An Adaptive Control Model for Non-Functional Feature Interactions

Christian Prehofer
Fraunhofer ESK & LMU München
Christian.Prehofer@esk.fraunhofer.de

***Abstract*:** Many systems, especially distributed embedded systems, have very strong emphasis on non-functional properties, which are often cross-cutting and difficult to capture in a modular way. Here, we consider non-functional feature interactions, which occur if two features show unexpected behavior regarding non-functional properties. The goal is to handle non-functional properties and interactions in a modular and flexible way on a separate control layer. On this control layer, we can adapt control components to different feature interactions. We use statecharts to describe control models and use statechart refinement to make interactions explicit. We present our approach by two examples with several non-functional feature interactions and argue that the control layer can address these. The main advantages are modular control of non-functional properties and explicit modeling of non-functional feature interactions on a separate control layer.

Keywords: non-functional properties, feature interactions, adaptive systems, statecharts, modularity, modeling

I. INTRODUCTION

For many systems, non-functional requirements like performance and responsiveness are quite important and there is considerable work [5] to capture these. We focus here on non-functional interactions between features. A feature is unit of functionality [9][10], similar to the notion of a service in [15]. Feature interactions are a very broadly defined phenomenon: two features interact if these two features exhibit unexpected behavior which does not occur when features are used in isolation. The problem of feature interactions originates from the telecommunications area [9] but has been increasingly investigated in other areas [1][2][3].

There exist many classifications and detection mechanisms of feature interactions [9]. Typical interactions are on the functional level, e.g. conflicting requirements, implementation or indirect interactions on the external context. Non-functional feature interactions occur if two features interact on their non-functional properties. For instance, two features may have specific performance or energy efficiency requirements which are violated when both features are used jointly. This is also called a non-functional dependency.

Many systems, especially distributed embedded systems, have very strong emphasis on non-functional properties like performance, energy efficiency and reliability [5]. In these systems, we typically have several components which run independently (in parallel) and implement services with several features.

The main problem addressed here is how to control non-functional properties and interactions while preserving component modularity. Addressing non-functional properties is difficult as they typically concern system-wide properties like performance or energy consumption. The modularity problem of non-functional properties was observed by several other authors. E.g. in [17] such non-functional properties in operating systems are described as emergent and cross-cutting, which must be handled on a system-wide view. Similarly, in the area of web services [18] discusses the problem of non-functional properties which lead to scattered and tangled code.

Here, we consider modularity for non-functional feature interactions. Handling such interactions needs to consider the selected features, observed interactions and overall non-functional requirements. Hence, resolving these issues means to modify specific system or component behavior, depending on which other features are active. Modifications on code and possibly requirements are needed for resolving feature interactions which are hard to track and often violate modularity. For large systems with hundreds of features and interactions, this often leads to cluttered code and loss of modularity. For systems with variability, many special cases have to be considered for features which may not be present in a particular configuration, further cluttering the code.

Here, we propose a hierarchical control model, which handles the non-functional properties and interactions on a separate control layer. On this control layer, we can adapt control components to different feature and component configurations. We propose a system model for these layers and use statecharts as control models. The main goals are to handle these interactions on the control layer and to make these interactions explicit. In this approach, feature interactions are addressed by refining statecharts, depending on the current configuration and non-functional requirements. This can be used for both variability at design time as well as for adaptation at run time.

Using a separate control layer for handling non-functional requirements and interactions faces several challenges:

- For systems with strong requirements on performance and reactiveness (or soft real-time), the control layer and interaction handling shall not introduce overhead which could violate these.
- Feature interactions are often cross-cutting and affect several other components, making it difficult to handle these on a specific, single layer.
- Non-functional feature interactions often depend on the environment or overall system status and should only be

handled when needed. Examples are overload situations or power saving modes.

We will present a control layer which is decoupled from the critical path of component functionality and show how non-functional requirements and their interactions can be addressed in a modular and flexible way on this layer. By separating these concerns in a new layer, we can also validate these requirements in a simpler way, assuming that the control layer has sufficient and accurate data.

In the following, we discuss two examples, where we analyze and classify non-functional feature interactions. Then, we present our architecture, followed by two examples based on the examples below.

