Paper Number 09AE-0202

Integrated Functional and Non-Functional Design Verification for Embedded Software Systems

Christopher Ackermann, Arnab Ray
Fraunhofer Center for Experimental Software Engineering

Rance Cleaveland
Dept. of Computer Science, University of Maryland

Charles Shelton, Chris Martin
Robert Bosch North America, Research and Technology Center

ABSTRACT

This paper describes an approach to integrating functional and non-functional design verification for embedded control software. The method uses functional models, which have traditionally been used in functional verification processes, to drive non-functional verification also. This is achieved by defining strategies for extracting non-functional models, which contain structural and quantitative information about non-functional characteristics such as performance and modifiability, from functional ones. Non-functional verification tools may then be used on the resulting models to check that desired non-functional properties, such as ease of modification, are catered for in the design. An extended example involving the analysis of a model for modifiability is presented, as is tool support for extracting non-functional models from functional ones.

INTRODUCTION

An emerging best-practice in embedded-software engineering is to conduct extensive verification and validation (V&V) during the design phases of the development process, in order to catch errors and inconsistencies as early as possible. The widespread adoption of model-based development within the automotive industry has opened up broad new opportunities for design V&V, since the availability of executable models in notations like Simulink® / Stateflow®[1] and ASCET®[2] at design time enables extensive testing and analysis to be undertaken before any source code has actually been written.

In conventional model-based development workflows, design models are first constructed and then verified against the functional requirements (specifications of what the system must or must not do) using simulation, coverage-based testing, or model-checking. Requirements that relate to the non-functional aspects of system behavior (for instance resource usage, timeliness, modifiability) are typically reasoned about, if at all, in an ad-hoc qualitative manner.

The motivation for the work in this paper is to give embedded software designers tools to conduct rigorous, coordinated non-functional and functional design verification and validation. The proposed approach to achieving this goal relies on the use of functional models in notations such as Simulink to drive both functional and non-functional design analysis, using tools developed in the Computer Science research community to undertake

[1] MATLAB®, Simulink® and Stateflow® are registered trademarks of The MathWorks, Inc.
[2] ASCET® is a registered trademark of ETAS GmbH.

the latter. The core technical contribution described in this paper is an automated framework, and associated tools, for extracting non-functional design models from functional ones, so that appropriate verification of non-functional properties may be done at design time.

The rest of the paper is arranged as follows. The next section provides background in functional and non-functional modeling and verification, with a focus on the specific non-functional attribute of modifiability. Strategies for computing modifiability models from functional ones are explained, and an example and tooling described. The paper then concludes.

FUNCTIONAL AND NON-FUNCTIONAL MODELING AND VERIFICATION

This section briefly reviews functional modeling and verification and introduces the approach to non-functional modeling and verification that is used in this paper.

FUNCTIONAL VERIFICATION

A functional model of a system is a mathematical specification of the operational behavior of the system. It is typically encoded using an executable modeling notation. A functional requirement is a statement of what the system must or must not do, usually expressed in the form: if a given condition holds, then the system should respond appropriately. Functional verification consists of checking whether the functional model satisfies the functional requirements.

Simulink is widely used in the automotive industry for functional modeling. In Simulink, a model consists of a number of blocks (units of functionality) that exchange typed data (signals) through what are known as connections. These blocks are hierarchical; a block, in turn, may consist of other blocks and inter-connections between them. Blocks that contain other blocks are typically referred to as *subsystems*. In Figure 1, a Simulink diagram is shown of an abstracted body control application function. The top subsystem (X) handles all high level inputs and provides all outputs of the diagram. X processes the input signals and controls the subsystems Y and Z accordingly. Both Y and Z report on their current state by feeding back a signal to X.

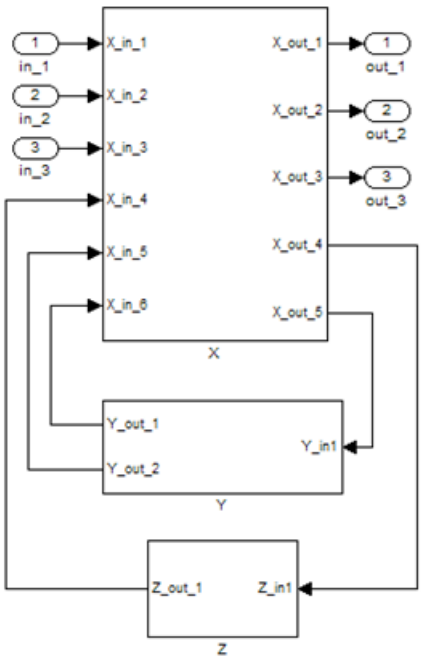

Figure 1: Example functional model given in Simulink.

