SOFTWARE—PRACTICE AND EXPERIENCE
Softw. Pract. Exper. 2005; (in press)
Published online in Wiley InterScience (www.interscience.wiley.com). DOI: 10.1002/spe.699

Model-centric software architecture reconstruction

Christoph Stoermer[1,*,†], Anthony Rowe[2], Liam O'Brien[3] and Chris Verhoef[4]

[1]*Robert Bosch Corporation, Research and Technology Center, 2 NorthShore Center #320, Pittsburgh, PA 15212, U.S.A.*
[2]*Carnegie Mellon University, 5000 Forbes Ave, Pittsburgh, PA 15213, U.S.A.*
[3]*Software Engineering Institute, 4500 Fifth Avenue, Pittsburgh, PA 15213, U.S.A.*
[4]*Free University of Amsterdam, De Boelelaan 1081a, 1081 HV Amsterdam, The Netherlands*

SUMMARY

Much progress has been achieved in defining methods, techniques, and tools for software architecture reconstruction (SAR). However, less progress has been achieved in constructing reasoning frameworks from existing systems that support organizations in architecture analysis and design decisions. These reasoning frameworks are necessary, for example, to assemble existing components and deploy them in new system configurations. We propose a model-centric approach where this kind of reasoning is driven by the analysis of quality attribute scenarios. The scenarios and the related quality attribute models guide the SAR effort by focusing on the elicitation of model relevant artifacts. The approach further drives the model construction towards the analytical support of *What If* scenarios that explore responses stimulated by new requirements, such as new deployments of existing components. The paper provides two real-world case studies. The first case study introduces the model-centric reconstruction approach in the context of a large satellite tracking system. The second case study provides the construction of a time performance model for an existing embedded system in the automotive industry. The model allows us to perform cost-efficient predictions of component assemblies in new customer configurations.

KEY WORDS: architecture; LIN bus; models; performance; prediction; quality attributes; software architecture reconstruction; views

1. INTRODUCTION

Software Architecture Reconstruction (SAR) supports organizations in understanding and analyzing software in situations where, for example, existing:

*Correspondence to: Christoph Stoermer, Robert Bosch Corporation, Research and Technology Center, 2 NorthShore Center #320, Pittsburgh, PA 15212, U.S.A.
†E-mail: christoph.stoermer@rtc.bosch.com

Contract/grant sponsor: Robert Bosch Corporation
Contract/grant sponsor: Dutch Ministry of Economic Affairs; contract/grant number: SENTER-TSIT3018

Received 2 October 2004
Revised 27 April 2005
Accepted 28 April 2005

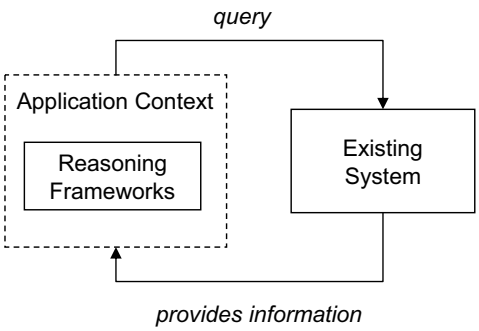

Figure 1. SAR context.

- software implementations have to conform to design descriptions;
- systems hit their architectural boundaries [1,2];
- software products have to evolve into product lines [3];
- software components have to be assembled and deployed in new system configurations, as described in the case study in Section 4.

The typical method to carry out a SAR effort is to extract information from various sources, provide collapsing strategies to aggregate implementation details into architectural abstractions and visualize them in architectural views. Much research has been done on the methods and techniques, resulting in a variety of approaches, such as manual reconstruction with tool support [4], query languages for writing patterns that automatically build aggregations [5], clustering [6], data mining [7], and the use of architecture description languages [8].

In the cases we are aware of, SAR is usually a facet in a much broader organizational context where software architectures play an important role in achieving particular business goals. The organization requires information from an existing system in order to support its decision-making processes. The support for business or design decisions is provided by reasoning frameworks which process and query for information about existing systems, as illustrated in Figure 1. Consequently, architecture reconstruction has to comprise the elicitation of information from the system as well as the provision of reasoning frameworks to support a particular application context in which decisions are made. The latter activity is the key for successful SAR applications. At the heart of SAR are the reasoning frameworks that embody models of existing systems, and not primarily the reconstruction techniques and tools themselves. Someone who carries out a SAR in a business context is foremost experienced in models and provides them in the context of an organization's decision support process.

The major ingredient of a reasoning framework is the model. In our approach, architecture relevant models are driven by quality attributes. Quality attributes are formulated in scenarios, typically growth scenarios for the system in SAR efforts. Growth scenarios may include:

- porting a system to a different operating system;
- increasing the throughput for transactions;
- exchanging legacy components with components-of-the-shelf.

The stimulus of each scenario is given by the particular application context; the constructed models provide the response. It is important to note that a single model does not provide responses to all scenarios. Typically, a model is tied to a particular quality attribute. For example, a portability model is not concerned about all components but is rather interested in components that have dependencies to the system platform. The automotive case study in this paper will introduce a model for time performance for a particular distributed embedded system. The performance model will not respond to other quality attribute scenarios, such as safety scenarios. In addition, the particular model instance of the case study is most likely not suitable for other types of systems. It is limited to the performance quality attribute and application context.

Our approach to architecture reconstruction centers the activities on the construction of models that provide response to business situations that are articulated in growth scenarios, such as new requirements for an existing product. With this, SAR is a means of constructing models obtained from existing systems that allow architectural understanding and analysis. Architectures are primarily manifested in models that support qualities. These models require program understanding of existing systems in order to be constructed and analyzed in the context of a reasoning framework.

Once a model is constructed from the existing system it provides a powerful base for additional prediction expertise, such as the support of *What If* scenarios. *What If* scenarios capture new stimuli of an organization. The model—with additional expertise—provides ways to predict the response. The automotive case study introduces an example time performance model-expert on top of a constructed model with formalized expertise to evaluate new design settings, such as time performance explorations in new deployment settings. With this, the organization is able to provide cost-efficient performance predictions before the real system is built.

The SAR reasoning frameworks, illustrated in Figure 1, consist of the following major parts:

- quality attribute scenarios;
- models;
- model experts;
- SAR methods and techniques.

Model experts are optional, because not every context requires support for *What If* scenarios, for example conformance measurements. This paper focuses on models and model experts in SAR. The interested reader is referred to [9] for more information about quality attribute scenarios and related elicitation techniques.

The remainder is organized as follows. In Section 2 we will relate our work to existing research. Then, we will illustrate the approach by using two case studies.

- In Section 3 we introduce the model-driven approach along a SAR project that we carried out for a satellite tracking agency.
- Section 4 provides an in-depth case study about time performance that we carried out in the automotive industry. This case study will further illustrate the usefulness of building a model expert in a commercial setting.

Finally, Section 5 summarizes the paper and outlines our future work.

2. RELATED WORK

From our experience we believe that the model-centric approach driven by quality attributes is the right mechanism for performing a SAR of existing systems. Indeed, Tahvildari *et al*. [10] outline an approach that uses non-functional requirements or quality attributes, such as performance and modifiability, to guide the reengineering process. Bengtsson and Bosch [11] outline a similar approach for reengineering based upon quality attribute scenarios that drive architecture transformation.

In our case we apply a quality attribute driven approach to architecture reconstruction and goal-based system understanding. The goal of the reconstruction is to provide information that will assist in the analysis of the quality attributes and provide further analysis for predictions in the form of *What If* scenarios.

In the past, several SAR efforts have related their work with more common/standard notations such as UML [12]. The goal of our approach is not to align architecture visualizations with mainstream notations, such as UML. The key to our approach is to enable architecture analysis of existing systems via a quality attribute driven approach. The analysis is motivated by the knowledge that software architectures are driven by business goals that incorporate quality attribute scenarios [9]. The work outlined in this paper describes progress that we have made since we started the work on Quality Attribute Driven Software Architecture Reconstruction (QADSAR) [13]. The work underpins the QADSAR approach by providing a more in-depth case study on a performance model. In addition, it goes beyond the QADSAR approach by illustrating the prediction capabilities of model experts for *What If* scenarios.

3. MODELS ESSENTIALLY

As outlined in the introduction, our approach to architecture reconstruction centers on the reconstruction activities for the construction of models that provide responses to particular business contexts articulated in quality attribute scenarios. We formulate our approach in the following way: SAR is a means to construct models obtained from existing systems that allow architectural understanding and analysis. The approach describes the activity (construction), reasoning (models), and the goal (understanding and analysis). The three parts will be explained using a real-world example, which we will refer to in subsequent sections.

Example. A Satellite Tracking Agency (STA) supports efforts to develop, acquire, and deploy satellite-tracking systems. In our example, the STA wanted to better understand the architecture of one of its legacy systems, the Satellite Tracking System (STS), so as to be able to port the system to a new environment. The STS consists of about 500KLOC, which is a mixture of C, C++, and Fortran that currently runs in a Silicon Graphics environment. The system has been in operation for many years and certain people know parts of the system for which they are responsible, though no one knows the architecture of the entire system, thus the need to apply architecture reconstruction techniques.