A. Signaling Example

In communication networks, signaling protocols such as SIP [16] are used to control the user sessions. A signaling server implementing such protocols is part of the end-to-end communication between the communication parties and receives call-setup, call-tear down and several other messages for handling sessions. In addition, the signaling server has to manage the local resources of the network, e.g. admit only calls if the network has resources for it. Typical requirements are both fast and responsive call setup as well as efficient resource usage.

The problem is that such signaling servers have to handle thousands of sessions and often also thousands of messages per second. The server has to handle overload, failure and emergency situations efficiently.

Due to these performance and overload constraints, we have a number of non-functional feature interactions. For instance, in case of an emergency call, other calls may have to be dropped if no resources are available. In case of failures in the network, affected calls have to be re-routed or terminated properly. Another challenge is overload protection, e.g. when many requests arrive, these should not impede efficient handling of existing calls. (E.g. call tear down should still be possible.) In many practical systems, the implementation of the base functionality of call control is less difficult than system control which has to resolve the above interactions.

B. Automotive Infotainment Example

Consider the case of an infotainment system in a car which mainly performs navigation, but also has a few important features such as a parking assistant which automatically parks the car in a lot. During this parking operation, the current situation is shown to the user, e.g. a graphical view of the position of the car wrt sidewalk and other cars. We assume that this feature is more critical than the navigation, as the driver needs real-time information in order to control the car in unforeseen circumstances.

A possible feature interaction is the following: If the navigation system is in the mode of route calculation, it consumes considerable system resources (CPU, memory, I/O) and other critical features like the parking assistant cannot perform as needed. This interaction may be resolved in several ways, e.g by adapting priorities or canceling / delaying some operations.

In addition, we may have different modes, say normal and energy efficieny mode. The latter is used when the (hybrid) car working on electrical energy only and energy optimization is important. Thus, we have mode management and depending on the mode, the interaction resolution is done differently.

In case of the energy mode, the infotainment system focuses on a single priority task. This means, the navigation route search is stopped during parking assistance and continued after completion of this. In case of normal mode, the CPU speed may be increased.

In addition, there may be safety critical features, e.g. a crash warning system to detect dangerous driving situations and display visual warnings in this case. For this, the system may have an emergency mode, which disables non-critical features to ensure optimal performance of other features.

II. SYSTEM ARCHITECTURE

In the following, we introduce our system architecture based on the illustration in Figure 1. The architecture consists of following three layers:

- Control layer, focuses on the control of non-functional properties.
- Component layer, executing the actual tasks. One component implements the core functionality of one or more features.
- Environment layer, representing the external environment of the system.

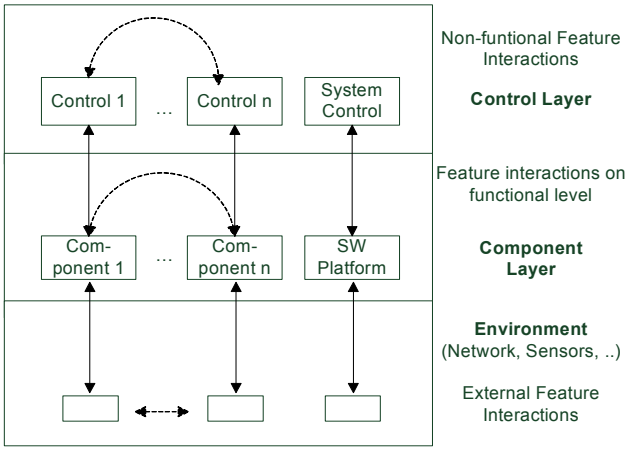

Figure 1: System Architecture

The main functional part is provided in the component layer. We assume here that the actual tasks of the system are executed by several (active) components, typically running in parallel on a specific software platform. The software platform consists of the operating system and optional middleware. The system may be distributed over several network nodes. Note that the arrows indicate the feature interactions and control dependencies. Clearly, there are

other interactions as expected, e.g. between the components and the SW Platform.

A. An Asynchronous Control Layer

The control layer includes control components for each component in the component layer and system control components. The system control components typically model system-wide concerns or settings and also interact with the software platform. From the component layer there are interfaces to the control layer, on which the components send runtime notifications about their current internal state. These are sent asynchronously, hence the component layer does not have to wait for acknowledgements and there is minimal overhead. To reduce overhead, the components only report changes in their state as needed by the control layer. Also, the control layer is not directly interacting with the environment as this may interfere with the system performance and cause other interactions. The intention here is to minimize the performance impact of the control layer. On the other hand, this means that real-time critical control must be handled on the component layer.

