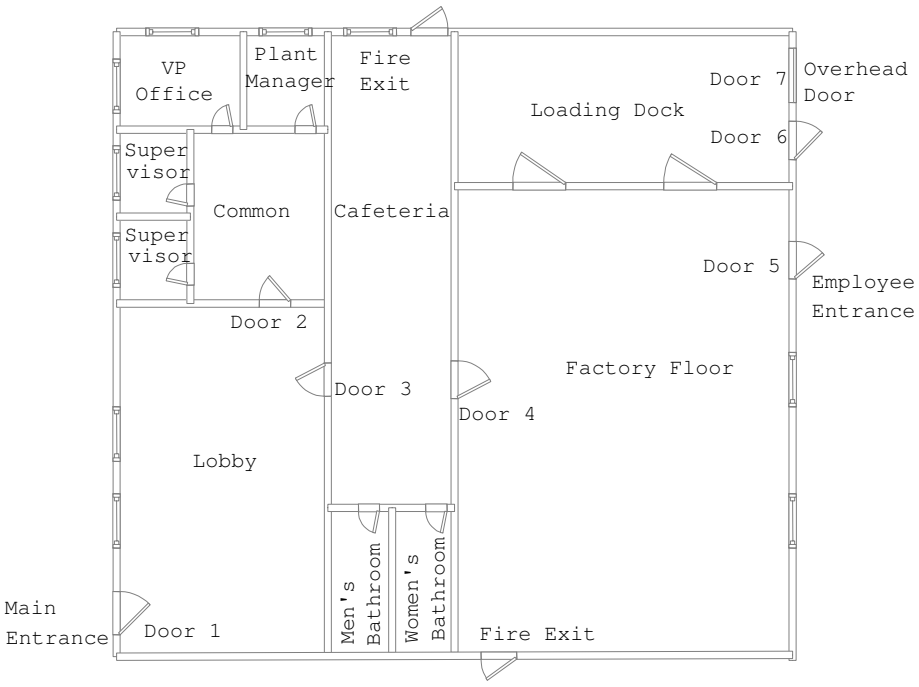

Figure 7. Building floor plan for the production plant storyboard.

In this case, they provide example locations where the security system will be deployed and users will interact with.

One of the stories comprised a small production plant producing goods. A simplified floor plan is illustrated in Figure 7. The story contains the following elements:

- A building with a Front Office, Lobby and Plant Management, Cafeteria with restrooms, Factory Floor and Shipping.
- Employees, such as plant management, receptionists, factory workers.
- Rules, such as 'Lobby is open during normal business hours'; 'Plant Management offices are only open if a member of the management team is in the plant'.
- Event-flows, capturing typical events throughout a work day, such as 'Receptionist enters the lobby at 7.50am'.

The storyboards turned out to be very useful because they combined fire, access, and intrusion functionality. Additionally, by discussing events in the floor-plan of Figure 7, the participants developed an understanding that an intrusion system is a process-oriented system. For example, activities are permitted depending on particular users and calendar/time events.

The floor plan illustrated the end-user perspective on the system and fostered domain understanding in a concrete application setting.

4.2. Scenario Mapping

Steps 3 and 4 were performed by utilizing use case maps [4]. Use case maps provide casual paths - flow of events - cutting across structures, such as systems, subsystems, and components. The use case maps supported the elicitation from the developers of the existing system about:

- The decomposition of a system. The use case maps provide the components with their responsibilities
- The flow of events across the structure. The use case maps provide a flow of events through components with one start and one or several end points

The scenario flow is presented by use case maps cutting across the structure. For example, Figure 8 illustrates an overview of the use case map *Arm area delayed - Overview*. Three components are participating in this scenario: the user interface Keypad, Control Panel, and a Motion Detector. The Motion Detector is synonym to a point in a security system. A point is a signal source/destination, such as a sensor or actuator. An area is a collection of points. Arming an area changes the points into a state where sensors can identify security breaches in the vicinity. A delayed arm changes the state of all points in an area to 'arm' after a given time. This allows, for example, employees to leave an office space without generating alarms.

The use case starts with a solid circle (start point) representing a user typing an *arm area delayed* command at a Keypad. The Keypad translates the keystrokes into logical information and provides the information to the Panel. The Panel collects the status of a sensor (Motion Detector) detecting if the sensor is already armed. The sensor is the only point in the area. The 'X' on a use case map represents a responsibility. The paths of the use case map end with the bar termination symbol. The full use case map notation is described in [4].

The use case map overviews created in Step 3 are refined in Step 4 as illustrated in Figure 9. The lower decomposition level allows the identification of missing responsibilities of the existing architecture [16]. For example, the component Area is not present in the existing architecture, because the existing panel supports a different grouping concept for sensors. The architects of the existing system suggest a solution on how to integrate the missing responsibilities in their structure. The solution addresses either refinements without architectural impact, or architectural modifications of the existing architecture [16]. The component Point in Figure 9 had to be extended by the architects with a sub-component Area. Changes to the architecture and their impact are recorded in the use case maps.

Steps 3 and 4 can include quality attribute concerns, such as formulating performance properties along the use case maps, or modifiability scenarios with slot concepts [4] to exchange or parameterize components along paths.

4.3. Analysis

Finally in Step 5 an analysis of the developed use case maps is carried out. The use case maps were reviewed with walkthrough techniques to evaluate the paper-based models [24].

