Concern Impact Analysis in Configurable System Software—The AUTOSAR OS Case[*]

Wanja Hofer, Daniel Lohmann, Wolfgang Schröder-Preikschat
Department of Computer Science 4
Friedrich-Alexander University Erlangen-Nuremberg
{wanja,lohmann,wosch}@cs.fau.de

ABSTRACT

System software for cost-sensitive special purpose-systems has to be configurable and tailorable. AOSD should be beneficial for this purpose, as it provides means to untangle the system's concerns in a very fine-grained way. An important prerequisite for a fine-grained software design based on aspects is, however, that all concerns and their interactions present in the system have been comprehensively captured and understood.

We propose a method called <u>C</u>oncern <u>I</u>mpact <u>A</u>nalysis for this purpose. Based on a system's specification, CIA provides a guideline to iteratively grasp the concerns present in a system, and their interactions. A speciality of CIA is that it also takes unspecified "internal" concerns into consideration as early as possible. We have tested CIA with the AUTOSAR OS specification and the design of our CiAO operating system family, where it led to a very fine-grained, aspect-aware kernel design.

Categories and Subject Descriptors

D.2.2 [**Software Engineering**]: Design Tools and Techniques; D.4.7 [**Operating Systems**]: Organization and Design—*Real-time systems and embedded systems*

General Terms

Languages, Experimentation, Design

Keywords

Aspect-Oriented Design, AOP, AOSD, CiAO, Configurability, AUTOSAR, Aspect-Aware Operating System

1. INTRODUCTION

1.1 Motivation and Background

The employment of AOSD principles in the design and implementation of system software is often said to lead more or less "automatically" to a higher level of tailorability. Tailorability, in the sense of being able to leave out everything that is not needed, is one of the most important properties in the domain of cost-sensitive special-purpose systems, such as automotive embedded systems. AOSD fosters tailorability by enabling the designer to untangle the system's concerns in a more fine-grained way than by using traditional decomposition techniques.

However, these benefits of AOSD can not be taken for granted. It has been known for a long time that if tailorability is to be reached in system software, it must be pursued as a primary design goal from the very beginning [11]. In our studies on increasing the separation of concerns in the PURE and eCos embedded operating systems [14, 8], we learned that this also holds if AOSD is applied. In both cases, AOSD was incorporated at a relatively late stage of the project and merely used to refactor already implemented concerns into aspects. While this worked well with some concerns (the application of AOP was, overall, successful), it also turned out to be suprisingly difficult with several other concerns—some of which we had assumed in advance as "ideal candidates" for a separation by AOP. The design and implementation of those "refusing" concerns was dependent on other concerns in a way we had not expected from their functional specification. In most cases, these subtle dependencies were caused indirectly via a third, internal concern. In the eCos kernel, for instance, we were not able to completely separate the (conceptually independent) concerns of *interrupt synchronization* and *thread synchronization*, because both subtly interacted with each other via design decisions regarding the *dispatching strategy*. This strategy is commonly considered as implementation-internal and, as such, not described in a system's functional specification.