The actual control actions by the control layer can be implemented indirectly by control interfaces to the components or by interactions with the system control. Also, component control can be aggregated and one control component can handle several components.

As a generic example consider Figure 2, which shows two general system modes, A and B. The SW platform monitors the system and triggers the state transitions in the control layer. For component n, we similarly have a control component with two states.

Similar control architectures have been considered in different lines of research, e.g. [6]. Here, our goal is to show how non-functional properties can be handled on this layer.

B. Feature Interactions

In our system architecture, we can now differentiate three kinds of feature interactions as follows:

- Component-level interactions are typically functional interactions where the purpose of a component interacts with others in unforeseen ways. E.g. both need the same internal resources or conflict on their functional requirements.
- External interactions (see also [1]) are interactions which occur in the environment caused by two features. They are not visible within the system and are caused indirectly in this sense.
- Non-functional feature-interactions are similarly caused by the effect of two features on non-functional properties and are typically indirect in nature.

Typical non-functional feature interactions are resource usage patterns which cause performance conflicts or violate energy efficiency and security requirements. Other cases are reliability, which often means recovery from failures or other unwanted behavior.

C. Non-Functional Feature Interactions and Refinement

In this section, we discuss how to model non-functional feature interactions in our system architecture. The goal is to handle non-functional feature interactions on the control layer in a modular and flexible way by refining statecharts. The problem is that these interactions need to be handled by special cases in the code. Such a piece of code is only needed if both features are present and is often cluttering the code, as discussed in Section I. Furthermore, this does not scale well for large systems with many optional components and many interactions.

We explain our use of statechart diagrams based on the example in Figure 2. Each statechart implements a control component. The control components interact with the component layer and among themselves. We distinguish three kinds of transitions and actions, for which we also use different markers.

- Normal transitions are triggered asynchronously by the controlled component on the component layer.
- Control transitions are caused by control events or depend on conditions within the control layer and are shown by bold arrows.
- Control actions as part of normal transitions, which trigger actions on the control layer or on the component layer. These are also shown in bold.

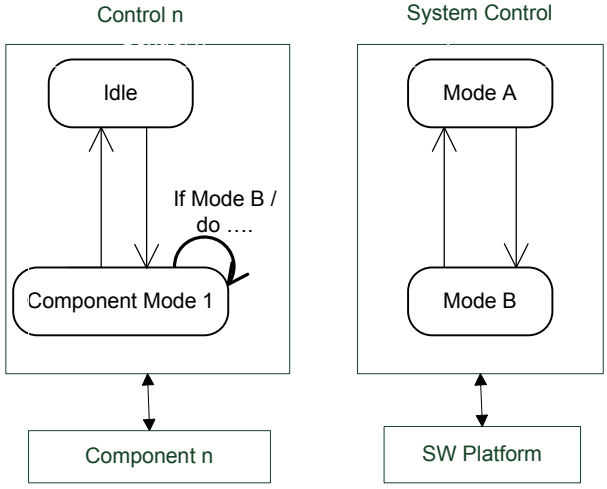

Figure 2: State Charts for System Control

Feature interactions can be handled by the last two categories which include communication between control components. In the example of Figure 2, Mode B may interact with Component Mode 1 and some special case needs to be handled. This is shown by an internal transition (without external actions) which depends on the state of another control component. Such feature interaction handling is only needed if the interaction features are present. By making interactions explicit in statecharts, it is easier to use the right control components or even to adapt control components at runtime

Control transitions may affect the controlled components directly or indirectly. This means that they may set parameters or control the SW Platform, e.g. by setting scheduling preference. Alternatively, direct commands may influence or even terminate/restart the controlled component.

Further, we aim to show that adding feature interaction handling is a behavioral refinement. Behavioral refinement means that the original behavior of a control component is preserved when adding such interaction handlers [10][15].

As an example, the bold transition in Figure 2 is caused by System Control being in Mode B, and may trigger specific actions not shown here for simplicity. Thus, the original behavior is preserved if the interaction handling is not invoked. This is a form of conditional refinement: if the special case of a feature interaction does not occur, the component behaves as before.

In the examples below, it is easy to see that the additional control actions and transitions for feature interaction handling are a semantic refinement of control statechart. As in the case above, only additional behavior is added, while the original is not modified. A formal treatment is presented in [10][15], which goes beyond the scope of this paper.