Functional verification of models can be performed in a variety of ways: through exhaustive, guided simulation [1]; testing of the code automatically generated from the model [2];formal methods-based model checking or deductive reasoning [3]; and instrumentation-based model testing [4].

NON-FUNCTIONAL VERIFICATION

While functional modeling and verification has been studied extensively and is becoming widespread in practice, comparatively little attention has been placed on formally checking designs against non-functional requirements. Such requirements typically refer to desired structural or system-related properties, such as performance (will the software ensure desired timing?), security (does the software preclude unintended use?) and modifiability (is the software easy to modify and extend?).

In the Computer-Science research community, technologies such as ArchE [5] have been developed for reasoning rigorously about the non-functional aspects of software designs. ArchE focuses on software-architecture models as the design artifacts of interest; it allows designers to build such models and analyze them for different non-functional properties. In ArchE, basic software architecture information is enriched with quantitative information regarding the non-functional aspect of interest (modifiability, performance, etc.). The resulting *quality-attribute model* can then be analyzed to answer questions such as likely time to perform different modifications, expected performance, etc. Different quality-attribute models have been built and applied within ArchE [6], which is also the basis of Bosch's Rapid Architecture Prototyping Tool (RAPT). The remainder of this section describes ArchE's non-functional quality-

attribute models and requirements in more detail, with particular attention paid to the *modifiability* quality attribute [7] that is the focus in the remainder of the paper.

ArchE quality-attribute models contain a mixture of structural and quantitative data. The structural, or architectural, information consists of the following.

- **Responsibilities**: A responsibility is defined to be a unit of system functionality.
- **Relationships**: A relationship exists between two responsibilities if one depends, in any way, on the other.

For the modifiability quality attribute, a non-functional architecture is given as an *impact graph*. The nodes of the impact graph are the responsibilities and the edges are the relationships.

The quantitative information in ArchE models associates numbers to responsibilities and relationships. The interpretation of these numbers depends on the quality attribute being modeled: in the case of performance, the numbers may represent timing information, for example. In the modifiability model two types of quantitative information are used.

- **Cost of change (coc):** Each responsibility has an associated cost of modification (say *d*). Any change request that directly affects this responsibility is assumed to incur a cost of *d* person-days.
- **Probability of change propagation (pcp):** Because of the presence of relationships between responsibilities, a responsibility B may contribute indirectly to the total cost of modification of responsibility A if B connected via a relationship to A. The cost of propagation of a change from A to B is calculated by multiplying the probability of change propagation (p) from A to B with the direct cost to modify B.

The cost to modify a responsibility is typically used to label the corresponding node in the impact graph, while the probability of change propagation labels the relevant edge. In this way, an impact graph incorporates both structural and quantitative information.

In ArchE, non-functional requirements, and modifiability requirements in particular, are framed as *quality-attribute scenarios*. Quality-attribute scenarios consist primarily of a stimulus and a response – a stimulus in this case corresponds to a request for a specific modification, and a response describes an upper bound on the amount of time the modification should take. It should be noted that, in contrast with traditional functional requirements workflows, in which requirements assumed to be known in advance of fielding the system, non-functional requirements arise both before design and after deployment. In the case of modifiability, for example, designers may have some pre-deployment expectations about expected change requests and may conduct modifiability analyses on the associated quality-attribute scenarios during design time. The same modifiability models can be used post-deployment to assess the likely costs of change requests that arise after the system is fielded.

Figure 2 illustrates an impact graph derived from the functional model shown in Figure 1. Each top-level subsystem in this case is assumed to represent a different responsibility, while the connections among the subsystems reflect the relationships. Note that the responsibility X shares relationships with both Y and Z, but the latter two are not directly related to each other. The quantitative information is represented as annotations to the structural elements. The number in each responsibility represents the cost of change in man-days and the number at the relationships expresses the respective probability of change propagation. Later in this paper, automated mechanisms for computing this information are described.

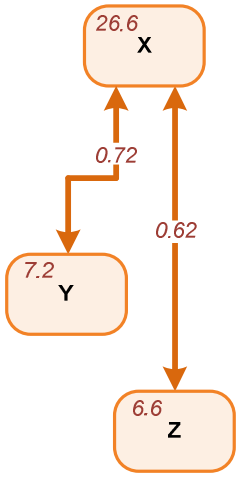

Figure 2: Impact graph capturing the structural and quantitative information of the example functional model.

The following describes a quality-attribute scenario, or more specifically a modifiability scenario, for adding a new input signal to responsibility X that also indirectly influences the signals to responsibilities Y and Z.