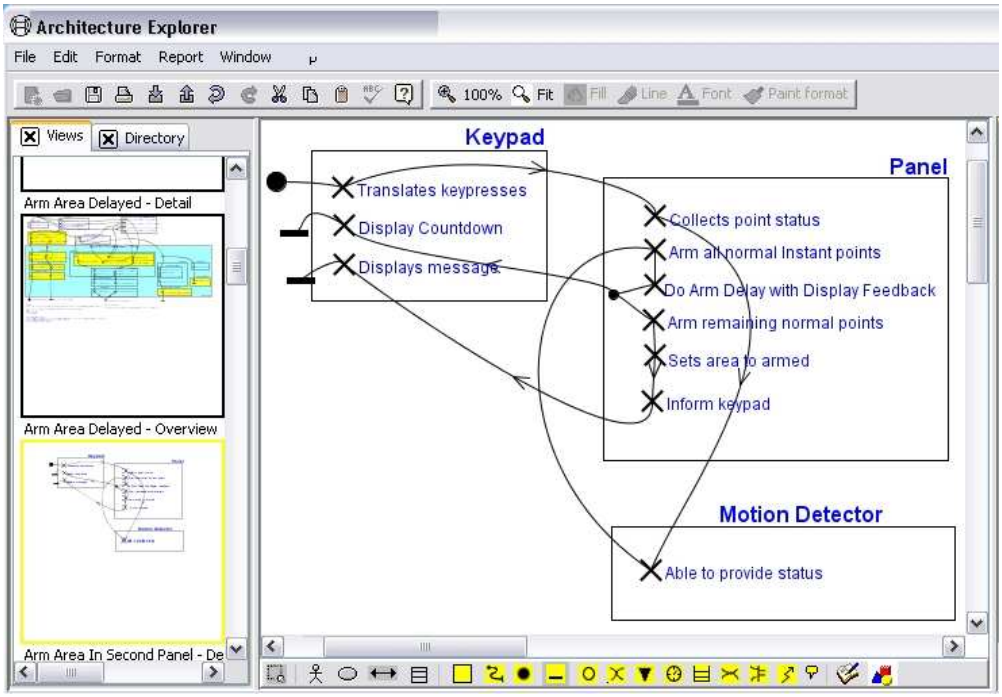

Figure 8. Use Case Map 'Arm Area Delayed - Overview'.

The developers illustrated how the system would respond to a scenario by explaining the use case map paths to the reviewers. The analysis results included:

- A set of missing responsibilities and components and a suggestion as to how to modify the existing architecture.
- Open issues currently unresolved, such as a multi-panel setup in a distributed system.
- A set of risks and sensitivity points, such as performance bottlenecks or internationalization with user-dependent language support.
- An initial effort estimation to adapt the architecture of the existing system.

The process outlined above was performed with 5 engineers in 5 consecutive days. Around 10 use case maps were developed in Step 3 and refined in Step 4. The major benefits of this process were:

- It put the developers from fire, access, and intrusion systems in the same room.
- Enforcement of a clear presentation of the existing architecture and its concepts.
- Increasing domain understanding, as fostered with the factory story.
- Clarified terminology, such as groups, areas, and protection points.

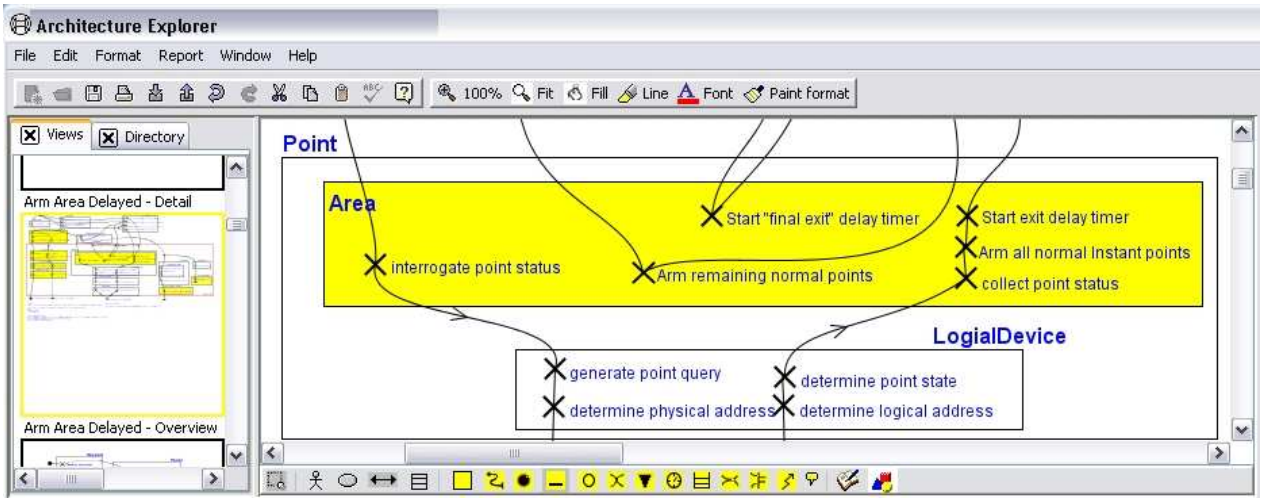

Figure 9. Use Case Map 'Arm Area Delayed - Detail'.