Statecharts have been considered for system control and adaptive systems, e.g. [7][8] model systems and their reconfiguration or adaptation via statecharts. The main differences are that we use multiple control components which control different parts and aspects of the system. In this way, non-functional feature interactions can be handled on the control layer. Clearly, this only works if the feature interactions are properly modeled in the control components.

We will illustrate the above in the examples below. For detailed treatment of semantic statechart refinements and interactions we refer to [10] and [15].

III. EXAMPLES

In the following, we will discuss the above concepts in several examples.

A. Signaling Example

In this example, we consider a call control entity, which controls calls and has to handle large number of control messages per second in an efficient way as discussed above.

In our setting, we assume that each call has one component for its call control on the component layer and one control component on the control layer. The latter is modeled by a control statechart for each of the calls, as shown in Figure 3. This models the different stages of call setup as needed for the control layer. The transitions are triggered by the events in the component layer. The actions associated with the transitions are in this case only local to the control layer. Here, a global, synchronized variable "resources" monitors resource usage of all the calls.

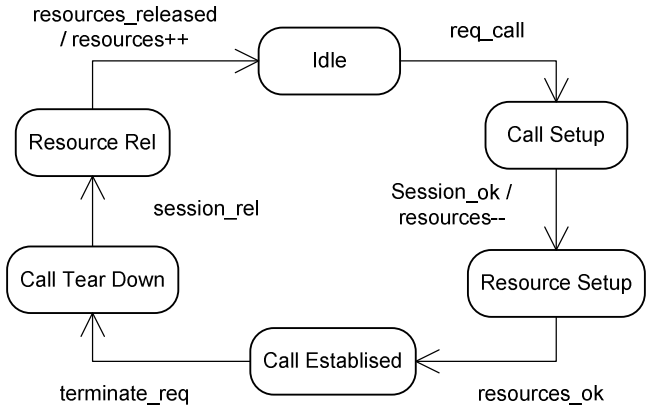

Figure 3: Call Signaling Statechart

In addition to these, we have several components which constitute the platform control. Figure 5 and Figure 6 model two independent modes of the system. In Figure 5 we model overload control. This means that there is a trigger from the Software Platform if the resources, here free phone lines, are below a specific limit. The control action here is to prioritize calls which are already in tear down mode. This can e.g. be implemented by adapting scheduling priorities.

Figure 6 models the case a of denial of service attack. This is again triggered by the Software Platform by specific events, here network load parameters. In this case, the control action is to setup specific filters for call setup. For

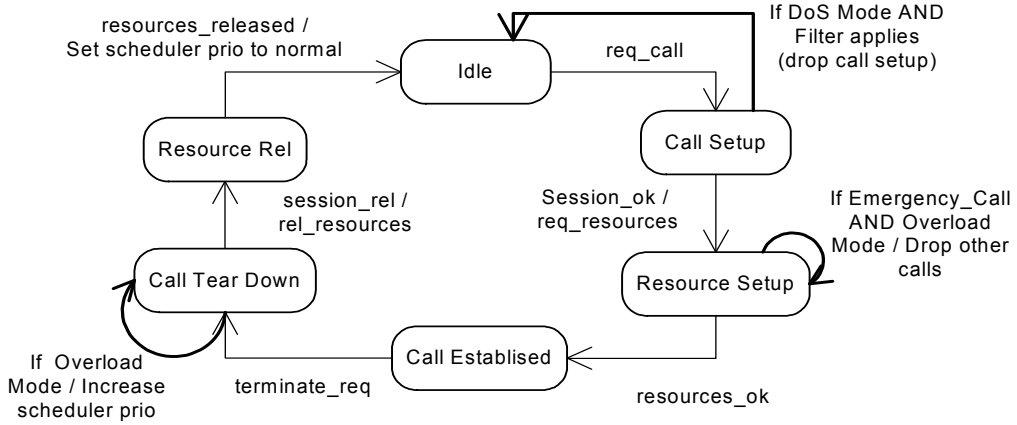

Figure 4: Call Control Example with Feature Interaction Handling

instance, only calls from specific regions may be accepted.