1.2 CiAO and AUTOSAR

Whereas in the PURE and eCos studies AOP was employed at a relatively late stage, it has been used in the CiAO project from the very beginning. The research question behind CiAO is *if* and *how* the application of AOSD can lead to operating system designs that advance the state of the art with respect to configurability and tailorability. By means of an *aspect-aware kernel design*, CiAO aims at tailorability and configurability of even fundamental architectural and non-functional OS concerns, such as *isolation* or *synchronization* [9, 10]. The kernel should be *aspect-aware* in the sense that it is itself minimal, but offers all necessary join points to be ideally extensible by aspects, which implement the actual
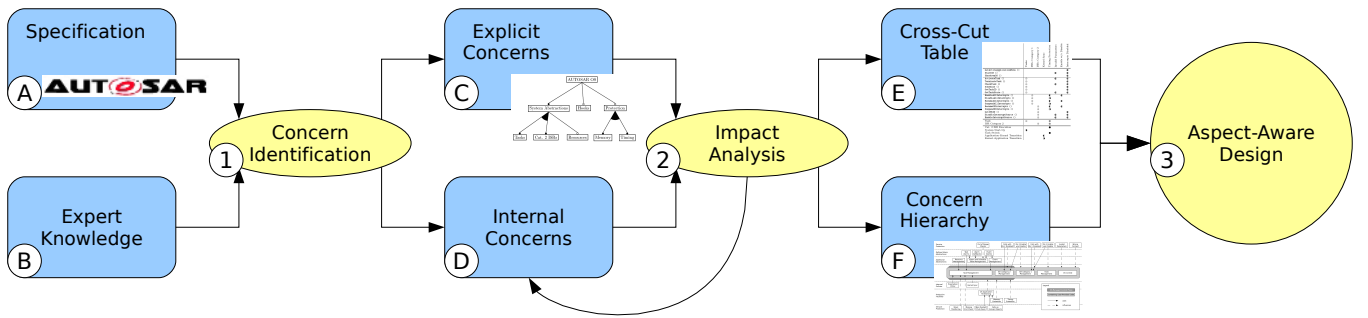
Figure 1: The process of concern impact analysis: In step 1, all concerns present in the target piece of system software are identified by both extracting explicit concerns (C) and internal concerns (D) from the specification documents (A); some of the system-internal concerns (D) can only be gathered using systems-engineer expertise (B). Some of the internal concerns, however, will only turn up when analyzing the other concerns (iterative step). Step 2 consists of an analysis of the impact of all concerns on the abstract system; the results can be depicted in a matrix cross-cut table (E) exposing important join points and a concern hierarchy (F) revealing concern dependencies and influences. Both diagrams facilitate the following aspect-aware design development (3).

system features.

The CiAO OS prototype is based on the OS specification adopted by AUTOSAR [2], a consortium founded by all major enterprises in the automotive industry in order to specify a new system software standard for car applications. Their OS specification [1] defines the functionality and API of the AUTOSAR OS layer. An interesting point in the AUTOSAR OS specification is that it actually *does* include tailorability with respect to some functional and architectural concerns to a certain extent. This makes it a promising subject for aspect-aware kernel design.

To reach optimal tailorability in CiAO OS, it was necessary to grasp all concerns and their necessary interactions present in the system. A major challenge was to avoid problems similar to those that we found in PURE and eCos: unanticipated dependencies that emerged subtly (and probably accidentally) by design decisions regarding internal concerns that are not part of the system specification.

1.3 About This Paper

For this purpose, we developed the method that is the subject of this paper. The method of *Concern Impact Analysis (CIA)* provides a guideline to iteratively grasp the concerns (including internal concerns) present in a system, and their interactions. The eventual goal of CIA is to prepare for and guide to the aspect-aware design of a piece of system software.

The rest of the paper is structured as follows: In Section 2 we present our method both in general and with the example of AUTOSAR OS. In Section 3, we discuss some issues of our approach; Section 4 describes some related work, and the paper is concluded in Section 5.

2. THE PROCESS OF CONCERN IMPACT ANALYSIS

Figure 1 briefly depicts and describes the process of concern impact analysis as we propose it. The goal is to provide the system software designer with the necessary input to build a flexible and tailorable, aspect-aware design. A special focus

is given to the system-internal concerns (D), which are often not directly reflected in a system specification. However, it is particularly those concerns that have to be respected from the very beginning when designing a system using AOP because they tend to affect the system in very peculiar places and are therefore hard to add to a system *ex post*. Often it is only during the analysis of the impact of *other* concerns (step 2) that system-internal concerns and *their* impact are revealed and can then be investigated further. Hence, that part of the concern impact analysis is an iterative process.

The rest of this section presents the method and its steps in a more thorough form using the analysis of an AUTOSAR OS kernel as an example.

2.1 Step 1: Concern Identification

As mentioned in Section 1.2, the AUTOSAR OS specification [1] records the requirements on an embedded real-time operating system. It is very well suited in our context since it is partly focused on configurability through the postulation of so-called *scalability classes*. These scalability classes provide an indication for concerns that are to be kept configurable in an AUTOSAR system; however, there are only four of them, so they only provide for a very coarse-grained configurability.

Hence, we had a thorough look at the specification (artifact A in Figure 1) and identified features[1] hidden both in the plain-text specification and the specification of the operating system interface (step 1 in Figure 1). Those features were then classified and arranged in a *feature diagram* [6], which reflects the variability points of the configurable system to be built. Figure 2 depicts a very small part of the feature diagram we built for the AUTOSAR OS specification.