3.1. The goal

The starting point for this effort came from the business context of the STA organization. This context required the existing software to operate on a different platform. However, the software was not

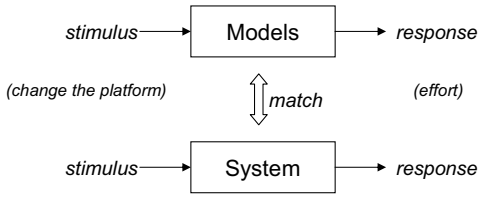

Figure 2. Model/system.

designed to support a portability scenario in the first place. At this point the STA has a couple of questions. For example, what is the effort to redesign the existing STS? Are there other complexity issues that require building parts of the STS from scratch? Is the whole system or only part of the system affected?

The impact of the business context required understanding and analysis: understanding the existing STS software in order to analyze the impact of the business goal. We conducted a SAR for the STA with the STS since no one was available who understood the architecture of the entire system and who could deal with the portability scenario.

Architecture reconstruction is not about effective partitioning algorithms, graphical visualization tools, and source code parsers. It is about answering business objectives. Stakeholders are interested in understanding or verifying architectures for re-documentation, architecture conformance, reuse investigations, product line migration efforts, trade-off analysis, comparisons, and reengineering. Each context has different objectives to accomplish that are primarily driven by the stakeholder interests. Consequently, not every possible architecture model is constructed but rather models that are important for the stakeholder context. Architecture reconstruction is therefore a goal-driven approach.

Although the application contexts and objectives differ, there is a common goal in reconstruction efforts: gaining of architecture understanding that enables architecture analysis. Understanding and analysis are complementary: understanding those system aspects that are relevant for the analysis of the business case impact. Architectural understanding of an existing system is not an isolated effort, but rather a necessary activity to enable an analysis or design effort justified by a business situation.

3.2. The reasoning

The construction of models is the core activity in every architecture reconstruction. Models are abstractions or conceptions of the existing system that closely match stimulus/response pairs as illustrated in Figure 2.

In our example, STA had to estimate the effort of exchanging the platform of an existing system. The stimulus is the demand to change the platform; the response is the effort to perform this change. Prior to a potential platform migration the architect has to estimate the effort and therefore needs a model that closely predicts reality. The technical part of the model will probably provide information such as the dependencies to the Silicon Graphics platform. The dependencies will have different weights in terms of cost of change. Dependencies of the current platform to other parts of the

system will uncover high or low probability of change and therefore have to be carefully analyzed. Models allow analyzing, exploring, and verification of architectures. They provide a foundation to evaluate change scenarios, performance predictions, and even project management considerations.

It is critical that models adequately reflect the reality of an existing system. Otherwise they may be used to generate the wrong conclusions. Representative candidates are mental models that are sufficiently adequate during the early phases of a software design but erode over the course of the product's lifetime due to growing complexity and people change.

In the STS example, portability is designed in terms of early documentations but the intention did not make it into the realization. A model that does not sufficiently match the existing system is a prominent reason to consider a reconstruction effort in the form of a conformance measurement.

3.3. The activity

Our approach intentionally uses the word 'construction' instead of 'reconstruction' in order to reflect the typical situation. In the majority of cases there is no explicit model initially but rather distributed mental models documented in box and arrow drawings. It is not the intent to reconstruct these mental models but rather looking at a snapshot of existing sources and construct models that probably did not already exist explicitly. In the STS case there was no existing model for portability. The reconstruction of an architecture for portability would be less helpful. However, a model to reason about a portability scenario would identify dependencies on the platform and be used to analyze the cost of change when it comes to replacing, adding, or emulating functionality. This analytical capability provides an important advantage over informal box and arrow drawings as often used in architectural views. We illustrate this by providing a few myths regarding architectural views in the following subsection.

3.4. The view myths

Software architectures are presented as views [14]. Views are a major vehicle to communicate architectures to stakeholders. Although they are of importance, their emphasis leads to side discussions that underestimate and undermine the purpose of reconstruction efforts. Here are three myths that are the result of such side discussions.

1. Architecture reconstruction is the generation of views in appropriate notations.
2. Views are models.
3. There is a set of common views for each class of systems.

The first myth is in fact the reduction of architecture reconstruction to a re-documentation activity. The re-documented views should accurately reflect the architecture of a system, which sometimes did not have an architecture documentation from the beginning. There are two implications to this opinion.

1. The re-documented views are the result of the constructed models. Views are difficult to analyze because the semantic context and traceability to existing facts is difficult to obtain from drawings. This is also valid for simplistic views, such as directory and file views. Even in this case it is frequently up to the 'viewer' to determine the mental model of why files are located in particular directories.

2. Secondly, understanding and analysis are complimentary tasks. It is not an efficient approach to re-document existing architectures without knowing the intention for which the resultant artifacts are used. In addition, the re-documented architecture can be eroded by the time a later analysis is required. In the majority of cases both tasks are therefore intertwined.

The second myth has at its core that views are a sufficient vehicle for architectural analysis. Views are not models; they are representations of models in a particular notation. For example, views can visualize performance aspects. However, they do not describe the performance algorithms or the reasoning that leads to the construction of a particular view. Moreover, they hardly reveal the consequences of adding or changing units of concurrency.

The third myth addresses the illusion that a particular set of views characterizes a larger system category such as a set for object-oriented systems, multi-language systems, etc. Each system has its own models. One reason is the different purposes and business goals systems are build for. The derived design decisions result in a wide variety of models. Therefore, it is hard to envision a 'one set of models/views fits all' approach or an automated model/view construction from existing systems without any previous built-in mechanisms to elicit the models and views.

Back to the example, this situation initially produced some confusion at the STA:

The STS is a classified system and access to the system and any information about it is tightly controlled. Consequently, the reconstruction of the real system was performed by the developers of the STA. The architectural views and the collapsing strategies were developed on a manipulated version of the information extracted from the system. For example, the developers extracted source artifacts and exchanged entity names with numbers. The collapsing strategies were applied by the developers on the real STS system to generate a set of architectural views. Although this time-consuming process produced a lot of overhead in effort and communication, it enabled STA to perform architecture reconstructions on further system versions by themselves.

Due to the classified nature of the system, the developers performing the reconstruction asked for a tool that would provide them with the—at this point in time—unknown architecture views. The STA was unaware of the effort and scope involved in the construction of the model to accomplish their particular goal.

Views are important to describe, understand, and communicate the architecture. In the process of obtaining these views from existing systems there is the cost-intensive effort to construct the underlying models. The cost is justified when the analysis requires a close match to architecture realities.

3.5. Models in practice

Despite the variety of models, there are recurring architectural parts that constitute models that have been discussed in the community over the past several years, such as patterns and their detection in existing systems, architecture styles, and quality attribute models. Further, there are a rich set of strategies and techniques available to collapse detailed source information into more abstract structures. Figure 3 provides an overview of the major ingredients that are typically required to construct models.

It is important to obtain an initial concept (hypothesis) for the model. The hypothesis guides the identification of:

- facts from the existing system;
- useful collapsing strategies;

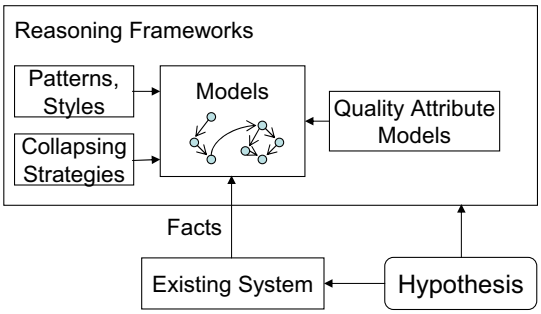

Figure 3. Model ingredients.

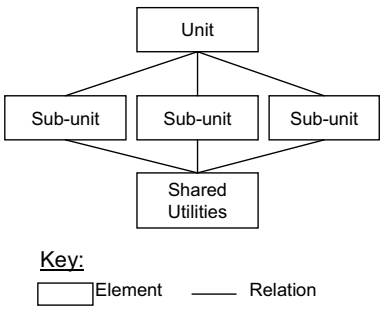

Figure 4. Initial concept.

- potential patterns;
- relevant quality attribute models.

Architecture reconstruction is hard to achieve without any initial idea about the architecture of the existing system.

Back to the example system:

The initial model was elicited in interviews with the architects and reading of existing documentation. As a result, we sketched the concept as illustrated in Figure 4. In this concept a unit relates to a set of sub-units that have shared utilities. The unit is more abstract than a sub-unit. For example, 'Weather' could be a unit which contains sub-units for different types of weather conditions.

The example started with the intention to port the STS to a new system. The initial concept has therefore to be transferred into a modifiability model to analyze the dependencies between the units and the platform specific routines. The process to create the model was performed in the following sequence.

1. Identify an entity/relation schema suitable in a multi-programming language environment (C/C++ and Fortran).
2. Extract the facts from existing sources.
3. Aggregate the facts into the unit concept of Figure 4 with suitable collapsing strategies. Isolate the platform-specific modules in a separate unit.
4. Navigate, verify, and analyze the model to estimate the qualitative cost of change. Develop a high-level presentation view with the associated cost of change.

Navigating and verifying the model reveals, for example, that there are relations between the sub-units of Figure 4. Depending on their weight, the distinction between sub-models does not reflect the realization. A pure reconstructed view uncovers dependencies between sub-units, but the decision as to whether the architectural construct sub-unit exists in reality depends on the analysis and judgment of the architect. In fact, the decision can be taken due to the use of particular program structures at the code level rather than the high-level design.