Stimulus: A new input shall be added to the X subsystem that carries a Boolean value. The new input signal represents an additional condition under which the subsystem Y and Z shall be activated.

Response Measure: Within 7 man-days.

Note that while this scenario primarily affects responsibility X, it also indirectly affects the responsibilities Y and Z. X may need to send additional signals to Y and Z based on the new input signal and both must be able to handle the modified signals. Thus

the change is expected to propagate from the responsibility X to the two other responsibilities Y and Z.

Given an impact graph with assigned costs of modification and probabilities of propagation of change, ArchE's reasoning framework for modifiability estimates the costs of changing one or more responsibilities. The framework calculates for each responsibility the probability that changing a specific responsibility will propagate to it. The average cost for this change can then be calculated by computing the sum of all the cost of change times the calculated probability

EXTRACTING IMPACT GRAPHS FROM FUNCTIONAL MODELS

The current state of non-functional reasoning frameworks fits uneasily with existing model-based development approaches, since the models required for functional and quality-attribute analysis are in different notations and must be constructed and managed separately as a result.

The contribution of this paper lies in defining a mapping from functional models given in Simulink to non-functional models in the form of impact graphs that ArchE can analyze for modifiability. Specifically, the structural and quantitative information required to populate the non-functional models is derived, with a minimum of user intervention, from the functional models. This serves the purpose of unifying the verification activities for functional and non-functional design attributes by integrating the two hitherto disconnected activities into a tightly coupled one that works on a unified model. In the workflow that this work is intended to support, designers would construct a single functional model and subject it to both functional and non-functional verification using the mapping procedure outlined in this section.

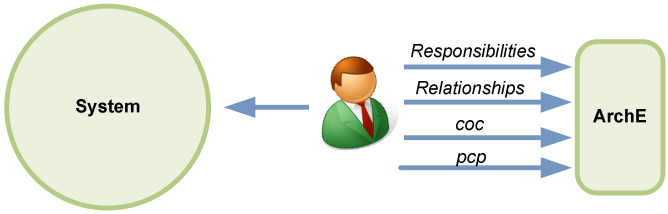

Figure 3: The user manually extracts functional and non-functional attributes from the system and feeds it into ArchE.

Figure 4: Our approach retrieves the functional and non functional attributes mostly automatically from the functional model with only little user intervention.

As noted in the previous section, impact graphs contain two sorts of information: structural and quantitative. The section describes how each of these types of information is computed from functional models.

EXTRACTING STRUCTURAL INFORMATION FROM FUNCTIONAL MODELS

The impact graph, as introduced in the last section, consists of an underlying graph whose nodes are responsibilities and whose edges are relationships. The first step in extracting an impact graph from a Simulink model is to identify these structural elements within the Simulink model itself.

The intuition underlying this work is to view the top-level subsystems of a given Simulink model as system components. In the non-functional domain, the analogues to these components are the responsibilities that, through data dependencies, together discharge the overall purpose of the system. Based on this intuition, the approach in this paper maps the top-level subsystems in the functional model to responsibilities in the non-functional model.

Similarly, the connections between subsystems in the functional model have relationships as their analogues in the non-functional domain. More specifically, if subsystem A is mapped to responsibility R and subsystem B to responsibility S, then a relationship exists between R and S if and only if there exists a connection between A and B in the functional model. Note that the directionality of the connection is not taken into account: if A reads inputs from B or writes outputs to B there is a connection between R and S. The reason for this has to do with a sometimes unappreciated bi-directionality of modifiability: if one modifies the data type a variable, for example, then both the statements that write to that variable as well as the ones that read from that variable must be modified.

The conceptual mapping of the structural elements of the functional model to the non-functional architecture is summarized in the Table 1.

Table 1: Mapping between non-functional and functional structural elements.

Non-Functional Structural Elements	Corresponding Functional Elements
Responsibilities	Subsystems
Relationships	Connections

EXTRACTING QUANTITATIVE INFORMATION FROM FUNCTIONAL MODELS

The quantitative information required to populate impact graphs is obtained from two metrics computed on the functional model.

- **Modifiability Metric:** This metric captures how expensive in terms of man-days it is to modify a given subsystem.
- **Connectivity Metric:** This metric measures the degree of connectedness between two subsystems.

These metrics are defined below. Table 2 summarizes the relationship between these metrics and the associated quantities used to annotate impact graphs.

Table 2: Mapping between non-functional and functional measures.

Impact Graph Quantities	Corresponding Functional Model Measures
Cost to Change a Responsibility (coc)	Modifiability Metric
Probability of Change Propagation (pcp)	Connectivity Metric

Modifiability Metric

In the modifiability framework, each responsibility has an associated cost of change (coc) number that captures the estimated number of man-days considered necessary to perform any kind of modification to it. The modifiability metric provides a means of deriving these coc numbers based from functional models.