As stated before, however, there are *system-internal concerns* (artifact D in Figure 1) that are often not reflected in a system's specification. Classic operating system concerns of that category include synchronization, isolation, and interaction. In AUTOSAR OS, for instance, an isolation mechanism is explicitly defined (named memory protection there), but

---

[1] A feature is a concern of importance to a certain stakeholder [6], in this case the system deployer and the application programmers.
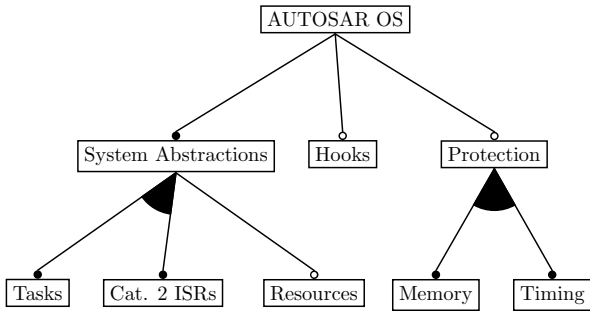
Figure 2: A part of the feature diagram for an AUTOSAR OS system, presenting some of the identified *external concerns*. Hooks and protection facilities are optional (empty circle), whereas system abstractions are mandatory (filled circle). A filled arc means that *at least one* of the features needs to be selected.

synchronization of the kernel is completely missing although vital if kernel state is accessed both from within tasks and asynchronous interrupt handlers. Those concerns typically affect many parts and other concerns in the system; they are therefore to be identified as early as possible so that they can be taken into account when analyzing and preparing for the design. Furthermore, it is often those concerns that influence the so-called *non-functional properties* of a system, such as its performance or robustness. Hence, internal concerns are to be determined and added to the concerns list by the skilled systems engineer (artifact B in Figure 1).

2.2 Step 2: Impact Analysis

When all concerns of the target system have been identified (artifact C and to a certain extent artifact D in Figure 1), their impact on the system software to be built and its components can be analyzed (step 2 in Figure 1). System software like AUTOSAR OS often possesses

1. *system services*, manifested in its accessible API that is stated in the specification if available,

2. *state* to be held by the kernel to be able to implement the system services,

3. and services or important *state transitions* internal to the system, which cannot be accessed directly by the application programmer.

These are the three groups that have to be considered when investigating the impact of each concern that was previously identified. The first two items can be detailed by taking a look at the AUTOSAR OS specification; there is a comprehensive API definition consisting of the services to be implemented. Some of these services can possibly be integrated in service *groups*, being influenced by all concerns in the same way. When investigating the impact of the AUTOSAR OS concerns on the system, it becomes clear that each system service and its corresponding API is introduced by exactly one concern; hence, in our case, the mapping of service introductions and concerns is a 1:1 relationship. (In other systems, a service may be introduced if *either one* of several concerns is present in a target configuration, for instance.) Besides the possibility that services can be

introduced to a system's implementation and its API, many other concerns conceptually *extend* an existing system service by providing additional behavior *pre-service* or *post-service*. These results can be depicted in a *matrix cross-cut table* (see Section 2.3).

Furthermore, through their parameter definitions, the given API services hint at the state that is to be held and managed by an AUTOSAR OS system in the form of instantiable system entities (e.g., tasks or resources). Some specifications even detail these system entity types by providing a concrete layout to be implemented; AUTOSAR leaves that to the implementing party, though. The state needed by the system is composed of state that needs to be modified by the kernel at run time as well as constant configuration information, which is provided statically before compile time. Some of the concerns previously identified introduce a specific system data type; this is always done by the one concern that also introduces the notion of that object type *conceptually* (e.g., the task management concern introduces a task object type). Other concerns merely *extend* system data types by single data members needed by these concerns; this extension can itself comprise dynamically modified state or static configuration information (e.g., a task's state is dynamically modified while its priority is basically configuration information). Another class of impact a concern can have is to extend the value range of member data types in OS object types; for instance, the event management concern introduces an additional *waiting* state a task can reside in.

Hence, a comprehensive list of system services and system entity types can be compiled when taking the specification as a starting point, eventually providing a guideline to check for impacts of a specific concern.