Note that the analysis step is not primarily performed on a view level but rather at the modeling level. It is essential to understand the current platform dependencies and compare them to dependencies caused by the new platform. Not surprisingly, the initiated analysis task migrates naturally into a reengineering task.

3.6. Adding expertise

Once a model is constructed, it provides a useful base for further analysis. For example, the STA organization considers possibilities, such as:

- porting STS to several real-time operating systems and isolating the used platform operations in an operating system abstraction layer; components and relations have to be added, manipulated, or reorganized;
- adding a further sub-unit to the STS and analyzing the impact on the timing behavior.

In both cases the change input does not come from the source (the existing system) but from interventions with the intention of exploring new scenarios. These scenarios are called *What If* scenarios, because they explore new stimuli and the expected system responses. *What If* scenarios add to the complexity of system understanding by requiring the prediction of future system responses. Models provide a description about the current system. However, they do not have the capability of predicting the reactions as stimulated by *What If* scenarios. The stimuli of *What If* scenarios are input to a model expert as illustrated in Figure 5. Responses of model experts to *What If* scenarios include, depending on the particular expert, improvement feedback for the current model as well as to the existing system.

Bachmann *et al*. [15] introduce the notion of tactics as a means to control a quality attribute response by manipulating some aspect of a quality attribute model through architecture design decisions. A tactic is an architecture strategy that *is concerned with the relationship between design decisions and a quality attribute response*. There are collections of tactics available to achieve particular quality attribute goals (for further details, refer to [15]). We outlined an initial approach to develop model experts in our QADSAR paper [13].

Different quality attributes have quality attribute models with different accuracy requirements. Therefore, model experts are formal or informal, depending on the related quality attribute.

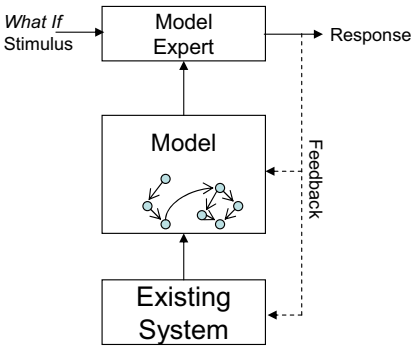

Figure 5. *What If* scenarios.

For example, performance models can be fairly formal (queuing models) while usability models for customer satisfaction can be more informal. A modifiability expert provides rather informal estimations of change efforts.

The case study of Section 4 provides an example model expert, where predictions are required to provide time performance design explorations for customer configurations. Once the performance model is constructed, the case study shows ways to modify time budgets, network properties, or changes of component deployments. The developed performance expert knows about performance properties, such as scheduling algorithms. In addition, the expert provides prediction capabilities for exploring worst-case reaction times. However, *What If* scenarios are limited to the knowledge and scope of the constructed model expert. Consequently, not every *What If* scenario may be covered by the expert.

4. AUTOMOTIVE CASE STUDY

The case study was carried out on a system in the automotive industry. The system is a door module comprising a set of hardware modules, such as window lifters, door locks, associated software, and more. The case study focuses on the construction of the time performance model and the development of the model expert. This section provides:

- a system overview of door modules;
- a description of the scenarios;
- construction of an initial time performance model;
- fact extraction from sources;
- abstraction of detailed information;
- model construction;
- model expert creation and verification;
- scenario feedback;
- cost and benefit.

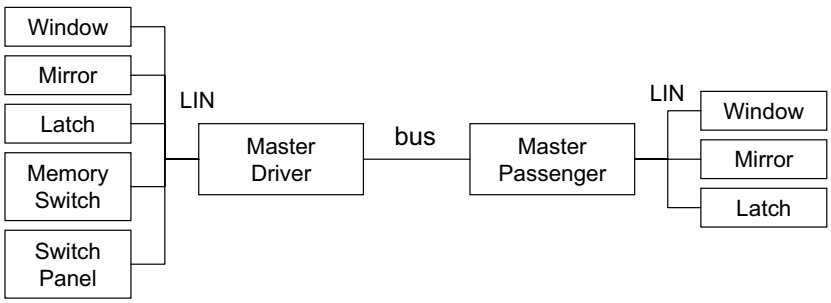

Figure 6. Example configuration.

This sequence reflects the process that we followed to perform the case study. The *fact extraction* and *abstraction* sections are common activities in a reconstruction process. However, this paper does not consider further aspects of the reconstruction process. The interested reader is referred to [16] for more information about the process.

4.1. The door module

A door comprises a set of hardware modules, such as a window lifter, door lock, switch panel, and mirror. In the past, door modules were mechanical systems with manual hand cranks and manual adjustable mirrors. Today, these systems evolved into mechatronic systems, combining mechanics and electronic controls. The added electronic parts allow the addition of distinguishing customer features, such as slow and fast window sliding, or key-less entry. It also adds the necessity for safety regulations, such as anti-pinch standards in case of encountering obstacles during the window closing operation. Other popular examples for mechatronic systems in the automotive domain are adaptive cruise control, or rain-sensing wipers. Electronic door modules are not independent from each other. They build a master–slave system, with centralized functionality on a master, and specialized functions on the slaves. For example, a button press on a switch panel (slave) is translated in the master component to a *move window* command which eventually gets transferred to the window lifter component (slave). In this functionally partitioned model, slaves are reduced to their elementary functions. Coordination functions and administrative tasks, such as storing of personalized mirror positions, are located on a master module that orchestrates the door functionality.

Figure 6 illustrates an example for a configuration of a two-door car with a master for the driver side and a master for the passenger side. Both masters are connected via a bus, such as CAN [17] or FlexRay [18]. Besides the two masters there are a variety of further electronic control units (ECUs) connected to the bus that are not considered in this case study. The LIN bus for the slaves is a cost-optimized serial bus for low-traffic communication [19].

4.2. Scenario collation

The organization wanted to accomplish three goals from this case study.

1. Exchange hard-wired connections with flexible bus structures to reduce wire harness. The cable length for an average vehicle is about 1.6 km with 300 plugs and 2000 cable pins [20]. The cable cost is a cost-sensitive part in a modern vehicle (average cost about €400 [21,22]).
2. Partition the functionality in such a way that the mechatronic module cost is as low as possible.
3. Flexible topology of mechatronic systems for a rich variety of customer configurations. For example, replacing a two-master configuration with a one-master configuration for driver and passenger side.

The organization developed two- and four-door prototypes in cooperation with car manufacturers (customers) to explore design alternatives and prove the validity of the model. One of the main issues arising out of these projects was time-performance characteristics, such as worst-case reaction time for different system configurations. In general, customers wanted one of the following scenarios.

- Scenario A: adding a new peripheral such as climate control to an existing LIN channel.
- Scenario B: adding a second master to the configuration and migrating functionality from the original master to the additional master.

The organization was in the situation of either building further prototypes and performing measurements for worst-case reaction time, or defining a formal model that is able to calculate time-performance scenarios for a variety of configurations, including the addition of further LIN devices, such as a climate control unit. The organization decided to take the approach of building a model, because of:

1. the cost involved in building prototypes; and
2. the ability to explore further customer scenarios using the model.

Scenarios A and B represent the *What If* scenarios that were introduced in Section 3.6. Therefore, the model approach includes the development of both the time performance model (Sections 4.3–4.6) and the model expert (Section 4.7).

4.3. Constructing an initial time-performance model

Constructing the initial time-performance model forms the hypothesis that allows for fact elicitation from sources, as earlier illustrated in Figure 3. Only those facts that affect performance aspects have to be considered.

The initial model follows a stimulus/response pattern. The model response consists of minimum, average and worst-case end-to-end system response times. There are many ways to define minimum, average, and worst-case reaction time, depending on the type of system, networks, and more. In this case, we favored a pragmatic solution particular for this system.

- The minimum response time is a best case with a single button press just before the polling cycle, clear network channels, and the master not busy with previous events.

- The average response time differs from the minimum case in that the button press happens just after the polling cycle.
- The worst-case response time determines a situation with network traffic while the master is busy with other events, such as previous button presses.

Note that the minimum, average, and worst-case scenarios have to be weighted with probabilities in order to reflect real-world scenarios. For example, the probability of a worst-case scenario is extremely low. Most real-world response times will be slightly faster than the response time given by the described average scenario.

The initial model construction primarily required interviews with the system developers. The interviews resulted in the following model.

- **System topology:** the system consisted of three system component types: master, slaves, and connecting buses. The master was connected to five slaves using the LIN version 1.3 bus protocol [19]. Master and slaves each contained a microprocessor. The system topology is represented in a graph where each node is a system component. The graph is restricted by the following rules:

  1. slave and master nodes are connected via a bus node;
  2. bus nodes can only be connected via a master node.