No direct method to compute such a metric is known to exist in research literature; the closest approach was that followed by Vitkin et al [8], who derived structural metrics from Simulink models for the purpose of estimating auto-coding effort. In the following discussion, their approach is first summarized, and then the customizations needed to reflect an appropriate metric of modifiability are detailed.

Autocoding Effort

The calculation of the auto-coding effort for a functional model is based on complexity values for the blocks and connections in the model.

A functional model consists of different types of blocks. Each block has a set of parameters. When auto-coding a functional model each parameter must be set by the user, thus contributing to the total effort. This is accounted for by assigning a complexity value to each block (BSON) which is defined to be equal to the number of parameters the block has. Based on the individual complexity values for each block, the total block complexity of a given model is the sum of all individual block complexities.

$$BC = \sum_{k=1}^{N} BSON_k$$

BC = Total block complexity for a functional model.
N = Number of blocks in the functional model.
$BSON_k$ = Individual block complexity of block *k*.

Regarding connections, only two types of connections are distinguished in Vitkin's work: connections that carry binary signals and those that carry non-binary signals. While auto-coding binary connections is trivial, the same task for non-binary connections requires more effort, as it is necessary to ensure that the variables in the generated code are precise enough and can hold large enough values. The complexity for binary connections are ignored (MCN=0) and all non-binary connections have a complexity value of 1 (MCN=1). In the same fashion as for the blocks, the total connection complexity is the sum over all individual line complexities.

$$CC = \sum_{l=1}^{L} MCN_l$$

CC = Total connection complexity of functional model.
L = Number of connections in the functional model.
MCN_l = Individual connection complexity of connection *l*.

The total auto-code complexity of the functional model is a weighted sum of the total block complexity and the total connection complexity. Both the block and the connection complexities are weighted using the factors K1 and K2, respectively.

$$MC = K1 * BC + K2 * CC$$

MC = Total model complexity
K1 = Weight for total block complexity
K2 = Weight for total connection complexity

The value of the tuning coefficients K1 and K2 depend upon the type of auto-generated code and other organizational factors. By adjusting the coefficients, one can take into account characteristics that are specific to the generated code and thus increase the accuracy of the result. Vitkin et al. did not provide any guidelines as to how to determine these factors but mentions that these rely on expert judgment.

Defining the Modifiability Metric

Several modifications to the original formulation of Vitkin numbers are necessary in order to adapt them to the domain of modifiability. One the one hand, the complexity values for blocks and connections that express the effort for auto-coding the respective element do not appropriately reflect the effort for modifying it. Further, the values for the coefficients K1 and K2, i.e. the

weights for blocks and lines, also need to be appropriately defined for the modifiability domain. The basic principle for calculating the modifiability metric of modifiability however remains the same as that used by Vitkin et.al: the modifiability measure for a subsystem is calculated by accumulating the complexities of all blocks and lines that are contained in it.

Block Complexity. When a new basic block is added to a functional model, there are a number of tasks that need to be executed. First, the engineer must decide which basic block is to be added. This necessarily means understanding the semantics of the block. Second, the engineer needs to determine the part of the model in which the block is to be inserted. Third, the engineer needs to locate the basic block in the library, drag it into the model and connect all its input and output ports. Lastly, the functional model must be tested after the block was added to it. Based on the above observations, a complexity schema that assigns complexity values to each block type was developed.

A list was created with 30 basic blocks that were used in the models on which this approach was applied (for the purposes of this work, effort data for 74 different models constructed as part of a body-electronics modeling project were used). Each block was then evaluated regarding the difficulty and effort of the tasks for creating that block. The complexity of the block was then rated on a scale from 1 to 10 with 1 being the least complex and 10 being the most complex block (the term BCN is used to refer to this number). This produced a schema that rated the complexity of each block in relation to other blocks. For instance, the "outport" block was considered to be of little complexity and was hence assigned a complexity of 1. The "switch-case" block requires defining a number of Boolean expressions and was assigned a complexity value of 5.

Table 3: List of sample blocks and the complexities that were assigned to them.

Block	BCN
Subsystem	1
Outport	1
Costant	3
Inport	1
Merge	2
Memory	3
Switch-Case	5
Action-Port	3
If	5

Connection Complexity. The connection complexity, CCN, for every connection was then set to 1. When creating a new connection, one does not need to spend more effort than on non-binary connections. Thus, in contrast to the Vitkin approach, no distinction is made between binary and non-binary connections and an equal complexity value is assigned to both of them. The effort for determining which blocks and ports to connect is already accounted for in the BSON numbers. The low connection complexity (i.e. 1) expresses only the effort for adding the connection.