- Understanding of domain commonalities and differences.
- Documented difficulties and limitations of the existing architecture and potential for improvements to satisfy the requirements of the new system.

The evaluation process provided fundamental insights in the architecture of the existing system and the impact of the new requirements. The process that we carried out with the developers of the organization was driven by decomposing the system, identifying and analyzing components and responsibilities in the architecture. The analysis results produced in Step 5 did not uncover composition concerns due to the decomposition-oriented bias. We will discuss the reasons for this limitation in the following sections.

4.4. Other Methods

There are mature architecture evaluation methods available, such as the Architecture Tradeoff Analysis Method (ATAM) [7] and the Software Architecture Analysis Method (SAAM) [14]. Both methods are mainly quality attribute and stakeholder driven. In this case study, we selected an approach that considers the experience of the involved developers and the organizational situation. The emphasis was on understanding the sequence of events in the architecture design and the decomposition decisions, reconstructing missing information, interviewing developers on request, applying change and performance scenarios, and estimating change efforts. This forced us to flexibly adapt the method steps to the organizational situation promoting an efficient exchange and evaluation of information between two development teams.

5. The Composition Paradox

Decomposition is primarily driven from a designer's perspective to create a suitable partitioning of the system that can manage its complexity (divide and conquer). Accordingly, the evaluation process of the previous section with the component and responsibility identification from use case maps was primarily driven by designers evaluating and navigating the decomposition structure.

5.1. Limitations

Severe problems with a designers perspective solely on decomposition arise when the system has to be composed from its parts. Some examples are:

(1) Assembly - Hardware and software components have to be verified, packaged, and deployed. In security systems Fire, Access, and Intrusion elements may have to be packaged independently, because the low-end computing platforms cannot afford additional code and data resources to store intrusion and access elements in a fire-only system.
(2) Scalability - Architectures have to keep pace with changes and growth. For example, commercial security systems require a distributed approach for control panels to manage large sensor systems; a network of control panels should be presented as a single system to users; fire systems support few users whereas access systems have thousands of users to provide employee access in larger buildings.
(3) Location Transparency - Elements of the architecture have to be distributed transparently in networks. For example, an Access element of a security system may reside on a PC platform and communicates with Fire and Intrusion elements located on proprietary control panels.
(4) Market differentiation - Features of systems are required for particular markets whereas they are optional in others. In this case study, Fire, Access, and Intrusion elements must be customized for different markets to fulfill local security standards.
(5) Interoperability - Components may be implemented in different programming languages and interoperate with several third party software packages. In security systems, control panels operate with management systems from different providers.

The common theme in this list is that there is no architecture view available at design time about how the system is deployed at installation or startup time. Integrators do not have rules to assemble and deploy the software in diverse deployment settings. One reason is that components from the decomposition view do in many cases not directly translate to components at installation time. A further reason is that components will be deployed on many platforms. Components turn from containers capturing some functionality/responsibility to components that have to communicate via process and language boundaries, involving network communication, fault tolerance and consistency requirements.

The above listed examples from the integrators point of view let the system look different than the decomposition structure of Figure 6. Are we still talking about the same system? Unfortunately, this is the case. One reason for the disparity was already mentioned in Section 3:

Allowing short-cuts in the design phase and focusing only on the next product version ignores deployment and customization concerns.

5.2. The Paradox

However, there is a more significant reason for this problem, which is rooted in the composition paradox: **The Composition Paradox is the tendency of the decomposition to produce rather monolithic software systems whereas composition tends to produce more componentized software systems.** The originally well-defined decomposed architecture is compromised by several obstacles:

- Tight coupling. The component barriers are lowered by tight coupling of component interactions. The relation between coupling and cohesion is shifted towards coupling. A typical example is the dismantling of object access restrictions. The interested reader is referred to [15] for further details on cohesion and coupling from a reverse engineering perspective.
- Disappearance of explicit component interfaces in follow-on development steps. For example, objects of one component access objects contained in another component without using an explicit component interface. Composition enforces rules how parts are assembled to a system and therefore restricts the design and implementation to follow these rules. With a primary focus on decomposition, subsequent development steps (detailed design/implementation) do not have composition rules to conform to.
- Implicit synchronization on data and component execution sequences via the restriction to a particular scheduling algorithm, such as preemptive fixed priorities. Enforcing execution sequences via fixed priorities adds synchronization uncertainty in case the scheduling changes caused by a port to a different platform.
- Startup Manager for central component configuration. The responsibility for the configuration and parametrization of a component is shifted from the individual component to a central manager. This added configuration sequence dependencies that had to be resolved from the startup manager.

The decomposition bias makes it difficult to integrate composition concerns during the course of refinement and implementation. Boundaries between components and their interactions will eventually be dismantled, thus resulting in more monolithic systems that will not support integrators in assembling systems from components for various customer configurations. A complementary view on composition tends to enforce and restrict components and their interactions early in the design because it has to ensure the assembly of a system from its parts, thus resulting in more componentized software systems. Composition is therefore primarily not a feature but rather a quality concern that has to be addressed throughout the life-cycle.