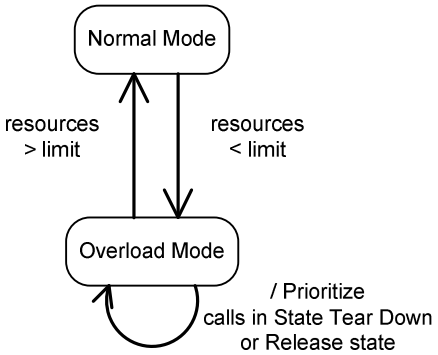

Figure 5: Overload Control Statechart

So far, we have three state charts as part of the control layer and have identified several interactions on non-functional properties. Now, we show how to adapt the call control statechart above to handle the feature interactions. While some interactions can be handled indirectly via the software platform, we show in Figure 4 a refinement of the above statechart where several interactions are considered. The filtering of calls as discussed above is implemented by a transition from Call Setup state which depends on DoS mode. Note that this implies a control action to the controlled component to drop this call and return to state Idle. (The actual implementation of dropping calls is not shown here for simplicity.) We assume here that control transitions must be taken and the controlee may not reach any other state in between.

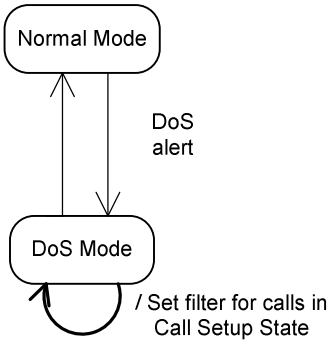

Figure 6: DoS Control Statechart

The setting of the scheduler priorities to resolve feature interactions is shown in this statechart as a additional transition in the Call Tear Down state. Note that this implementation is complementing the transition in the Overload Control statechart.

Another example is emergency calls, where we add another control transition. In case of such calls, we drop other calls if we are in an overload situation.

B. Automotive Navigation

Following the above example, we now model a control layer for the navigation case. We start with a control statechart for the basic navigation functionality, as shown in Figure 7. The control statechart models the different phases of a navigation system.

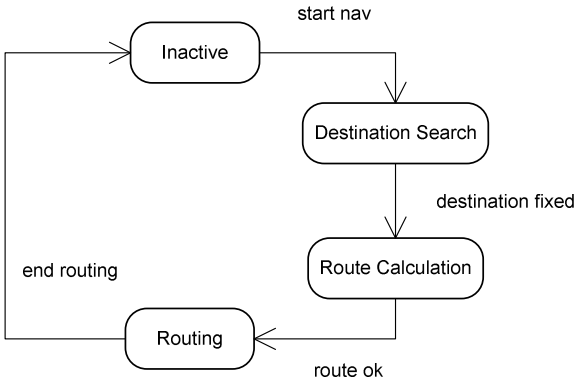

Figure 7: Navigation Control State Chart

Now, we model the control of a park assistant and then a global mode for energy efficiency, as shown in Figure 8 and Figure 9.

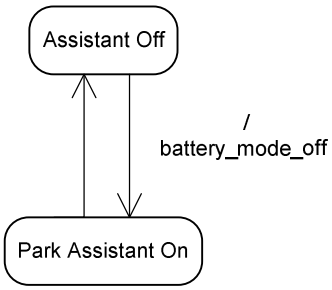

Figure 8: Control for Park Assistant

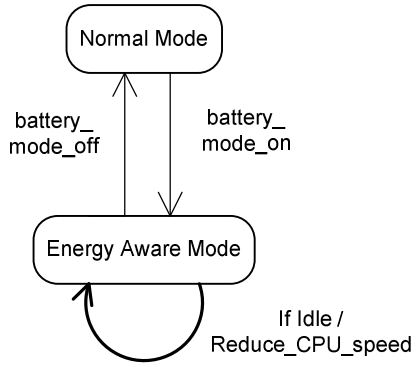

Figure 9: Energy Control Modes

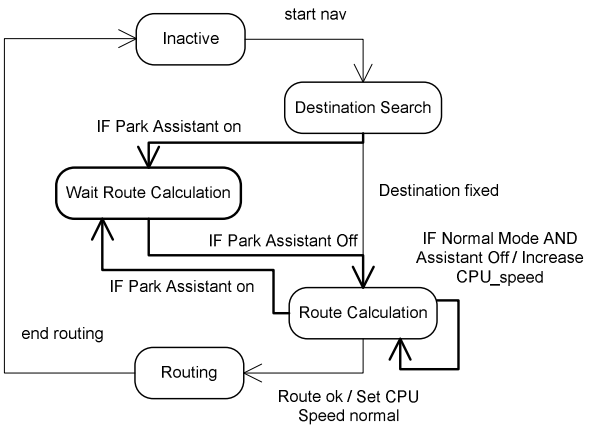

Figure 10: Navigation Control with Feature Interaction Handling

In Figure 10, we show an adapted statechart which takes care of several non-functional feature interactions. First, the navigation goes to a state Wait Route Calculation in case the park assistant is active before or during routing. This avoids resource conflicts as discussed above. Similarly, it increases CPU speed while route calculation in case of Normal Mode

IV. EXTENSIONS

In this section, we discuss several extensions of the model presented above.