Although these values were determined by engineers who were already familiar with the design of functional models, they might not reflect the situation in different contexts or domains.For example, the numbers will be higher for novice modelers than for experts. To adapt to a different environment, one can simply modify this schema according to particular needs and preferences.

Coefficients. Using the coefficients K1 and K2 provided in the example by [8] resulted in an effort estimation that was much higher than the experimental data collected in the course of building the 74 models alluded to above. Accordingly, adjustments were made: 0.02 for K1 and 0.001 for K2 produced the results that coincided most closely with recorded effort.

Modifiability Metric Formula. In a last step, the formula for computing the total complexity of a subsystem must be given. The modifiability of a subsystem is calculated by applying the formula for the total complexity to the blocks and connections that are contained in the respective subsystem. The metric of modifiability for each subsystem is as follows:

$$MM = K1 * BMC + K2 * CMC$$

BMC stands for the total block complexity and CMC stands for the total connection complexity. The block complexity is the sum of block complexities of the blocks contained in the subsystem. Likewise, the total connection complexity is the sum of all connections contained in the subsystem.

$$BMC = \sum BCN_k$$

$$CMC = \sum CCN_l$$

Here k ranges over the blocks in the subsystem, while l ranges over the connections. The example below (Figure 5) shows a high level subsystem having three inputs and one output signal.

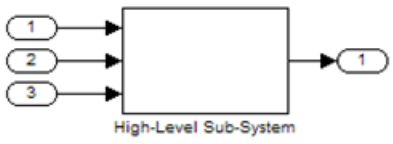

Figure 5: High level subsystem of a functional model.

Figure 6 shows the blocks and connections that are contained in that high level subsystem. Each block is annotated with its complexity (BCN) value. The modifiability value of the high-level subsystem is

computed based on the complexity of the subsystems and connections it contains.

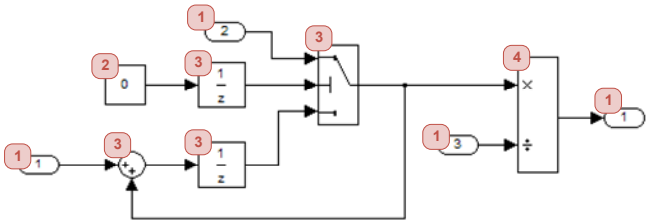

Figure 6: The subsystems inside the high level subsystem annotated with their respective complexity values.

The total block complexity is determined by multiplying the count of each subsystem with its complexity value and then summing over these values.

$$BC = 3*1 + 2*3 + 1*3 + 1*3 + 1*2 + 1*4 + 1*1$$
$$= \mathbf{22}$$

Since each connection has a complexity value of 1, the connection complexity is the sum over all connections.

$$CC = \mathbf{10}$$

The modifiability of the high level subsystem is then computed by adding up the weighted block and connection complexity.

$$MM = 0.02*22 + 0.001*10$$
$$= 0.44 + 0.01$$
$$= \mathbf{0.45}$$

Modeling the high level subsystem is estimated to require 0.45 man-days according to our model.

Connectivity Metric

The connectivity metric assigns probability of change propagation numbers (i.e. edge weights of the impact graph) from design models rendered in Simulink. The metric is based on the following intuition: the greater the number of connections between two Simulink subsystems, and the deeper they are, the greater the chance that modifying one will lead to the modification of another, i.e. the greater will be the probability that change propagates between their corresponding responsibilities.

The connectivity metric is based on the notion of signal propagation. A change that affects one block propagates to its neighbors through modifications of the signal that travels on the connection. For instance, a change to a subsystem (called hereafter the *source*) might cause a change to the range of values, the precision, etc. of a signal that it outputs. The signal then travels to a neighboring subsystem (the *target*), which may also require modification as a result. The reverse is also true. A modification to a target subsystem might cause modifications to the signal, which in turn might require changing the source subsystem to adapt to the modified signal.

If the target subsystem is atomic (i.e. does not contain any other blocks), it is said to be fully impacted by the signal, i.e. the connectivity between source and target is 1 (100%). When the target block is contains multiple blocks, it needs to be determined to what extent these are affected by the signal in order to estimate the total impact to the target subsystem. The illustration in Figure 7 shows such a set-up. Subsystem a is the source subsystem, and it is connected to the target subsystem b, which contains blocks c, d, e, f, g, h, and i. The signal that is output by the source subsystem a travels to subsystems contained in b, in this case c. Since the signal is an input to c, it also affects its output signal, which is input to subsystems d and e. The signal propagates in the same way to subsystems f and g. However, the signal never reaches the subsystems h and i.