Interestingly, a conformance analysis of the as-implemented and the as-designed architecture can result into no discrepancies in a decomposition driven design. The decomposition structure may still not be violated as long as there is clear traceability between the implementation and the design. For example, explicit component boundaries can disappear in the implementation. But a rule that associates files in a directory with a design component still allows the

traceability of the implementation to the design. Although the implementation conforms to the design it can prevent composition due to missing architectural decisions, e.g. enforcing explicit component interface objects.

6. Decomposition and Composition Foci

Composition is concerned with assembling, verifying, and packaging a system (hardware and software) from its parts. One of the foremost questions is: does the system perform in a concrete customer setting as promised? If there are many customer configurations, then they are hard to predict at system design time. Examples are customers that want to integrate a security system into a building system, or a scalable system of security panels to build a security network operating up to 100,000 sensors.

Experts suggest the usage of deployment views to tackle allocation of software on concrete topologies, describing CPU, memory, disk space, and bandwidth properties [6]. Referred elements of the decomposition view are [24]: types of hardware required, specification and quantity of hardware required, third-party software requirements, technology compatibility, network requirements, network capacity required, and physical constraints. This perspective still underestimates the impact of the customer specific composition task that the distributed control panel software is especially challenged with.

Additionally, the proposed models of the deployment view are [24]: runtime platform models, network models, and technology dependency models. These models do not suggest a composition model in the deployment. Deployment describes the environment into which the system will be deployed, including dependencies the system has on its runtime environment. Composition ensures the construction of a system from its parts and verifies the construction and properties for particular deployments.

The different foci of the composition and decomposition activities are listed in Table I. Decomposition is one of the most prominent activities for a designer to tackle requirements and design complexity. The partitioning is done along separation of concerns resulting in break-down structures. The analysis of the design is done to verify the decomposition decisions, driven top-down from the requirements.

Composition is often one of the main concerns of system integrators. They have to assemble parts to fulfill system promises in concrete customer settings. Here, the system hits the reality of the customer. The architecture has to ensure structures that allow for predictable component assembly and explicit mechanisms in the architecture for component isolation and interaction. The integrator is confronted with component versions and their compatibility in different configurations that contribute to the complexity of the packaging task. Due to proprietary customer networks and security policies, the Integrator has to predict if the product with its assembled and deployed components still can deliver on its promises for the particular customer configuration.

The different foci of decomposition and composition became a severe concern in the architecture design activities of the case study. The integrator's concerns were not sufficiently accounted for at the beginning of the development of the control panel's software. One of the reasons was the focus on versions for initial customers in the architecture design due to time

Table I. Decomposition/Composition comparison.

Decomposition	Composition
Designers perspective	Integrators perspective
Design complexity	Integration complexity
Separation of concerns	Assemble separated concerns
Separation of requirements	Overall requirements
Break-down structures	Packaging structures
Design analysis	Assembly analysis

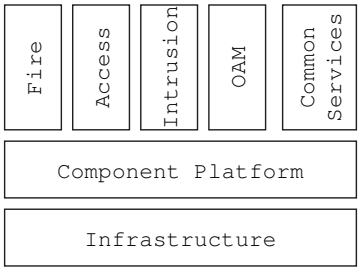

Figure 10. Runtime view.

and resource limitations. A re-design of the decomposition-driven architecture occurred when the runtime and deployment structures of the system were taken seriously.

Figure 10 illustrates a runtime view. Of particular interest are the Common Services and the Component Platform elements. The former Common element from Figure 6 mixed two different objectives:

- Common shared services between Fire, Access, and Intrusion, such as a Printer Service, Sensor Node Information Service, or a Communicator that reports alarms to call centers.
- A common object-oriented application framework that captures a common object model for Fire, Access, and Intrusion.

The common application framework is not visible in this view because it is not a separately deployable component. However, the common application framework is important for further decompositions to identify commonality and variability for Fire, Access, and Intrusion. Common Services are already identified as independently distributed elements and therefore visible in the runtime view.

A key element for this approach is the Component Platform, which is an embedded component middleware similar to the *Pervasive Component Systems* approach [19]. The Component Platform supports a component model that primarily provides a set of rules

for component types, such as active and passive components, and component interactions. For example, components are only allowed to communicate via the component platform and explicit component interfaces, or software exchange will be possible on a component level. The Component Platform was originally not present in the software architecture due to the lack of composition concerns. Introducing the Component Platform along with a component model allowed for enabling assembly.

An interesting observation is that the runtime view has the tendency to reduce or hide layers that were introduced during decomposition. The reason is that a Component Platform is mainly a broker for components. It does not know about layers and treats components equally, independent on their place in the decomposition hierarchy. In fact, the boxes in Figure 6 and Figure 10 have a different meaning. Figure 10 identifies components, present at deployment and runtime. Composition enforces the identification of deployable components in the system. Figure 6 identifies elements in the meaning of layers that are design time entities that are possibly without rhyme or reason at deployment or runtime.

Overdue considerations of compositional aspects in the architecture design and implementation will lead in many cases to expensive modifications. The reason is that composition is not associated with one particular piece of software, such as the Component Platform in the case study, but rather a cross-cutting concern that affects many components and relations.

Decomposition and composition are complementary efforts and equally important to establish a well-designed software architecture. Missing composition results in monolithic systems; missing decomposition results in systems with unmanageable complexity, such as leaving out the reuse enabling opportunity of an application framework for Fire, Access, and Intrusion.