We have covered several cases of non-functional properties in the example above. One missing, important item is to handle failures in the component layer. For instance, a component may not respond or not send updates as expected. For this purpose, we can use timeout events to handle such cases. An example is shown Figure 11, where a control transition is triggered by a timeout event or by process control of non-responding processes.

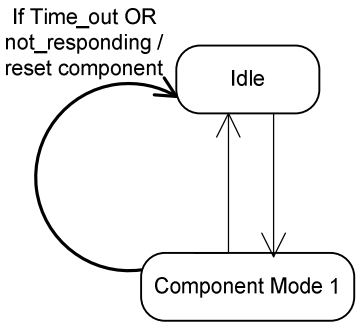

Figure 11: Failure Handling

Another interesting extension is the explicit modeling of time. Similar to resources, we can model the effort of transitions also abstractly on this layer. In this way, we can monitor time progress and for instance stop certain actions if some timing constraints have been violated. There exist also timed automata models, which can be used for this [12]. However, hard timing constraints are better implemented on the component layer, as the control layer is asynchronous and cannot react in real time.

An important property of statecharts is the opportunity of model checking, e.g. see [12]. In this way, we can validate properties of the control layer and possibly detect non-functional feature interactions if the model is comprehensive enough.

V. DISCUSSION AND RELATED WORK

We have shown that many non-functional feature interactions can be modeled in our layered architecture by control modes and refinement of control components (statecharts). The main assumption is that these non-functional violations can be observed and timely addressed by a control layer which runs independently of the actual component layer. Hence it is not on the critical path of the system reactions, which often have timing constraints. Typical examples are resource limitations, system wide configurations and modes as well as failures and reliability issues. Such properties can be observed by control events and the control layer can react like an independent control loop on the overall system state.

We decouple feature interaction handling from the time-critical interaction of the software components, which improves modularity. Also, the control layer overhead is hence minimized by keeping the layer asynchronous, i.e. the controlled components inform the control layer, but do not have to wait for notification messages to return.

The main, important assumption of this approach is that the control layer is receiving correct and timely information. For instance, if status information is not transmitted due to software bugs, the control layer cannot act appropriately and may have to abort the component.

There exist several prior works using control layer concepts from self-adaptive systems [6], model at runtime [11], and autonomic computing and self-organized, organic systems [13]. Other works also use statecharts for modeling control, e.g. [7] [8], but do not focus on non-functional properties. The main difference here is that we aim at a modular control layer, were each component has a separate control component. In this way we can make non-functional properties and their feature-interactions explicit. Other work such as [17] and [19] focuses on modularity regarding the specification of non-functional properties, but do not consider feature interactions on this level.

For modeling non-functional properties or aspects, there is work to model concerns or aspects separately or on separate control layers [14]. The problem is that the concerns may again interact and this is then difficult to relate to the components. Here, we model system wide concerns as global modes, and adapt the component controllers accordingly.

VI. CONCLUSIONS

We have shown a control architecture where non-functional properties are monitored and resolved by control components. The control layer is modular in the sense that each component has a control component and non-functional feature interactions are modeled by refinements of the

control components. We use modes to represent system wide concerns or settings, which then influence the software platform and/or other component controllers. Non-functional feature interactions are represented explicitly by refinements on the control components, represented as statecharts. This modularity enables adaptation of the system, both regarding the system components as well as non-functional requirements. Statecharts can be formally composed and refined in a property preserving way as shown in [10] or in [15], which goes beyond the scope of this paper.

In summary, we have shown that many non-functional feature interactions can be modeled in a modular way on a control layer by control modes and statechart refinements. This modular model makes it easier to track such interactions and to adapt the control layer in case of changes, both regarding requirements and system features.