- **Messaging:** under normal execution, the master polls each of the slave nodes querying for state change information. Upon notification of a state change, the master does a calculation that results in a new set of messages that are sent as a response to other slave nodes residing on the bus. This new set of messages is scheduled by statically defined message priorities as well as debounce times given by the individual slave node. When all the events have been handled, the system returns to its normal polling state.
- **LIN as the key element:** from the performance perspective, the LIN communication seemed to be the key in understanding the system's timing. All LIN messages are required to be a multiple of a global time tick. Given the bus nature of LIN, every message can be thought of as a global broadcast, which allows the master to arbitrate slave-to-slave communication while at the same time ensuring coarse-grained time synchronization. This helps further reduce the cost of the system by eliminating the need for expensive hardware crystals for synchronization while maintaining that all nodes on the bus are synchronized to within at least one global time tick.
- **Scope reduction:** the structure of the LIN bus allows us to model each slave as a system component that either succeeded or failed to meet the LIN specified deadline. The slave typically reads analog or digital port values, sets an output value to control an actuator and then services a communication request. Once this has been guaranteed to occur within the LIN global time tick, no further analysis is required. Instead, the important transactions occur on the master. Even in the case where a slave fails to meet a deadline, the error must be handled on the master in order for the system to effectively compensate. Given this, slaves can be reduced to system components that do not have to be refined in the initial model.
- **Cyclic executive:** due to cost reasons, the master does not run a preemptive operating system. Instead, it runs a cyclic executive loop. The loop operates at the LIN global tick frequency so that it can process and produce the next LIN message without missing a communication opportunity.

Previous work has analyzed cyclic executives [23], but these do not typically account for dynamic runtime dependencies. Granted, the same set is called each cycle, but they generate chains of events that alter what type of execution will be required on the next iteration of the executive loop. Because this application logic is mixed into the dynamic scheduling, there are no obvious guarantees usually associated with periodic scheduling.

The initial model resulted in the primary analysis of the master with its LIN communication package to determine node properties in the system topology graph.

4.4. Fact elicitation from the master

The master software was written in C and comprised around 45 KLOC with 760 functions, and 3100 variables. The elicitation for the master was performed in two steps.

1. Elicitation of the call graph—to identify threads of execution.
2. Elicitation of the control flow graph—to investigate the control flow via data dependencies.

4.4.1. Call graph

The call graph was generated to investigate whether major parts of the cyclic executive could be represented as a branch off the *main* function. This was a first effort towards extracting the major components of the system. In an ideal case, the call graph of the program would have built a hierarchy of components. Understanding the main components at the initial depths of the call tree would have been a first step towards building a system component map. Figure 7 shows a spring layout of the call graph in the master. This graph was automatically generated by a custom program that utilized C-scope [24] to find functions and the Graphviz [25] graphical drawing package to plot the resulting data. The call graph depicts particular functional responsibilities, such as the LIN communication, and parts of the application logic. Some branches in the graph share functions, but most of the functions are directly or indirectly connected to the *main* function. This structure tends to occur in low-memory footprint-embedded environments, because stack space needs to be conserved. Instead of using a hierarchy of function calls, the software on the master uses conditional flags that tightly connected many of the functions. Even though the main cyclic executive loop calls nearly 50 functions, on any particular cycle, only a handful of those execute. Not only does this mean that the separation of different execution threads is not automatically extractable from the call graph, it also indicates the difficulty in achieving this task by manual inspection. A different technique is required in order to extract the threads of execution.

4.4.2. Control flow graph

The second fact elicitation step to resolve threads of operation was performed by investigating the control flow graph of the master. Using a reverse engineering tool, such as Imagix 4D [26], it is possible to isolate individual regions of the source code and generate a control flow graph, listing all possible paths of program execution. We chose Imagix because of its ability to resolve advanced C language

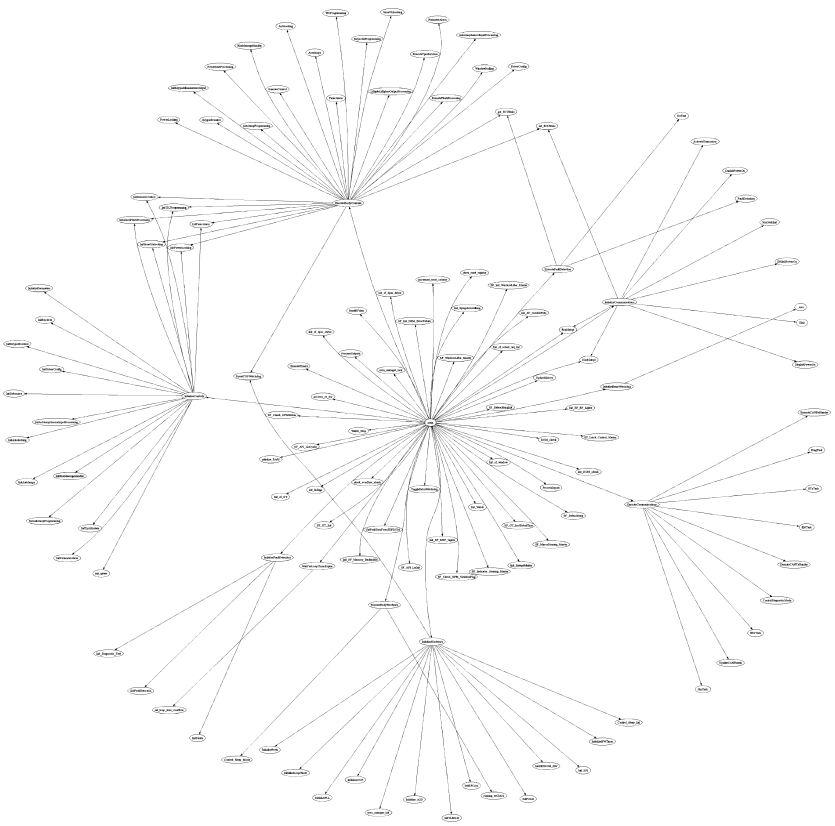

Figure 7. Call graph.

constructs such as function pointers. Unfortunately, the tool is unable to decompose sub-functions, prohibiting the creation of graphs that are multiple function calls deep. It also has the limitation of only being able to export an image of the flow graph instead of providing the traversed information. However, it is possible to enable a debugging mode where traversal information is stored in a trace file. By writing a short script, the flow graphs for each function from the debugging file can be recreated. These graphs could then be linked together to form the entire control flow graph of the program. Figure 8 shows an example graph on the left of a small flow diagram. The graph of the entire source contained thousands of nodes with tens of thousands of connections making it unsuitable for visual analysis.

In order to reduce this flow graph, regions of the source consisting of consecutive logic blocks separated by function calls that were not interrupted by a branch were merged together. The right side of Figure 8 illustrates this compression.

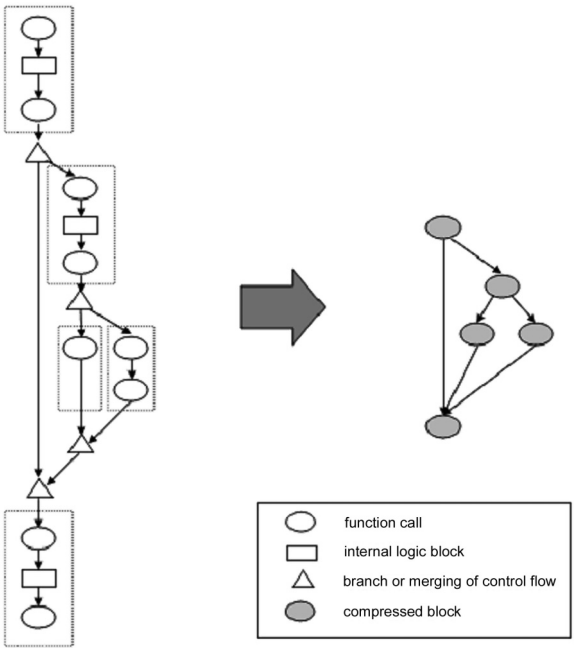

Figure 8. Compression of a control flow graph.

4.5. Abstraction

Abstraction in the context of architecture reconstruction strives to collapse detailed source information into architectural elements. The strategy to achieve this objective depends on the type of system and the model to be constructed. In this context the emphasis is on the master and useful abstractions for time performance. Source analysis, expert interviews, and analysis of specifications and/or written documentation are important to build a model describing what the important elements in the system are, and how they relate to each other. In the door module, these elements may or may not have been physical hardware or software components. We did find that physical location did tend to help in organizing the components. Figure 9 illustrates areas of interest and encapsulates a performance-related state machine of different event paths that can occur in the system. The dashed rectangles illustrate the physically isolated system components: master and slaves with sensors and actuators. These boundaries help clarify what the nodes in those particular regions are responsible for. For example, the cyclic executive inside the master contains all of the elements of the master's main loop that will consume time. The shaded regions in Figure 9 designate three major areas of interest with respect to performance as well as where this information was discovered:

- cyclic executive of the master (source code analysis);
- LIN network (source code and Line Description File analysis);

- the interfaces between the slaves and the LIN bus as well as the interface between the slaves and their environment (component specifications and/or measured timing).

These regions were determined to be most important because they have the largest effect on the end-to-end latency of the system. Other minor components such as the *Other Comm.* on the master have a fixed or trivial influence on the overall system timing.