The third item group, however—important internal points in the system software—can only be established during the concern evaluation itself. Therefore each concern has to be checked for whether it has an influence on a point in the system not covered by the two groups considered before; that is, points not at all visible in the system interface. Further functionality can then be given to these points by the involved concerns. By making these influences explicit, exposing the corresponding join points in the design is facilitated, effectively enabling those features designed as aspects to give advice to them. An example for such a point in AUTOSAR OS is the point of dispatching to another AUTOSAR task, which can happen detached from an API call if the system is configured to realize preemptive scheduling. This point is heavily cross-cut by many concerns (not all concerns are depicted in Figure 3). Anticipating these internal points in advance takes some degree of experience in systems engineering and strongly depends on the kind of domain the system is being built for. It also involves partly thinking about the system *design* already; see Section 3 for a discussion of that fact.

2.3 Output E: Matrix Cross-Cut Table

The first result artifact to be provided by the analysis in order to facilitate the subsequent design process is a comprehensive *matrix cross-cut table*. This kind of diagram aims at summarizing the concerns and their impact on the system and at revealing critical join points. For an excerpt of the table that we prepared for AUTOSAR OS, see Figure 3; it shows a few of the concerns and impact points of the three groups detailed in Section 2.2 (system services, state,

		Configurable Concerns							
		Tasks	ISRs Category 1	ISRs Category 2	Kernel Sync	Timing Protection	Invalid Parameters	Interrupts Disabled	Hooks
1	GetActApplicationMode()							◑	
	StartOS()						◐	◑	
	ShutdownOS()							◑	
	ActivateTask()	⊕					◐	◑	
	TerminateTask()	⊕						◑	
	ChainTask()	⊕					◐	◑	
	Schedule()	⊕						◑	
	GetTaskID()	⊕						◑	
	GetTaskState()	⊕					◐	◑	
	EnableAllInterrupts()		⊕			◐			
	DisableAllInterrupts()		⊕			◑			
	ResumeAllInterrupts()		⊕			◐			
	SuspendAllInterrupts()		⊕			◑			
	ResumeOSInterrupts()			⊕		◐			
	SuspendOSInterrupts()			⊕		◑			
	GetISRID()			⊕				◑	
	DisableInterruptSource()			⊕			◐	◑	
	EnableInterruptSource()			⊕			◐	◑	
2	Task	⊕				⊛			
	ISR Category 2			⊕		⊛			
3	Cat. 2 ISR Execution					●			
	System Start-Up	◑							◑
	Task Switch					●			●
	Appl.–Kernel Transition				◐				
	Kernel–Appl. Transition				◑				

Figure 3: An excerpt of the matrix cross-cut table for an AUTOSAR OS system, showing the impact of configurable concerns on AUTOSAR system services (multirow 1), state held by the system (multirow 2), and system-internal points (multirow 3); ⊕ = introduction of service/state, ⊛ = modification/extension of an existing type, ◑ = modification after service execution / OS-internal pointcut, ◐ = modification before service execution / OS-internal pointcut, ● = modification before *and* after service execution / OS-internal pointcut.

internal transitions), and how the concerns affect each other. Consider, for example, the task management concerns: It introduces the implementation of six task-related services to the system and its API, as well as internal state to be held for each task. Furthermore, the system-internal start-up point is given additional functionality, which stems from the AUTOSAR feature of autostarted tasks. The timing protection concern, however, does not introduce any services at all; instead, it adds additional behavior before or after six of the listed services. Additionally, the state held per task and category 2 ISR is extended by timing-specific information. Finally, this concern needs to be informed whenever a new control flow is dispatched (category 2 ISR or task).

A well-prepared matrix cross-cut table can be analyzed in multiple dimensions. A *horizontal analysis*, for instance, focuses on the system services and how they are introduced or modified. Hence, it can be seen if a service is introduced by a single concern (as is the case with all AUTOSAR services; i.e., there is exactly one ⊕ per row) or when either one of a group of features is present in a given system configuration. Moreover, heavily cross-cut services can be easily spotted

and targeted for a careful design of those exact services and the join points they provide. Similar points apply to the investigated OS object types and system-internal points; those of interest to multiple stakeholders can be designed accordingly. For instance, both the timing protection concern and the hooks concern (amongst yet other concerns) are interested when a task switch takes place in the system. That is why this point has to be given extra thought; for example, which feature has to be activated first, since this has an impact on the time budget for the current task managed by the timing protection concern. Furthermore, since the cross-cut table clearly lists consumers of such internal state transitions, it becomes clear that these join points have to be exposed for the aspects implementing the concerns to be able to attach to. This is not always straight-forward; consider, for instance, that *after advice* given to the internal `dispatch()` function (implementing the task switch) does not work as expected since that function conceptually returns in the context of another control flow.