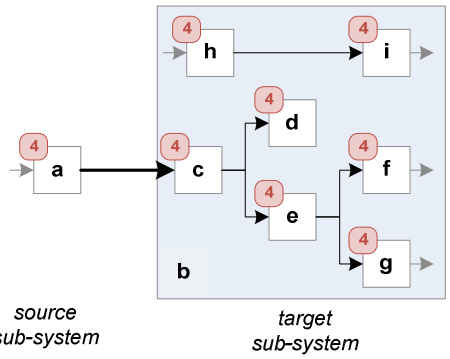

Figure 7: Conceptual view of functional model in which the external connection emanates from subsystem a and impacts parts of the target subsystem b.

This intuition is used to define the degree of impact that can propagate though a connection, i.e. its connectivity. First, we assume that all blocks that are affected by the connection (i.e. by the signal traveling on it) have been identified i.e. the *affected blocks*. They can be determined by transitively traversing the connections touch an affected subsystem. In the example in Figure 7, the affected blocks are c, d, e, f, and g.

Previously a complexity value for each subsystem type was defined. Now the *affected complexity* is defined to to be the sum of the complexities of all affected blocks.

$$AC = \sum_{affected\ \ blocks} BCN_i$$

The affected complexity in the example is the sum of the complexities of the blocks a-f, i.e. 20.

The affected complexity expresses how much complexity in a subsystem is affected by a connection. This impact measure will be used subsequently instead of simply counting the affected subsystems in order to account for the different amount of effort that is needed to modify subsystems of different complexities.

In the next step we compute the connectivity as the percentage of total complexity that is affected by the connection. The total complexity is simply the sum of the complexity values of all the blocks it contains. For instance, the total complexity of subsystem *b* is 28.

CN = AC / total subsystem complexity

In summary, to compute the connectivity of a connection, we first determine all subsystems in the target subsystem that are affected by it, compute the affected complexity, and then divide it by the total subsystem complexity.

In the example, we have determined the affected subsystems to be *a-f*. The affected complexity is 20 and the total complexity is 28. The connectivity of the relationship in direction from *a* to *b*, therefore, 20/28 = 0.71 = 71%.

A simple extension to the approach enables it to handle multiple connections between subsystems. The affected subsystems are determined by identifying all subsystems that are affected by a signal from any of the external connections that originate in the source subsystem and end in the target subsystem. Computing the affected complexity and the connectivity is then done in the same fashion as described above for single connections.

NON-FUNCTIONAL VERIFICATION – AN EXAMPLE

This section illustrates the concepts defined in the previous sections with the example functional model introduced previously. The figure below (Figure 8) shows the top-level view of the model. Each step of extracting the structural and quantitative information from the functional model and the way the impact graph is updated in each step is illustrated in the following discussion.

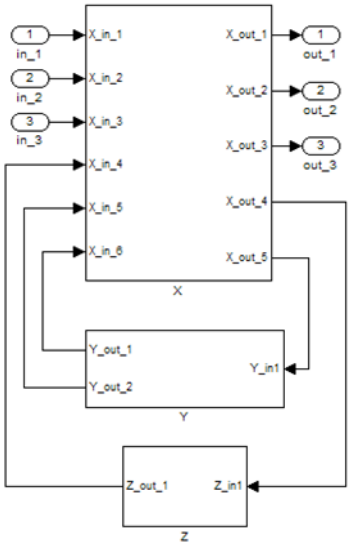

Figure 8: The high level view of the example model.

The steps in sequence are:

1. Extracting the responsibilities from the functional model.
2. Calculating the cost of change from modifiability metric.
3. Extracting the relationships from the functional model.
4. Calculating the probability of change propagation from the connectivity metric.

Each is discussed in turn below.

EXTRACTING RESPONSIBILITIES

The functional model of the example has three top-level subsystems: X, Y, and Z. Each of these subsystems will be represented in the modifiability model as responsibilities. Figure 9 shows all the responsibilities for the example system that are extracted from the functional model.

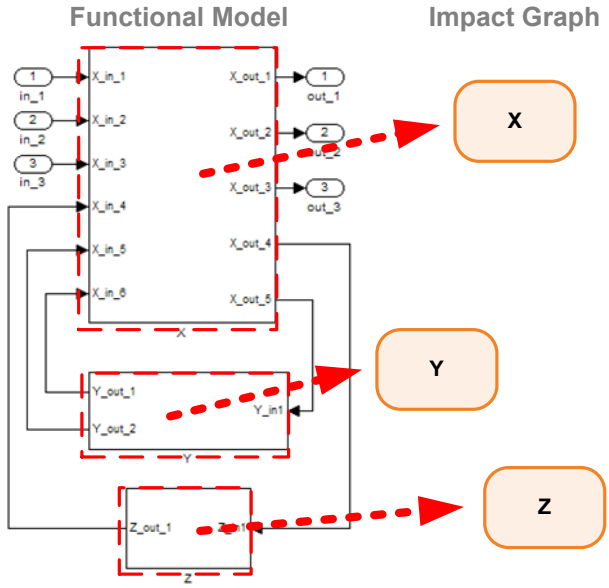

Figure 9: The responsibilities of the modifiability model on the right are created based on the subsystems in the functional model on the left.