4.5.1. Cyclic executive of the master

The starting point for the software model of the master was the compressed control flow graph. Much of the information contained in this control flow graph does not directly affect time performance. Initialization sections could be ignored since they only occur at startup. All LIN related communication could be grouped into a single block that would have to be analyzed separately. Other low priority communication with the rest of the car environment could also be isolated. As illustrated in the lightest gray region of Figure 9 (the *Cyclic Executive* block), we abstracted the main cyclic executive loop into several major components. The master block represents the aggregated and abstracted source level information. This sub-graph was generated using human inspection of the compressed control flow graph. Worst-case reaction time—and not functionality—defined the critical components found in this graph. The beginning of the cyclic executive loop waits for a timer to expire in order to remove computational jitter and drift. This allowed us to ignore branches in the control flow graph that bypassed major functionality.

4.5.2. LIN schedule analysis

The first step towards understanding the LIN communication in the code was to interview an expert that had previously worked on the code. During this interview certain naming conventions associated with LIN communication were outlined as well as a description of the overall flow of messages. The LIN schedule depicted in Figure 10 is constructed by LIN function calls that pass scheduling information in the form of message data structure arguments. For example,

```
status = fs_lin_oneshot_schedule (RX_SLAVE_2_Check);
```

schedules a message requesting information from slave two. `RX_SLAVE_2_Check` is a predefined LIN data structure to be requested. All messages are outlined in a LIN description file that is used at compile time by the LIN driver to set up the message data structures. Manual inspection was required to extract the different LIN function calls and message priorities from the master's source, but this process could have been automated given previous source code annotations. The LIN description file provides information about the size of the individual messages and the configuration of the LIN bus (baud rate etc.). Figure 10 shows a state diagram of the different possible message chains that could be generated by system inputs.

4.5.3. Slave node analysis

The details of the internal software executing on the slaves are not important with respect to overall system performance. The slaves are bound by the global tick of the LIN bus and are therefore required

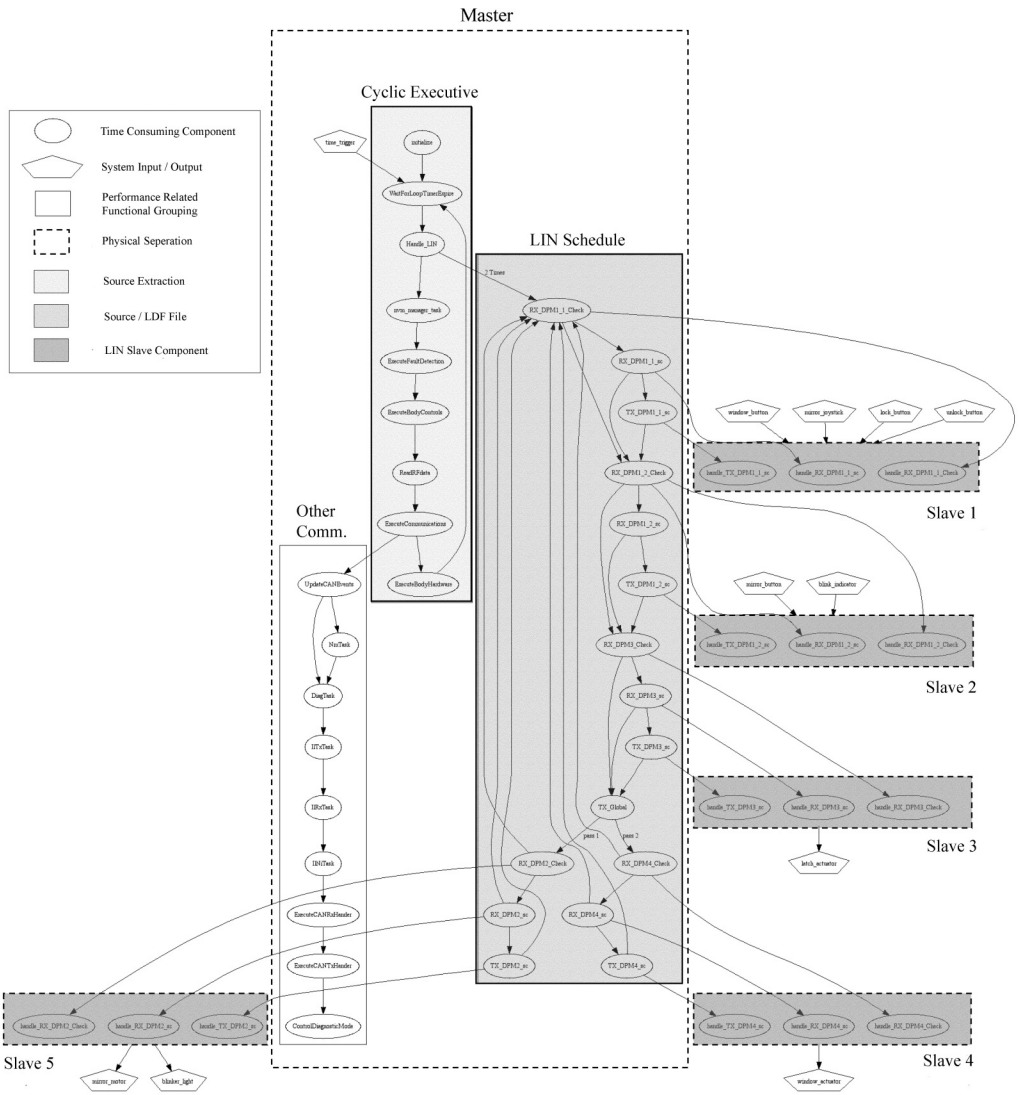

Figure 9. Aggregated information.

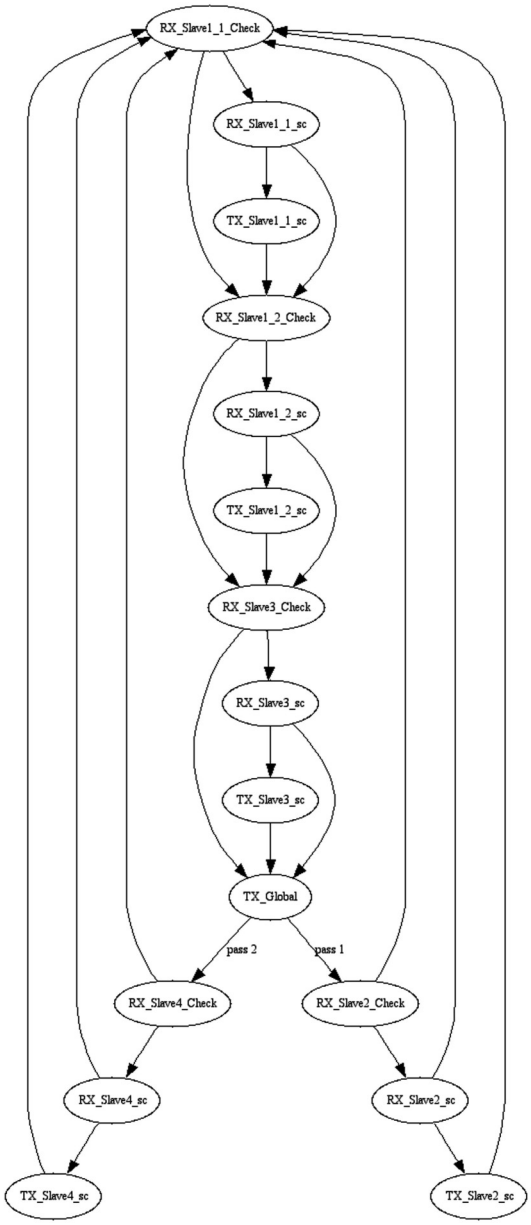

Figure 10. LIN state diagram.

to complete all of their tasks within that period. This allows us to view them as hardware components that only respond to LIN messages. Even if the slaves miss their LIN deadline, the error handling responsibility would still fall on the master node. The only other consideration is that it could take longer than a single LIN global tick to process sensor data or control an actuator. An example of this is the time required for the window lifter to raise the window. This time must be factored into the overall end-to-end latency, but it does not change the slave nodes' response time to LIN messages and therefore has no effect on the master node's timing. The physical timing values associated with different actuators were either measured by the designer of that specific node, or as is the case with the sensors, amortized by the LIN communication overhead. In Figure 9 each slave is shown containing a LIN message handler and connects to a pentagon shape that represents sensors or actuators. When developing the performance model, these LIN functions will be factored into the LIN component leaving the slaves to only contribute the extra time required to engage the actuators.

4.6. Model construction

It is important to develop a model that gives adequate responses while still operating at an abstract enough level to allow for important parameter adjustments. Past approaches to modeling automotive systems tend to create a single cohesive model made up of components that internally direct control and data flow [27,28]. In these approaches, components have a mapping that connects inputs to outputs, and then consume a resource. These systems can elegantly describe a single configuration of the system, but they are difficult to modify in order to perform design explorations because too much of the control flow logic is encapsulated in the individual components.

Other approaches [29,30] have gone beyond state machine-based models to provide accurate timing information through simulation. Given the complexity of modern systems, simulation-based approaches are rapidly becoming implausible. The door module comprises a huge number of states that are all dependent on relative timing between different environmental or user inputs. It would be impossible to exhaustively simulate these inputs in order to guarantee that all corner cases are covered.