Vertical analyses, however, are oriented at the different concerns and focus on the impact from their perspective. For instance, simple "verbal pointcuts" (i.e., simple phrases exactly describing a target set of services) can be identified for those concerns homogeneously cross-cutting many services and having a similar impact on them. These verbal pointcuts can then be used to build actual pointcuts formulated in the used aspect language later on. The AUTOSAR concern checking for invalid parameters, for instance, can be described as influencing "all services with an OS object data type as a parameter", whereas the concern checking for disabled interrupts affects "all services except the interrupt services". This subanalysis furthermore provides an indication for the aspect-oriented mapping in the later design and implementation stages by clearly summarizing the impact of each concern on services, state, and system-internal points.

2.4 Output F: Concern Hierarchy

The second type of diagram developed during the concern impact analysis is a *concern hierarchy* diagram; it is supposed to depict the dependencies and influences among the concerns to be implemented (see Figure 4). Its goal is to make the interaction of the involved concerns explicit in order to be able to respect that in the system's design and implementation. The concern hierarchy also needs to include the kernel-internal concerns, which are otherwise often neglected in the design process, leading to inappropriate or inflexible designs.

The concern hierarchy diagram integrates the knowledge gathered in the matrix cross-cut table in order to go a step further and arrange the concerns according to their "uses" and "influences" relationships. A "uses" relationship indicates a dependency of a concern on another one, whereas "influences" can be seen as a rather loose and optional coupling between two concerns; it effectively means that if one or all of the target components do not exist, this does *not* constitute an error. This is the kind of flexibility that allows otherwise tangled concerns to be well separated into distinct design artifacts. Consider, for instance, Figure 4 and the concern checking for out-of-range values, which only applies to alarm and schedule table services. Hence, the concern *influences* the corresponding alarm and schedule table management concern, but is not *dependent* on it. If that target concern is not present in the configured system, the check concern
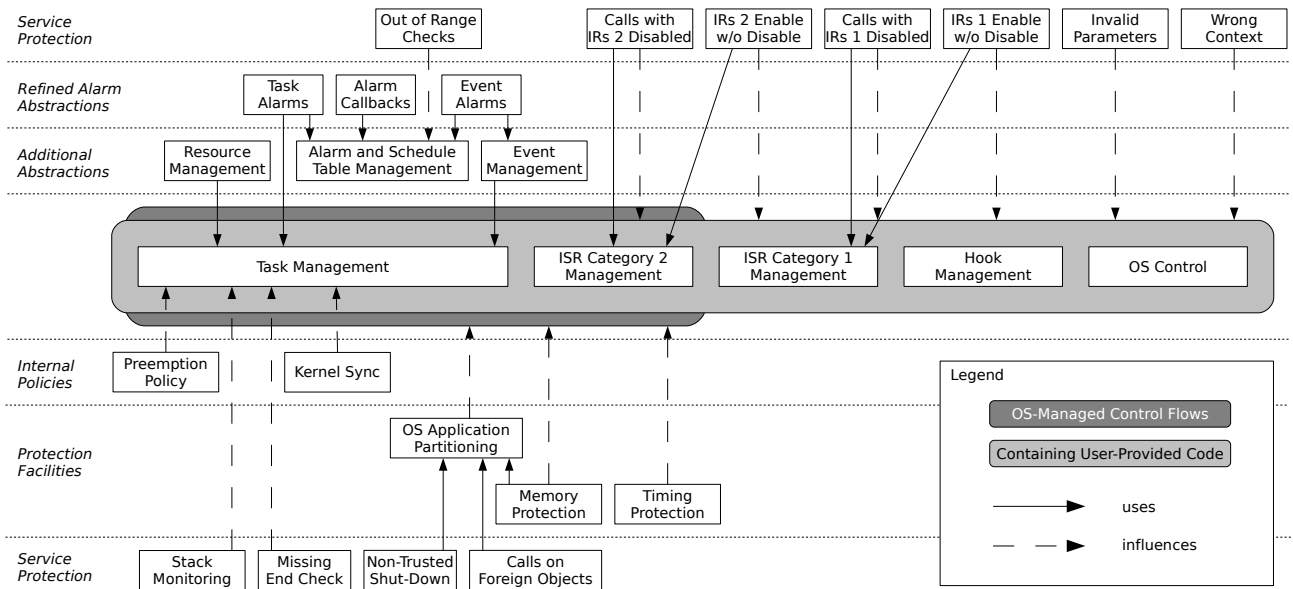
Figure 4: The concern hierarchy diagram for an AUTOSAR OS system, depicting the dependencies and influences involved. (In this picture, the Service Protection layer is divided into two layers for reasons of graphical feasibility.)

merely does not have any join points to apply its functionality to, but nevertheless fullfills its requirement not to allow any out-of-range values.