We also argue that many non-functional properties, which are often cross-cutting and cover the whole system, can be handled on separate control layer. While control layers are used for the purpose of self-management or adapation [6], our classification of feature iteractions gives new insight which properties can be handled on the control layer.

Our work also enables validation and verification of non-functional properties on the control layer, which is left for future work. This assumes that the control model is sufficiently expressive and has correct and complete information

REFERENCES

[1] A. L. Juarez Dominguez..Feature Interaction Detection in the Automotive Domain. In Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering (ASE '08). IEEE Computer Society, Washington, DC, USA, 521-524. DOI=10.1109/ASE.2008.97

[2] Y. Liu, R. Meier, Feature Interaction in Pervasive Computing Systems, in First International DisCoTec Workshop on Context-aware Adaptation Mechanisms for Pervasive and Ubiquitous Services (CAMPUS 2008), Oslo, Norway, University of Oslo, Department of Informatics, Technical Report no.374, 2008

[3] Michael Weiss, Babak Esfandiari, "On Feature Interactions among Web Services," Web Services, IEEE International Conference on, p. 88, IEEE International Conference on Web Services (ICWS'04), 2004

[4] A. Metzger. Feature interactions in embedded control systems. *Computer Networks*, 45(5):625–644, 2004

[5] M. Glinz, On Non-Functional Requirements," Requirements Engineering Conference, 2007. RE '07. 15th IEEE International , vol., no., pp.21-26, 15-19 Oct. 2007, doi: 10.1109/RE.2007.45

[6] B. Cheng et al, Software Engineering for Self-Adaptive Systems: A Research Roadmam, in Software Engineering for Self-Adaptive Systems, Springer 2009

[7] Li Tan; , "Model-Based Self-Adaptive Embedded Programs with Temporal Logic Specifications," *Quality Software, 2006. QSIC 2006. Sixth International Conference on* , vol., no., pp.151-158, 27-28 Oct. 2006

[8] Gabor Karsai, Akos Ledeczi, Janos Sztipanovits, Gabor Peceli, Gyula Simon, and Tamas Kovacshazy. 2001. An approach to self-adaptive software based on supervisory control. In *Proceedings of the 2nd international conference on Self-adaptive software: applications* (IWSAS'01), Robert Laddaga, Howie Shrobe, and Paul Robertson (Eds.). Springer-Verlag, Berlin, Heidelberg, 24-38.

[9] Muffy Calder, Mario Kolberg, Evan H. Magill, Stephan Reiff-Marganiec, Feature interaction: a critical review and considered forecast, Computer Networks, Volume 41, Issue 1, 15 2003

[10] C. Prehofer, Plug-and-play composition of features and feature interactions with statechart diagrams, Software and Systems Modeling, 2004, Springer-Verlag.

[11] D. Garlan, B. Schmerl, Using Architectural Models at Runtime: Research Challenges in Software Architecture, Springer LNCS, 2004

[12] UPPAAL for Component Based Real-Time Systems, www.uppaal.org/port

[13] G. Weiß, M. Zeller, D. Eilers, R. Knorr, Towards Self-Organisation in Automotive Embedded Systems. (The 6th International Conference on Autonomic and Trusted Computing, Brisbane Australia 7.-9. July 2009), New York : Springer 2009

[14] McKinley, P.K.; Sadjadi, S.M.; Kasten, E.P.; Cheng, B.H.C.; , Composing adaptive software, *Computer* , vol.37, no.7, pp. 56- 64, July 2004

[15] M. Broy, Multifunctional software systems: Structured modeling and specification of functional requirements, Science of Computer Programming, Volume 75, Issue 12, 1 December 2010

[16] SIP: Session Initiation Protocol, RFC 3261, http://tools.ietf.org/html/rfc3261

[17] G. Ortiz, J. Hernández und P. J. Clemente, How to Deal with Non-functional Properties in Web Service Development, Web Engineering 2005, LNCS 3579/2005, 111-128, DOI: 10.1007/11531371_15

[18] D. Lohmann, O. Spinczyk, and W. Schröder-Preikschat, On the Configuration of Non-Functional Properties in Operating System Product Lines, 4th AOSD Workshop on Aspects, Components, and Patterns for Infrastructure Software (ACP4IS 2005), March, 2005, Chicago, IL, USA

[19] O. Loques, A. Sztajnberg, R. Curty, and S. Ansaloni, A contract-based approach to describe and deploy non-functional adaptations in software architectures, J. Braz. Comp. Soc., 2004