7. Compositionality

Software architectures are in most part determined by quality attributes and their tradeoffs [1]. The Control Panel decomposition of Section 3 is determined by quality attribute goals for modifiability, including portability and scalability, and reusability. The authors of [1] suggest the use of quality attribute reasoning frameworks to analyze structures that must achieve particular pre-set quality attribute goals.

With respect to composition, we need an analysis framework to achieve two objectives:

- Determination of valid or invalid configurations.
- Suggestions for design improvements.

The purpose of both objectives is to improve and verify the design on an architecture level. Consequently, detailed methods and exact values, such as throughput values, are not known or speculative in this phase. Figure 11 provides an overview of the concept of validating the Composition. A *Configuration* consisting of components, hardware modules, and a topology has to be validated if it satisfies the expected product properties (*ExpectedProductProp*). It also can provide *Suggestions* on how to improve the design in order to fulfill *ExpectedProductProp*.

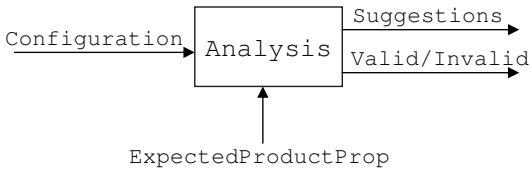

Figure 11. Validating a composition.

7.1. Configuration Example

Figure 12 illustrates a scenario for *Configuration*. An example configuration of a security system consists of four hardware modules: 3 Control Panels with one panel using a PC platform, and a Gateway. We assume that all hardware components have the software components Infrastructure and Component Platform. The panels are connected via a backbone network. A *Gateway* module is connected to *Panel Intrusion*. *Panel Intrusion* does not have a *Communicator* service. *Panel Fire* has a *Fire* component and a *Communicator* connected to a mobile network via a standard GSM module. The Access Panel (*PC Access*) has a *Communicator* connected to GSM and ISDN. The *Gateway* has a *Switching* component and a *Communicator* connected to Ethernet. The sensors and actuators connected to the panels and gateway are left out to concentrate on the *Communicator*. Only one *Communicator* should be used at a time, with a policy that prefers GSM over ISDN over Ethernet.

The analysis of this configuration provides information as to whether or not this system can be constructed. The analysis of the architecture verifies if the architecture incorporates mechanisms that supports the configuration. The decomposition as illustrated in Figure 6 does not provide this information. The runtime view of Figure 10 could provide hints, but is not meaningful enough on the box-and-arrow level. This was basically the situation in the case study, when the composition aspects were reviewed. An important review goal for decomposition-biased architectures is an evaluation if customer configurations can be actually composed from the system components. Although the scenario was not important for the first release of the product it had significant impact on the entire life-cycle: the total cost of ownership of a decently designed system where composition aspects are taken explicitly into account is much lower than by ignoring this in the first release.

The analysis framework resulted in the following suggestions:

- The Component Platform should be provided on Gateways and Panels. Due to little- and big-endian differences, the component platform has to accommodate for this.
- Components should incorporate metadata at the interface, for example information about the available communication interfaces (GSM, ISDN, and Ethernet) of a particular Communicator.

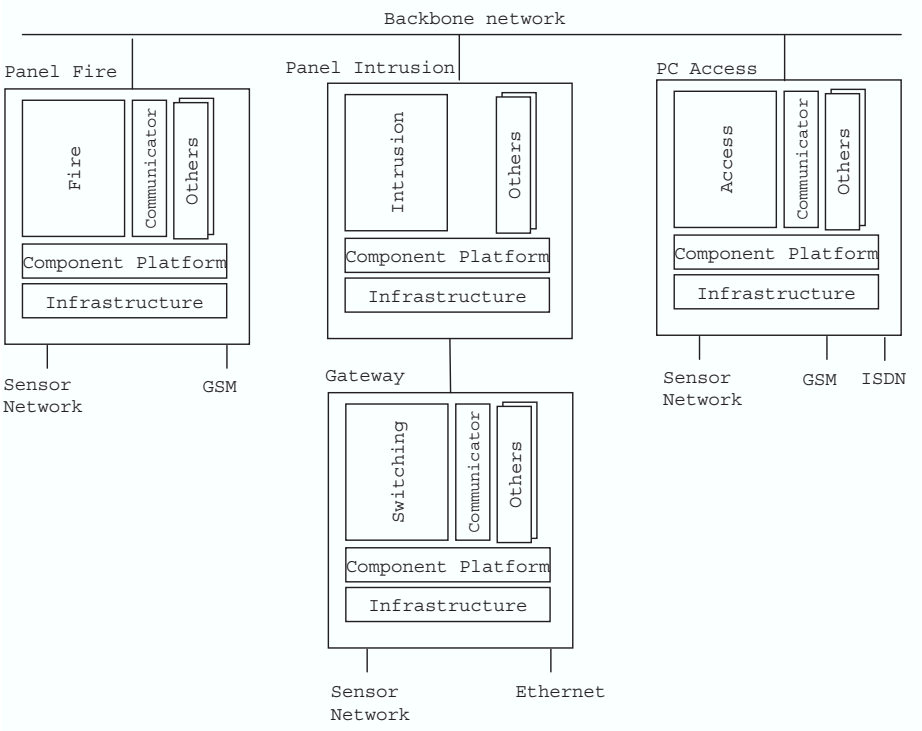

Figure 12. Simplified deployment view.