Besides depicting relationships, the concern hierarchy diagram is used to group concerns into groups that are collectively influenced or used by other concerns. If those groupings can be determined by some commonality, a step towards a pointcut for the influencing aspect's advice is taken that can be used in the design. Consider, again, the AUTOSAR OS concern hierarchy in Figure 4. Most of the service protection concerns have an influence on as many as five other concerns. These five concerns can be denominated by their common property "containing user-provided code"; it is exactly those parts of the system that need to be checked by the service protection concerns for potentially illegal circumstances.

All in all, the matrix cross-cut table and the concern hierarchy diagram build a basis for the successive aspect-aware design and the UML diagrams depicting it by clarifying the concerns in a system software and their relationships. The designer is therefore enabled to develop a comprehensive design that respects all concerns and concern interactions.

3. DISCUSSION

A noteworthy side effect of the third point of the analysis (identifying the concern influence on system-internal points) is that thinking about important internal state transitions partly implicates already thinking about the design and implementation of the system; this is usually deferred until *after* the analysis. In our opinion, this is necessary and even essential for a successful aspect-aware design since such a design requires to capture and expose important join points to be tackled by aspect advice. This, however, is only possible by partly bringing forward some of the design decisions. That is why the analysis is not performed solely from the interface perspective, but also with a basic design in mind.

Before designing the kernel of our CiAO operating system, we used the *concern impact analysis* on the AUTOSAR OS specification to provide the input for the design decisions to be taken. The two output diagrams (i.e., the matrix cross-cut table and the concern hierarchy) were generally very helpful since they give a concise and untangled overview of the system with its concerns and their influences. However, it turned out that some points had to be implemented differently than was anticipated in the analysis. Consider, for instance, the internal pointcut candidate "task switch" that was identified to be needed by several concerns (two of those are shown in Figure 3). This denotation is too far away from the implementation, though, since a task switch can refer to an *internal context switch* or an *external (AUTOSAR) task switch*, which are two related but different things. Hence, this needed to be distinguished further in order to determine which concerns need which of the two task switch notions. Thus, again, some degree of implementation knowledge is already needed in the analysis.

Since we both designed our kernel in an aspect-aware *and* AUTOSAR-like way, we gained experience in how well those two worlds can be combined. Using our *concern impact analysis* on the AUTOSAR OS specification enabled us to reach a very good separation of concerns already at the analysis level, which, thanks to AOP, could mostly be transferred to the design and implementation stages of the system. Several AUTOSAR concerns, especially from the service protection class, are homogeneously cross-cutting, which can be well tackled by means of a single piece of advice. Other concerns feature a very straight-forward representation in the design since AOP concepts like flexible pointcut matching allow those concerns to be expressed very directly and explicitly.

Though we developed CIA in the context of building an embedded operating system, we are positive that the general process is helpful and applicable to other domains of

configurable software as well; CIA helps to clarify inter-concern dependencies in order to develop a sophisticated aspect-oriented design eventually. However, regular software might not pose that many difficulties with hidden internal concerns as does system software, and, hence, CIA might not contribute as much value as in the system software domain.

4. RELATED WORK

There is a whole sub-branch of AOSD research that is dedicated to the topic of early aspects in requirements engineering. The importance of identifying cross-cutting concerns at this stage in the software development process has long been known; however, most of the corresponding papers argue that this is needed to reach increased traceability and therefore facilitate concern evolvability [12]. CIA, however, was mainly developed to allow for a good aspect-aware design of a system. Furthermore, CIA targets system software, whereas other research targets software in general and does not refer to and respect common system software concerns [4].

The need to weaken the *obliviousness paradigm* in the contruction of complex software has also been recognized by other research groups. Sullivan et al., for instance, found that oblivous software designs extended by aspects later leave important abstractions implicit in program details [15]; therefore, they propose *design structure matrices* (DSMs [3]) to depict concern dependencies. DSMs, however, depict the dependencies among different types of artifacts, including concerns (basic and cross-cutting), interfaces, and implementations; CIA and cross-cut matrices focus on the investigation of concern impact on concrete system implementation artifacts only.