As a solution to this problem, we propose a modeling system where the components can be isolated from the control and data flow through the system. The modeling system is represented by a graph, as described in Section 4.3. Each node in the graph has a set of configuration parameters that define the system components (masters, slaves, buses) and their interconnections. The messages, events, and data that are exchanged by the nodes are specified in a stimulus file. This file explicitly defines each hop across the graph topology accumulating performance information at each step. Figure 11 shows our scenario's topology with a sample trace diagram on the right. The numbers along each arrow in the trace diagram show the sequence of events that an *unlock button* action would take when unlocking the door. For example, lines 1–4 depict the node *master* requesting state change information from node *slave_1*. The data are transmitted through the node *LIN* to the slave node and then back to the master node. Each one of these intermediate hops along the trace needs to be formally analyzed in order to generate the worst-case reaction time as a function of the specific configuration.

The information required to generate each of these worst-case components came from the previously described software analysis of the master, the communication analysis of the LIN bus and the reaction times of the sensors and actuators on the slave nodes. In our example scenario, most of the important timing values are based on the LIN global time tick and the different message lengths extracted from the LIN description file. The relationship between these LIN timing values and the system's inputs comes

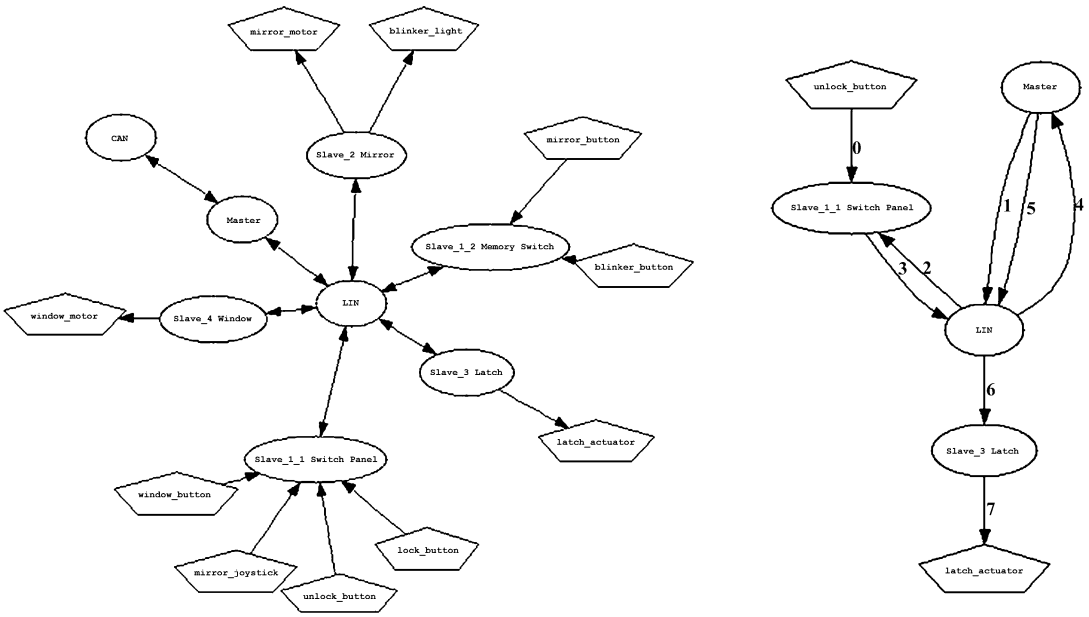

Figure 11. Graph topology (left) with sample trace (right).

from the abstracted control flow information that was extracted from the master's source. The final timing information associated with the sensors and actuators was either physically measured by the designer of that particular slave or, in the case of the sensors, amortized by the LIN communication.

4.7. Building the model expert

The model expert provides stimulus/response pairs for *What If* scenarios. With this, the model expert generalizes the model as constructed in Section 4.6. Figure 12 illustrates the model expert architecture. The user must supply the executive with two inputs.

- The node library—containing functions with the node performance characteristics.
- The stimuli file—containing a trace of events and messages between the nodes and the list of nodes that have to be processed.

The nodes in the node library are implemented as C functions. They are linked into the model executive, which interprets the stimulus file and facilitates the calling, as well as record keeping, of the individual node executions. When all of the stimuli data are interpreted and processed, the executive then packages and displays the performance response. In the following, we will explore the implementation of an example scenario by describing the details of the node library and the stimulus file.

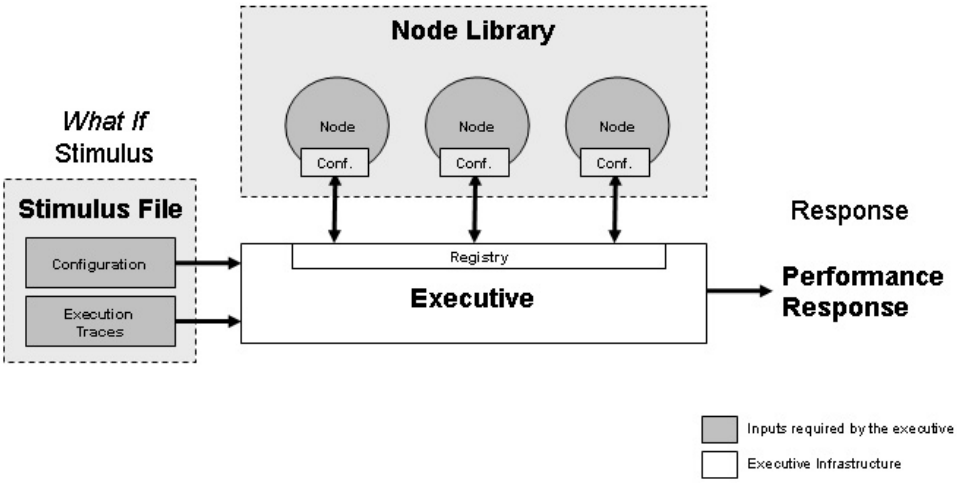

Figure 12. Model expert architecture.

4.7.1. The node library

Each node in the node library is defined as a function that takes in operational parameters and returns a performance data structure given a specific set of inputs. For example,

```
tm_data lin(void *args[]) {
            tm_data time;
            ...
            time.min=calculated_min;
            time.avg=calculated_avg;
            time.max=calculated_wcet;
            strcpy(time.title,"LIN network transaction");
            return time;
        }
```

For the purpose of performance modeling, the `tm_data` structure stores the worst-case reaction time, the average reaction time, and the minimum reaction time. It has an additional field that can be used to pass annotations that are used by the executive to clarify the output. The `void *args[]` is used as a generic method for passing arguments that will later be specified in the stimulus file. The executive provides a set of helper functions in order to extract arguments. For example,

```
instance=get_int_arg(args,0);
// argument pointer,  argument index
event_type=get_int_arg(args,1);
message_priority=get_int_arg(args,2);
```

In this case, the second parameter passed to the helper function `get_int_args( )` is the position of the argument in the argument list `*args[]`. The input parameters must be defined by the designer of the node and should incorporate as many of the critical performance-related factors as possible. In the above example we show that the `event_type` and `message_priority` are the two critical LIN performance factors (`instance` just allows for multiple instances of the same node). These critical input parameters came from a detailed analysis of possible LIN usages that were explored by manipulating different message combinations and system configurations on a spread sheet. Consequently, the worst-case reaction time is a function of the two input parameters in conjunction with numerous static configuration parameters. These parameters are passed to the LIN configuration function at startup before the node is executed. Below is an example of the LIN configuration function:

```
tm_data lin_config(void *args[]) {
int instance;
instance=get_int_arg(args,0);
LIN_GLOBAL_TICK[instance]=get_int_arg(args,1);
LIN_BAUDRATE[instance]=get_int_arg(args,2);
LIN_SLAVE_NODES[instance]=get_int_arg(args,3);
LIN_POLL_LOOP_SIZE[instance]=get_int_arg(args,4);
LIN_RX_MSGS[instance]=get_int_arg(args,5);
LIN_RX_MSG_SIZE[instance]=get_int_arg(args,6);
LIN_TX_MSGS[instance]=get_int_arg(args,7);
LIN_TX_MSG_SIZE[instance]=get_int_arg(args,8); }
```

The important parameters are shown in all capitals proceeded by `LIN_`. These parameters define the connections and configurations of the nodes previously described in Figure 11. These values are used during the stimulus file execution in the `lin()` node like this:

```
polling_time=(1000/(LIN_BAUDRATE[instance]/(LIN_POLL_SIZE[sel]*8)));
```

In this example, a variable `polling_time` is computed and will be used later in the node's timing calculations. Eventually, the `lin()` function will return the `tm_data` structure that will be recorded by the executive. Once the node library is constructed, it is important to write a specification that adequately describes the different model functions. This will allow future developers to reuse the nodes without having to fully understand all of the detail that went into the analysis.

4.7.2. The stimulus file

The stimulus file configures the nodes and executes different nodes with user-specified parameters. The stimulus file contains traces that outline a path through the node topology graph. The configuration is a special instance of a trace that only executes once and does not collect performance data. The stimulus file is a text file that is parsed and executed by the executive at runtime. This is important because it makes the addition of graphical user interfaces possible, it protects intellectual property by allowing the distribution of binary node libraries, and it facilitates rapid execution of trace permutations without requiring recompilation. Below is an example of the configuration section of a stimulus file:


```
#config
register_components();
slave_config(0,5);        //  Instance, cycle_time
master_config(0,5,8);     //  Instance, cycle_time, cpu_speed
lin_config(0,5,19200,5,4,2,16,3,16);
// Instance, global_lin_tick,baudrate,number_of_slaves,
//polling_loopl_size,RX_msgs,RX_msg_size,TX_msgs,TX_msg_size
```

The `#config` is a keyword used in the trace file to signify a trace that should execute only once, and should not collect performance data. As shown in Figure 12, all transactions between the executive and the nodes pass through a registry. The `register_components()` method is required in the node library and is responsible for associating a plaintext name (i.e. `slave_config`) with the correct memory addresses for that function. The lines of code remaining load the configuration parameters for the rest of the nodes described in Section 4.7.1.