- An Interface Definition Language [27] should be incorporated. The architecture gains flexibility by encapsulating the internal structure and mechanism for objects inside a component boundary that other objects are not allowed to penetrate.

Additionally, the analysis framework resulted into suggestions about quality concerns such as fault tolerance and performance:

- Depending on the fault model it is acceptable that Communicators are temporarily unavailable. An alternative Communicator has to be identified in the network. Also, panel data is replicated over the backbone network.
- Depending on the performance model the backbone network requires a minimum bandwidth for a particular amount of sensors.

The analysis framework represents the integrators perspective. The integrator is also a source for representative configuration scenarios.

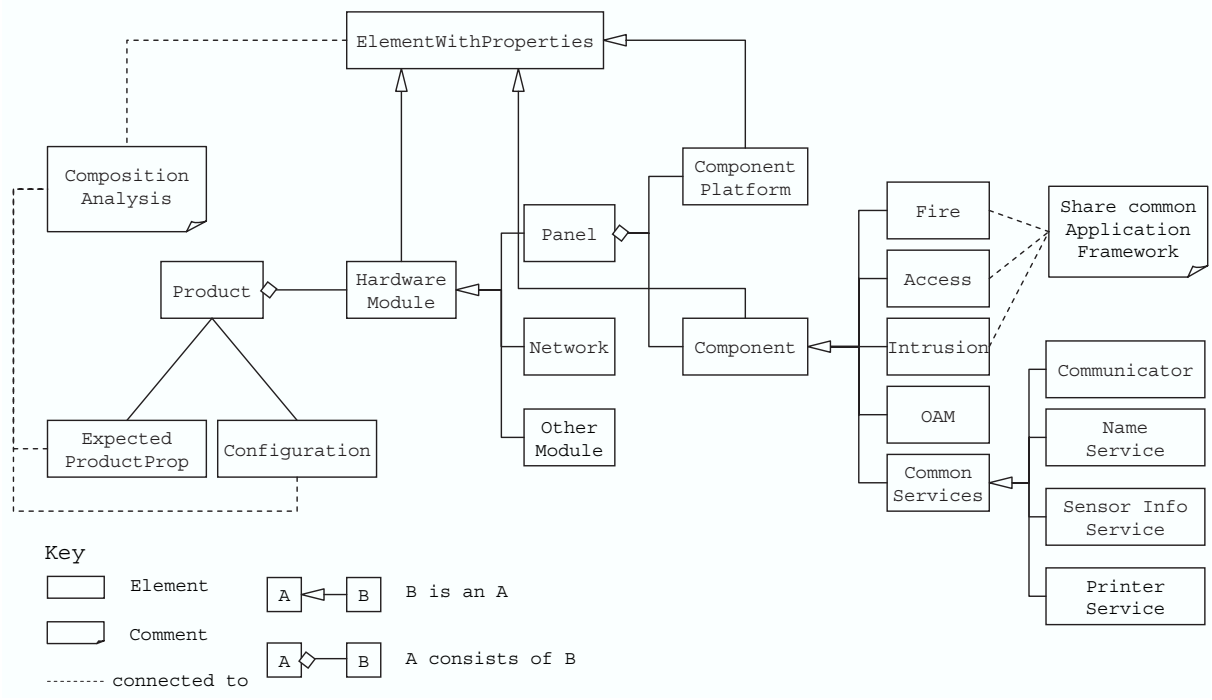

Figure 13. Mapped Composition.

7.2. Component Model

The resulting design decisions will impact the architecture, for example by requiring an explicit component model. Figure 13 illustrates an initial model. A *Product* consists of several *Hardware Modules* that can be *Panels*, *Networks*, or *Other Modules*, such as gateways to wireless sensor networks. A *Panel* consists of a *Component Platform* and an ensemble of *Components* that communicate via the *Component Platform*. *Configurations* for each customer can look different, in particular in the commercial market, as introduced in Figure 12. *Hardware Modules* as well as *Components* and *Component Platform* have properties (*ElementWithProperties*) that expose quality properties. A property provides a meaningful value, for example

- worst-case execution time (WCET) for a particular method at the component interface;
- available RAM and ROM of a particular *Hardware Module*;

The Composition Analysis includes rules for valid Configuration, for example:

- Each *Panel* has a *Component Platform*.
- *Panel* has at least one component out of *Fire*, *Access*, and *Intrusion*.

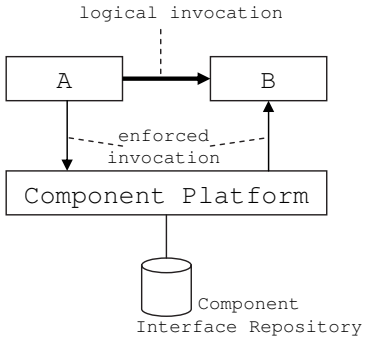

Figure 14. Impact of Component Model.

- Data of the *Sensor Info Service* has to be replicated in Commercial markets.
- *Network* requires certain bandwith with protocols A or B.