COMPUTING THE COST OF CHANGE

Figure 10 shows how the modifiability values for each responsibility in the example model are computed.

What the impact graph shows is that the cost of modifying the functionality represented by the responsibility X is significantly higher than both Y and Z. More precisely, a modification to responsibility X is estimated to take 26.6 man-days, while modifications to Y and Z takes below 7.2 and 6.6 man-days, respectively.

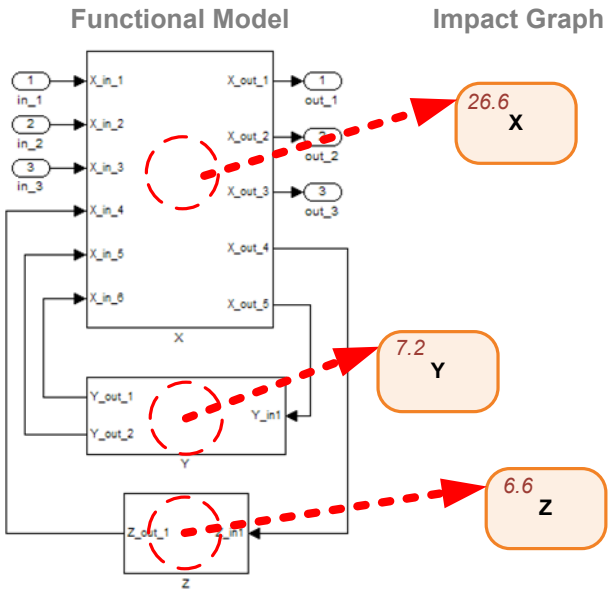

Figure 10: Mapping of responsibilities in the impact graph to subsystems in the functional model. The coc values for each responsibility have been computed.

EXTRACTING RELATIONSHIPS

Relationships in the impact graph express data dependencies among responsibilities. A relationship has a source and a target responsibility, i.e. a relationship is a directed dependency; however, in the case of modifiability, every relationship also has its inverse included, reflecting the bi-directionality of modification. To define a relationship one has to specify the following parameters:

The mapping of relationships to external connections is illustrated in Figure 11.

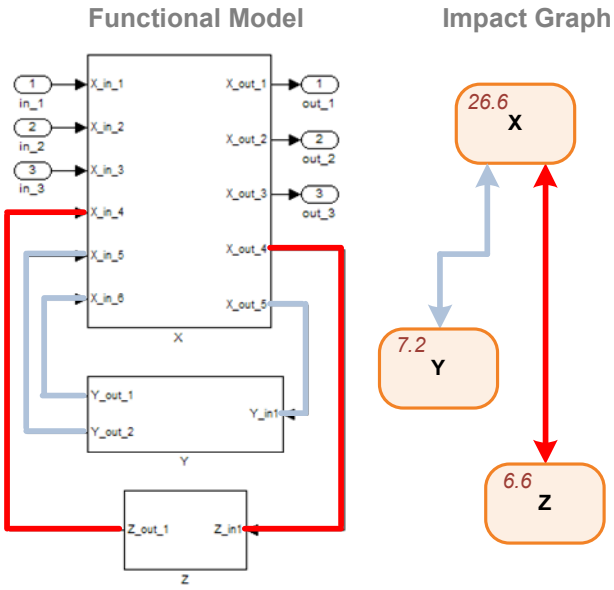

Figure 11: Result of this step. Blocks are mapped to responsibilities and connections are mapped to relationships.

After computing the set of connections that represent the relationships in the functional model, the following step calculates the probability of change propagation (pcp) attribute for the relationships.

COMPUTING THE PROBABILITY OF CHANGE PROPAGATION

The probability of change propagation can be directly derived from the connectivity metric. The source and the target responsibilities are mapped to a source and a target subsystem, respectively. The probability of change propagation of that relationship equals the metric of connectivity for all connections from the source to the target subsystem.