Next, in the stimulus file are the execution traces that are to be analyzed. For example,

```
*Door Lock Press
        slave(0,$BUTTON_PRESS);
        master(0,$LIN_DEP);
        lin(0,$RX,$LOW_PRI);
        slave(0,$PROCESS);
        master(0,$LIN_DEP);
        lin(0,$TX,$LOW_PRI);
        slave(0,$ACTUATE);

*Window Button Press
        slave(0,$BUTTON_PRESS);
        master(0,$LIN_DEP);
        lin(0,$RX,$LOW_PRI);
        slave(0,$PROCESS);
        master(0,$LIN_DEP);
        lin(0,$TX,$LOW_PRI);
        slave(0,$ACTUATE);
```

Traces have names that start with a `*` character. For instance, the functions following `*Door Lock Press` would be stored as the response associated with a door lock button being pressed. Each of the following function calls specifies the node that is to be called as well as the arguments that should be passed into it. Values starting with the `$` character are defined by the node library and alias to an associated value. Similar to `#defines` in C, this helps make the stimulus file more readable. The first value passed to each of the node specifies a particular instance. In this example, there is only one instance of each node. The model executive's output is separated into four major sections: the registration, the configuration, the execution, and the response phase. The registration phase displays the names of all nodes that are available as well as checking that they have a valid function pointer associated with them. The initial lines from the simulation look like this:

```
Running stimulus file "current.tra"
Registered [add_defines]
Registered [master_config]
```

The configuration phase checks the configuration argument parameters, and executes the associated configuration node. The configuration nodes can use this as an opportunity to log their current configuration. For example,

```
SLAVE_0 configured:
    cycle_time: 5 MASTER_0 configured:
    cycle_time: 5
    cpu_speed: 8Mhz
LIN_0 configured:
    Global Tick Time: 5ms
    Baud Rate: 19200
    Number of Nodes: 5
    Poll Msg Size: 4 bytes
    RX messages: 2
    RX size: 16 bytes
    TX messages: 3
    TX size: 16 bytes
```

The execution phase will call each node and pass it the parameters specified in the stimulus file. The return values of these nodes are then collected by the executive to be displayed compactly after all traces have completed. By default, the executive will display the node being executed and the times consumed. For example, the door lock button trace would produce

```
0: Door Lock Press SLAVE_0  called
    Waiting for input: min = 0ms, avg = 0ms, max = 0ms
MASTER_0 called
    Master Default Event: min = 0ms, avg = 0 ms, max = 0ms
LIN_0 network transaction
    LIN RX time: min = 15 ms, avg = 40ms,  max = 80ms
SLAVE called
    Processing time: min = 0ms, avg = 2ms,  max = 5ms
MASTER_0 called
    Master Default Event: min = 0ms,  avg = 0ms, max = 0ms
LIN_0 network transaction
    LIN TX time: min = 15ms, avg = 25ms, max = 50ms
SLAVE_0 called
    Driving Actuator: min = 1ms, avg = 5ms, max = 10ms
```

Finally, after all of the traces were executed, the total reaction time is displayed for each trace. For example,

```
Trace "Door Lock Press "
  min: 31 ms
  avg: 72 ms
  max: 145 ms
```


4.8. Model verification

The next step is to verify that the model expert is returning adequate timing information. The model expert is based on the analysis of a real system which it can be compared against. The difficulty is simulating the inputs into the system at the precise times and in the correct order so as to stimulate the worst-case response times derived from the earlier analysis. In our system, we chose to use Vector's Canalyzer [31] software, an analysis software for networks in distributed systems, to masquerade particular nodes in the system to infuse particular stimuli. Using the CAPL (CAN Access Programming Language) of the Canalyzer software, it was possible to set up the conditions when fake messages should be broadcast on the bus. These messages appeared to the master as if they had been sent by a slave and usually indicated that a sensor had been triggered. We then collected the remainder of the messages with timestamps allowing us to see how long it took for the master to respond to the stimuli. Since we are comparing the real system to a model that was directly based on it, it should be no surprise that our timing values were nearly identical. For example, unlocking a door took minimally 15 ms after pressing the button, and a maximum of 130 ms, while our model predicted 15 and 135 ms. The 5 ms difference in the maximum time is due to limitations in the network traffic analyzer we used. The limitations did not allow us to simulate events at the very end of the LIN global time tick. Even so, this demonstrated that our critical paths existed in the system and that the system was performing as the model expert had predicted. We experimented with a few random input sequences to make sure that none exceeded our calculated critical path. None of the random tests where nearly as long as the worst-case path and most times were similar to our estimated average latency. This was another indication that the model was adequately describing the real system.

4.9. Scenario feedback

At this point we are in a position to discuss the feedback that our model expert predicted about our design scenarios from Section 4.2.

The goal of scenario A was to investigate the implications of adding a climate control system to an existing LIN channel. In such an experiment we hoped to gauge the scalability of our current configuration. The left columns of Tables I and II show the resulting stimulus file and output of the newly configured system. As expected, the worst-case response times of the button presses increase from around 130 ms in our current system to 175 ms. This large increase in latency could make the system fail to meet customer end-to-end latency requirements. It is impossible to keep adding functionality to the master node without eventually seeing unsatisfactory performance.

Scenario B investigated a possible solution to the problems posed in scenario A. Instead of just adding a climate control system, we also added an additional master to support the new climate control system from scenario A. In order to load balance the devices we also migrated the actuators for the passenger side of the vehicle onto the new master. Button inputs should still be located on the original master's network since it already contains the other system buttons. The passenger side actuators would now utilize a new LIN channel. The driver and passenger side masters would then communicate over a third LIN channel in order to pass button commands from one master to the other.

Each component is developed with enough flexibility so that only the initialization parameters have to be adjusted when altering the node topology. Table I compares the original and modified stimulus file. In the new stimulus file, the additional master communicates via an added LIN channel to the

Table I. Scenarios A and B.

```
#config
   add_defines()
   slave_config(0,5)
      //ID, cycle_time
   master_config(0,5)
      //ID, cycle_time
   lin_config(0,5,19200, 5,4, 3,16,4,16);
      // ID, global_tick,baud
      // num_of_slaves,poll_size,
      // RX_msgs,RX_size,
      // TX_msgs,TX_size
*Door Lock Press
   slave(0,$BUTTON_PRESS);
   master(0,$LIN_DEP);
   lin(0,$RX,$LOW_PRI);
   slave(0,$PROCESS);
   master(0,$LIN_DEP);
   lin(0,$TX,$LOW_PRI);
   slave(0,$ACTUATE);

*Increase Temperature Button Press
   slave(0,$BUTTON_PRESS);
   master(0,$LIN_DEP);
   lin(0,$RX,$LOW_PRI);
   slave(0,$PROCESS);
   master(0,$PROCESS);
   lin(0,$TX,$LOW_PRI);
   slave(0,$ACTUATE);
```

```
#config
   add_defines()
   slave_config(0,5)
      // still only one slave configuration
   master_config(0,5)
      // original master configuration
   master_config(1,5);
      // second master for climate control
   lin_config(0,5,19200, 4,4, 2,16, 3,16);
      // original master-slave channel
      // ID, global_tick,baud,
      // num_of_slaves,poll_size,
      //RX_msgs,RX_size
//       TX_msgs,TX_size
   lin_config(1,5,19200, 1,4,1,16, 1,16);
      // channel between the two masters
   lin_config(2,5,19200, 1,4, 1,16,1,16);
      // channel from new master to slaves

*Door Lock Press
   slave(0,$BUTTON_PRESS);
   master(0,$LIN_DEP);
   lin(0,$RX,$LOW_PRI);
   slave(0,$PROCESS);
   master(0,$LIN_DEP);
   lin(0,$TX,$LOW_PRI);
   slave(0,$ACTUATE);

*Increase Temperature Button Press
   slave(0,$BUTTON_PRESS);
   master(0,$LIN_DEP);
   lin(0,$RX,$LOW_PRI);
   slave(0,$PROCESS);
   master(0,$PROCESS);
   lin(1,$RX,$LOW_PRI);
   master(1,$PROCESS);
   lin(2,$TX,$LOW_PRI);
   slave(0,$PROCESS);
```

Table II. Response before (left) and after (right) scenario B modifications.