With the Figures 6, 10, and 13 we developed three different views of the architecture. Views allow the understanding of architectures from different perspectives [6]. Figure 13 provides a view where the decomposition elements of Figure 6 and the composition elements of Figure 10 are mapped. The Common element of Figure 6 is refined as Application Framework and connected to the Fire, Access, and Intrusion elements of Figure 13. Other parts of Common are connected to Common Services. Mapping elements of different views is important to keep the consistency of the architecture description. The wrong approach is to omit the concerns of particular views and to overlay them in one, sometimes cluttered, view. However, the elements of different views should be related to understand the overall context as indicated in Figure 13.

Figure 13 poses the question if the illustrated architecture components are conceptual components of the design space or components in the implementation space. Our argument is that the components will remain as hard boundaries throughout the realization. Of course, conceptual components will be refined, interfaces will have additional and more detailed information with stronger commitments, property values will be measured and refined in the component interfaces, further components will be detected, and components can be split up into further components. However, the overall blueprint will not change in the sense that the component structure will be blurred, or fault-tolerance and performance concepts will be changed without re-evaluating and adapting the architecture.

The component model introduced in the security system of the case study ensured that components of the composition can use other components only via the component platform as illustrated in Figure 14. There is no short-cut for component invocations. This does not imply that a late binding mechanism exchanges the indirect invocations with direct invocations.

Additional significant advantages of the introduced composition model are:

- Qualities outside of the scope of individual components have an explicit place. For example, the Component Platform provides performance and fault-tolerance guarantees for local and remote invocations.
- Components are decoupled and rely on using the appropriate interface from component interface repositories.

The composition model allows a verification whether customer configurations are valid before system installation. This provides a significant advantage to achieve the *low installation cost* goal of the organization that we introduced in Section 2. Currently, we automate the verification with respect to performance with the TimeWeaver approach [10]. The tool provides a performance analysis depending on a set of given components and a system topology. A script language allows the automation of the analysis by selecting communication protocols, operating system and hardware characteristics, and component placements in the distributed system.

8. Related Work

Several areas of work relate to the presented case study. This includes

- The discussion of a fire-alarm system for a Swedish security company.
- Architecture Documentation.
- Aspect orientation and generative programming.
- Composition in the context of predictable assembly of software components.

In the following we discuss how the work areas relate to the presented composition paradox in this paper.

Jan Bosch et al discusses the software architecture of a fire-alarm system [2]. The fire-alarm system is used to illustrate a method for software architecture design and a product line approach. The focus of this paper is to use security systems (not fire-alarm systems solely) to illustrate the composition paradox. Further, intrusion systems are different from fire-alarm systems in two major ways: process-orientation and resource constraints. Process rules, such as various security levels for rooms depending on particular users and the time of the day, have to be programmable. Intrusion systems have to consider for access privileges several thousands of users resulting in different message throughput requirements and user management approaches. A software design for fire-alarm features only will therefore not satisfy the requirements for a combination panel.

Architecture documentation communicates design decisions to various stakeholders of a system. Examples for significant work in architecture documentation can be found in [6] and [23]. The cited work underlines the different view points of stakeholders on a system. This conforms to our discussion about relating elements from the decomposition and composition perspective.

Composition is in many cases a cross cutting concern. An example in this paper is the discussion on component configuration and parametrization for different customers in Section 5.2. Decomposition and composition as cross cutting concerns are discussed in aspect

orientation [11] and generative programming [9]. However, both bodies of work do not address the composition paradox in an architecture design approach and solutions for composition of components in different customer configurations.

Predictable assembly of software components is the composition of systems from existing and third party software components. The focus is on assembly rather than component design or coding. Composition and compositional reasoning are a central concern. Approaches are presented in [28] and [8]. Key is a component technology with a component model that allows reasoning for prediction. With this the approaches promise integrators a way to predict the behavior of new assemblies. However, [28] and [8] do not present how component composition approaches relate to architecture design approaches.

9. Lessons Learned

Decomposition and composition are two complementary activities in the architecture definition process. Both activities address different concerns of the system. Decomposition strives to achieve a coherent partitioning and break-down structure. Composition strives to ensure assembly of a system from its parts and bundle it for particular verified configurations. An architecture design driven with only decomposition in mind results in the **composition paradox**, where architects try to achieve a well-defined partitioning but integrators are confronted with monolithic implementations and highly coupled elements.

In summary, we capture the following lessons learned:

- Composition is largely driven by integration concerns. Neglecting these concerns during the architecture definition process can result in systems that are hard to assemble or integrate into diverse customer contexts. A complementary reflection of decomposition and composition would prevent the decomposition tendency to produce structures that end-up as monolithic systems in the implementation. In the case study it turned out that adding an explicit component model in the architecture solved the composition problem in this concrete case. The component model enforced rules that had to be followed throughout the design, implementation, and test.
- Decomposition and composition should be described in views that are complementary. The elements have different meaning and abstraction levels. Elements from the decomposition view do in many cases not directly translate to components at composition or installation time. Therefore both views should not be combined into one view. A separate view must describe how the decomposition elements are linked to the elements that are relevant for composition.
- Decomposition and composition are important for building architectures. They are also important for evaluating architectures. A decomposition driven evaluation will miss a rich variety of customer scenarios that require flexible assembly of systems from its parts. Later integration of composition techniques is tremendously more expensive because they require significant changes in the architecture, such as the explicit integration of a component model.