Other related work identified the implicit superimposition of concern behavior at the same join point as being one of the main challenges of AOP [7]. Undesired emerging behavior and conflicts may result; therefore, Durr et al. propose a technique to detect those conflicts, which is applicable to arbitrary pieces of software [7]. CIA, however, was derived from a concrete system software project and proposes concrete analysis steps and output diagrams for this domain.

Other research groups also identified the need to expose AOP join points, especially in the context of system software engineering. If the design does not expose those to begin with, the necessary amount of refactoring can become intolerable [13]. Another approach is to develop domain-specific aspect languages for the system software domain [5]. In either case, a clear idea of the concern interactions is needed to develop the design of the system itself or the one of the aspect weaver and its features, respectively.

5. SUMMARY AND CONCLUSIONS

System software provides no business value on its own. Its sole purpose is to ease the development and integration of applications. Hence, system software should be tailorable to exactly the amount of abstraction and functionality needed by the intended application.

To provide optimal tailorability, the features of a piece of system software have to be decomposed and designed in a very fine-grained way. This is especially challenging as the functional concerns of system software tend to interact with each other via internal, often non-functional system concerns in non-trivial ways.

Based on a system's specification and the engineer's exper-

tise, the CIA method reveals these "hidden players" and their interactions early in the software design. The resulting explicit representation of all system concerns, their join points and dependencies facilitates a fine-grained, aspect-aware design of system software, such as AUTOSAR OS.

6. ACKNOWLEDGMENTS

7. REFERENCES

[1] AUTOSAR. Specification of operating system (version 2.0.1). Technical report, Automotive Open System Architecture GbR, June 2006.

[2] AUTOSAR homepage. http://www.autosar.org/.

[3] C. Y. Baldwin and K. B. Clark. *Design Rules: The Power of Modularity*. MIT Press, 2000.

[4] I. Brito and A. Moreira. Towards a composition process for aspect-oriented requirements. In *Aspect-Oriented Req. Engineering and Arch. Design W'shop*, 2003.

[5] Y. Coady, C. Gibbs, M. Haupt, J. Vitek, and H. Yamauchi. Towards a domain-specific aspect language for virtual machines. In *1st W'shop on Domain-Specific Aspect Languages (DSAL)*, 2006.

[6] K. Czarnecki and U. W. Eisenecker. *Generative Programming. Methods, Tools and Applications.* 2000.

[7] P. Durr, T. Staijen, L. Bergmans, and M. Aksit. Reasoning about semantic conflicts between aspects. In *2nd European Interactive Workshop on Aspects in Software (EIWAS '05)*, 2005.

[8] D. Lohmann, F. Scheler, R. Tartler, O. Spinczyk, and W. Schröder-Preikschat. A quantitative analysis of aspects in the eCos kernel. In *EuroSys '06*, pages 191–204, Apr. 2006.

[9] D. Lohmann, J. Streicher, W. Hofer, O. Spinczyk, and W. Schröder-Preikschat. Configurable memory protection by aspects. In *PLOS '07*, Oct. 2007.

[10] D. Lohmann, J. Streicher, O. Spinczyk, and W. Schröder-Preikschat. Interrupt synchronization in the CiAO operating system. In *AOSD-ACP4IS '07*, New York, NY, USA, 2007.

[11] D. L. Parnas. Designing software for ease of extension and contraction. *IEEE Trans. Softw. Eng.*, SE-5(2):128–138, 1979.

[12] L. Rosenhainer. Identifying crosscutting concerns in requirements specifications. In *Proceedings of the Aspect-Oriented Requirements Engineering and Architecture Design Workshop*, 2004.

[13] J. Siadat, R. J. Walker, and C. Kiddle. Optimization aspects in network simulation. In *AOSD '06*, pages 122–133, Mar. 2006.

[14] O. Spinczyk and D. Lohmann. Using AOP to develop architecture-neutral operating system components. In *11th SIGOPS Europ. W'shop '04*, pages 188–192, 2004.

[15] K. Sullivan, W. G. Griswold, Y. Song, Y. Cai, M. Shonle, N. Tewari, and H. Rajan. Information hiding interfaces for aspect-oriented design. In *10th European Software Engineering Conference*, pages 166–175, 2005.