Running trace file scenarioB_1.tra	Running trace file scenarioB_2.tra
SLAVE_0 configured:	SLAVE_0 configured:
cycle_time: 5	cycle_time: 5
MASTER_0 configured:	MASTER_0 configured:
cycle_time: 5	cycle_time: 5
LIN_0 configured:	MASTER_1 configured:
Global Tick Time: 5ms	cycle_time: 5
Baud Rate: 19200	LIN_0 configured:
Number of Nodes: 5	. . .
Poll Msg Size: 4 bytes	LIN_1 configured:
RX messages: 3	. . .
RX size: 16 bytes	LIN_2 configured:
TX messages: 4	. . .
TX size: 16 bytes	
. . .	. . .
Trace “Door Lock Press”	Trace “Door Lock Press”
min: 30	min: 30
avg: 87	avg: 64
max: 175	max: 130
Trace “Increase Temperature Button Press”	Trace “Increase Temperature Button Press”
min: 30	min: 45
avg: 87	avg: 66
max: 175	max: 135

original master. The additional master then has a second LIN channel for communication with its own slave nodes. Notice that only the configuration settings and the immediately effected traces have to change. All of the other elements in the system, such as the door lock button, remain the same. There is still only one instance of a slave module, because all of the slaves in the system are identical with respect to response times.

Table II shows the abbreviated response produced by running the two traces. Contrary to our initial intuition, the end-to-end latency from the temperature button press on one slave to the temperature controller unit on another slave decreased from 175 to 135 ms. In fact, all of the system latencies decreased even though the modified stimulus to response path requires more hops. This demonstrates how significant the LIN network usage affects the system's overall performance. Specifically, the number of slaves on the LIN bus that actively pass messages drastically changes the worst-case latencies. This is due to the cyclic nature of the message scheduling. When a slave is removed, the polling loop time drastically decreases, yielding proportionally lower end-to-end message latencies.

4.10. Cost and benefit

Developing systems in a cost-efficient way forces companies to constantly improve their products in accelerating markets. One way to achieve this in mass markets is the trend towards product lines.

For example, different models are based on the same production platform. This translates directly to the software platform for vehicles with the creation of standards to integrate software in a platform as, for example, envisioned by the AUTOSAR consortium [32]. The resulting artifacts describe, among other things, *plug* standards for software components, such as for the automotive body domain of this case study. However, *plug* standards do not guarantee component *play*. Key to *play* standards is the ability to predict the behavior of assembled components. Solutions to this ability include models for quality attributes translated into particular domains.

The introduced performance expert of this case study is a natural but prerequisite step towards automotive body domain *play* standards as the LIN bus structure with a client–server approach will penetrate the market. *Play* predictions for other automotive manufacturer configurations with similar *plug* standards will become cost efficient. A particular solution can be predicted before the commitment of large resources. Of course, additional model experts are required towards the full realization of this ambitious goal. For example, safety model experts in x-by-wire constellations, where communication is done without mechanical backups, such as automotive steer-by-wire and break-by-wire [33].

Creating a model expert for an existing system is probably expensive in case the constructed model is used only once and/or the system is already delivered. The development of the time performance expert of this case study included the following:

- developer interviews, investigation existing documentation, scenario development;
- tool selections, source code parsing (master);
- initial performance model;
- building a target verification environment for the initial model;
- design and coding of the performance expert;
- documentation.

The total effort was approximately 2 person months. With further customer configurations, the investment in a model expert is extremely beneficial, not only as a tool for developers but also as a useful support tool during product acquisition phases.

5. CONCLUSIONS

The model-centric SAR approach establishes the link between quality attribute driven analysis and architecture reconstruction. The business goal driven approach of system understanding provides an efficient way to steer the reconstruction process by providing the required models for a particular system. The quality attribute related model experts are an efficient way to infuse the reconstruction and analysis process to measure the response to *What If* scenarios.

Model construction requires effort. Depending on the required model accuracy the construction effort can be too expensive. On the other hand, an inadequate model does not provide reasonable answers to the business context. Depending on the context, the understanding and analysis is performed with interview and presentation techniques, in a reconstruction effort based on source code analysis, or a mixture of both techniques.

The automotive case study has shown the application of the approach for a performance model in an embedded system. It shows that models are not restricted to software models. Often, they require additional system aspects, such as deployment, processor performance, communication protocols, and

available memory, in particular for embedded systems. The introduced formal approach allows a model expert to predict worst-case timing of new deployment scenarios for distributed mechatronic systems before the commitment of resources to build the particular configuration. This design exploration enables organizations to quickly respond to new customer settings. As systems are increasingly incorporating domain standards we expect rapid demand for model experts in the area of component assembly predictions.

This case study showed evidence that a model-centric approach provides a substantial contribution to leverage SAR in concrete organizational contexts on the basis of a detailed investigation of quality attributes for particular classes of systems.

ACKNOWLEDGEMENTS

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*.

REFERENCES

1. Klusener AS, Laemmel R, Verhoef C. Architectural modifications to deployed software. *Journal—Science of Computer Programming* 2005; **54**:143–211.
2. Seacord RC, Plakosh D, Lewis GA. *Modernizing Legacy Systems*. Addison-Wesley: Reading, MA, 2003.
3. Faust D, Verhoef C. Software product line migration and deployment. *Software—Practice and Experience* 2003; **33**(10):933–955.
4. Finnigan PJ, Holt R, Kalas I, Kerr S, Kontogiannis K, Mueller H, Mylopoulos J, Perelgut S, Stanley M, Wong K. The portable bookshelf. *IBM Systems Journal* 1997; **36**(11):546–593.
5. Kazman R, Carrière SJ. Playing detective: Reconstructing software architecture from available evidence. *Journal of Automated Software Engineering* 1999; **6**(2):107–138.
6. Mendonca NC, Kramer J. Architecture recovery for distributed systems. *SWARM Forum at the Eight Working Conference on Reverse Engineering*, Stuttgart, Germany, October 2001. Available at: http://www.program-transformation.org/Transform/SwarmForum.
7. Sartipi K, Kontogiannis K. A graph pattern matching approach to software architecture recovery. *Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2001)*, Florence, Italy, 7–9 November 2001. IEEE Computer Society Press: Washington, DC, 2001; 408–419.
8. Eixelsberger W, Ogris M, Gall H, Bellay B. Software architecture recovery of a program family. *Proceedings of the International Conference on Software Engineering*, Kyoto, Japan, April 1998. IEEE Computer Society Press: Washington, DC, 1998; 508–511.
9. Bass L, Clements P, Kazman R. *Software Architecture in Practice* (2nd edn). Addison-Wesley: Reading, MA, 2003.
10. Tahvildari L, Kontogiannis K, Mylopoulos J. Quality-driven software re-engineering. *Journal of Systems and Software* 2003; **6**(3):225–239.
11. Bengtsson P, Bosch J. Scenario-based software architecture reengineering. *Proceedings of the 5th International Conference on Software Reuse (ICSR5)*, 2–5 June 1998. IEEE Press: Los Alamitos, CA, 1998; 308–317.
12. UML Home Page. http://www.uml.org [September 2004].
13. Stoermer C, O'Brien L, Verhoef C. Moving towards quality attribute driven software architecture reconstruction. *Proceedings of the 10th Working Conference on Reverse Engineering (WCRE)*, Victoria, Canada, November 2003. IEEE Computer Society Press: Washington, DC, 2003.
14. Clements P, Bachmann F, Bass L, Garlan D, Ivers J, Little R, Nord R, Stafford J. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley: Reading, MA, 2002.
15. Bachmann F, Bass L, Klein M. Deriving architectural tactics: A step toward methodical architectural design. *CMU/SEI-2003-TR-004*, Software Engineering Institute, Carnegie Mellon University, 2003.
16. Kazman R, O'Brien L, Verhoef C. Architecture reconstruction guidelines (3rd edn). *Technical Report CMU/SEI-2002-TR-034*, Software Engineering Institute, 2003.
17. Controler Area Network. http://www.can-cia.de [September 2004].

18. FlexRay. http://www.flexray-group.com [September 2004].
19. LIN. http://www.lin-subbus.org [September 2004].
20. *Automotive Engineering Magazine*, February 1999; 102–104.
21. Beecham M. Vehicle electrical wiring systems: A global market review—forecasts to 2007, August 2003. Available at: http://www.just-auto.com.
22. Mercer Management Consulting. http://www.mercermc.de/upload_material/tuv/3.pdf [September 2004].
23. Agne R. Global cyclic scheduling: A method to guarantee the timing behavior of distributed real-time systems. *Real-Time Systems* 1991; **3**:45–46.
24. Cscope Home Page. http://cscope.sourceforge.net [September 2004].
25. Gansner ER, North SC. An open graph visualization system and its application to software engineering. *Software—Practice and Experience* 1999; **30**(11):1203–1233.
26. Imagix Home Page. http://www.imagix.com [September 2004].
27. Agha G. *Actors: A Model of Concurrent Computation in Distributed Systems*. MIT Press: Los Alamitos, CA, 1986.
28. Alur R, Dill D. A theory of timed automata. *Theoretical Computer Science* 1994; **126**:183–235.
29. Buck JT, Ha S, Lee EA, Messerschmitt DG. Ptolemy: A framework for simulation and prototype heterogeneous systems. *International Journal of Computer Simulation* (*Special issue on Simulation Software Development*) 1994; **4**:155–182.
30. Rowson J, Sangiovanni-Vincentelli A. Interface-based design. *Proceedings of the 34th IEEE Design Automation Conference*, Anaheim, CA, 1997. ACM Press: New York, 1997; 178–183.
31. Vector Home Page. http://www.vector-cantech.com [September 2004].
32. AUTOSAR Home Page. http://www.autosar.org/find02.php [September 2004].
33. X-By-Wire Project Page. http://www.vmars.tuwien.ac.at/projects/xbywire [September 2004].