- A key success factor in the development of software architectures is the understanding of quality attributes and their impact on structures. The case study illustrated that, for example, performance and scalability with several control panels had impact on the decomposition as well on the composition. For decomposition they provided verification of structures and suggestions how to improve hierarchies. For composition they suggested component models and impose rules on further refinements in order to allow for part assembly.
- Discussing composition in the case study with the organization uncovered new business and technical opportunities, such as the transparent usage of a Communicator in a Gateway or the composition assistance tool for contractors to support installations.

Security systems with large sensor networks are long-lived systems because of significant customer investments. Software in security systems is confronted with rapid improvement of sensor technologies, fusion of sensor information, and new deployment scenarios. Composition of software and hardware components and the ability to respond to changes in deployed software are important goals to secure customer investments. The case study illustrated that composition is important to achieve this goal. We exposed that architectural designs only driven by decomposition will severely hamper this goal. A complementary perspective on composition and decomposition will avoid the composition paradox and result into better decisions about what parts are hard to change and what parts will be easy to change.

REFERENCES

1. Bass L., Clements P., Kazman R. *Software Architecture in Practice* (2nd edn). Addison Wesley, 2003.
2. Bosch J. Software Variability Management. *Science of Computer Programming*. **53**(3):255-258, December 2004.
3. Brooks F. *Mythical Man-Month: Essays on software engineering*. Addison Wesley, 1995.
4. Buhr R.J.A., Casselman R.S. *Use Case Maps for Object-Oriented Systems*. Prentice Hall, 1996.
5. Callaway E., *Wireless Sensor Networks: Architectures and Protocols*. Auerbach Publications, 2003.
6. Clements P., Bachmann F., Bass L., Garlan D., Ivers J., Little R., Nord R., Stafford J. *Documenting Software Architectures: Views and Beyond*. Addison Wesley, 2002.
7. Clements P., Kazman K., Klein M. *Evaluating Software Architectures*. Addison Wesley, 2002.
8. Crnkovic I., Stafford J.A., Schmidt H.W., Wallnau K. Component-Based Software Engineering, *Proceedings of the 7th International Symposium, CBSE*, Edinburgh, UK, May 24-25, 2004.
9. Czarnecki K., Eisenecker U. *Generative Programming: Methods, Tools, and Applications*, Addison-Wesley, 2000.
10. de Niz D., Rajkumar R. TimeWeaver: A Software-Through-Models Framework for Real-Time Systems. *Proceedings of the 2003 Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES'03)*. San Diego, USA, 2003.
11. Filman R., Elrad T., Clarke S., Aksit M. *Aspect-Oriented Software Development*. Addison-Wesley, 2004.
12. Fowler M., Who needs an architect? *IEEE Software*, September/October, **20**(5):11-13, 2003.
13. Kalinsky D. Design patterns for high availability. *Embedded Systems Programming* **15**(8):46-61, August, 2002. , August 2002.
14. Kazman R., Bass L., Abowd G. and Webb S.M. SAAM: A method for analyzing the properties of software architecture. *Proceedings of the International Conference on Software Engineering - ICSE'16*, pages 81-90, 1994.
15. Krikhaar R.L., *Software Architecture Reconstruction*, Ph.D. Thesis, University of Amsterdam, 1999.
16. Klusener A. S., Laemmel R., Verhoef C. Architectural Modifications to Deployed Software. *Science of Computer Programming*, **54**(2–3):143-211, 2005. Available at `www.cs.vu.nl/~x/am/am.pdf`.
17. Lala J.H., Harper R.E. Architectural principles for safety-critical real-time applications. *Proceedings of the IEEE Publication* **82**(1):25-40, 1994.

18. Leffingwell D., Widrig D. *Managing Software Requirements: A Unified Approach*, Addison Wesley, 2003.
19. Li S., Wu J, Hu Z. A Contract-Based Component Model for Embedded Systems. *Quality Software, Fourth International Conference on (QSIC'04)*, pp. 232-239, 2004.
20. Liu C. L. and Layland J. W. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *Journal of the ACM* **20**(1):46-61, 1973.
21. Local Security Network, `http://www.boschsecurity.com` [Current August 2005].
22. McComb D. Who Needs Software Architecture? `http://semarts.com.decisivenet.com`, [November 10, 2003].
23. Nuseibeh B., Kramer J., Finkelstein A. A Framework for Expressing the Relationships Between Multiple Views in Requirements Specification. *Transactions on Software Engineering* **20**(10):760-773, 1994.
24. Rozanski N., Woods E. *Software Systems Architecture*, Addison Wesley, 2005.
25. How do you define Software Architecture?, `http://www.sei.cmu.edu/architecture/definitions.html` [Current August 2005].
26. Shaw M. and Garlan D. Software Architecture: Perspective of an Emerging Discipline. Prentice Hall, 1996.
27. Siegel J. *CORBA 3 Fundamentals and Programming* (2nd edn). John Wiley and Sons, 2000.
28. Wallnau K. *Volume III: A Technology for Predictable Assembly from Certifiable Components*. SEI Technical Report, (CMU/SEI-2003-TR-009)
29. Xu Y., Heidemann J. and Estrin D. Adaptive Energy-Conserving Routing for Multihop Ad Hoc Networks, 2000. Available at `http://www.isi.edu/~johnh/PAPERS/Xu00a.html` [Current August 2005].