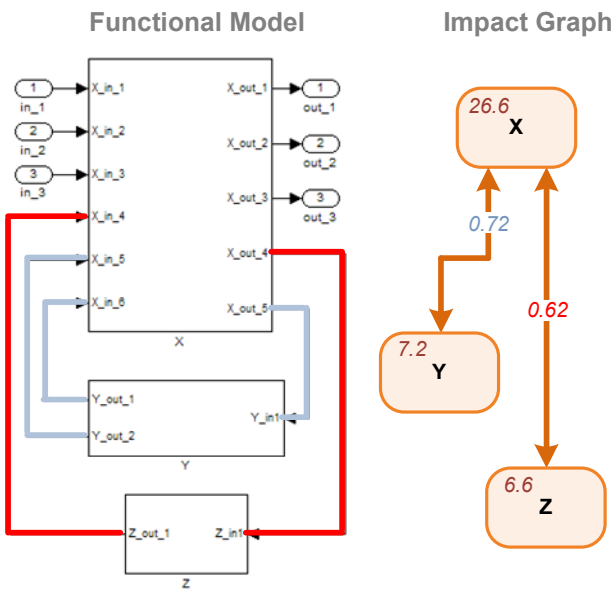

Figure 12: Computation of the probability of change propagation values from connectivity values of connections in the functional model.

The impact graph in Figure 12 is complete as it contains the structural and quantitative information that is needed to evaluate quality attribute scenarios. This section has illustrated how the information for building the impact graph can be extracted from functional models. The next section will discuss how quality-attribute scenarios can be evaluated using this information.

TOOL SUPPORT

In order to automate the extraction of non-functional information from functional models, a tool that supports the initial responsibility mapping and automatically computes relationships, cost of change and probability of change propagation has been developed.

The tool is implemented in Java as a plug-in for the popular development environment Eclipse developed by IBM. It consists of an importer for functional models in the Simulink notation, a graphical representation of the impact graph and an user interface to show details about the mapping between functional model and impact graph. Furthermore, the tool provides an intuitive way for specifying and evaluating quality attribute scenarios. The actual evaluation is done by the ArchE reasoning framework. Figure 13 shows the architecture of the setup. The user uses only the graphical user interface of the tool to provide her input and observe the output. The tool extracts all necessary information automatically from the functional model and communicates with ArchE to evaluate the scenarios.

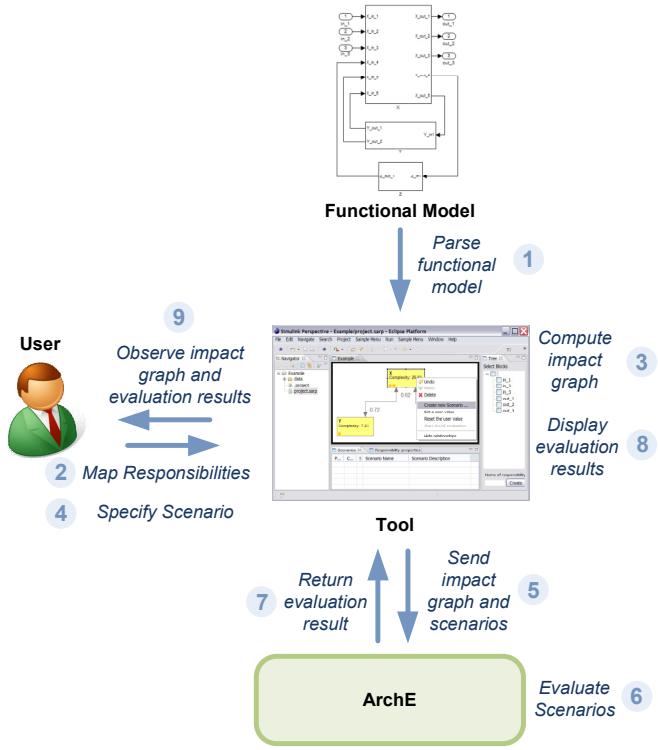

Figure 13: Architecture and information flow for evaluating modifiability scenarios using tool support.

The user begins by specifying the functional model on which non-functional properties are to be verified. She does this by choosing a Simulink model file. The tool then parses the block/subsystem hierarchy of the functional model and shows it in a tree structure. On the highest level of the tree structure are also the high level blocks and subsystems of the functional model.

The user selects the subsystems that should be represented as responsibilities in the impact graph.By default, the name of the responsibility is chosen to be the name of the subsystem that is mapped to it but can be modified. Upon creating the responsibility, the tool automatically establishes a mapping between all blocks and internal connections that are contained in the chosen subsystems. Also, the tool immediately computes the cost of change (coc) for the new responsibility. In doing so, it implements the approach presented in previous sections using a settings file that stores all block complexities. Customizing the coc computation to suit a different domain is only a matter of changing the settings file.

If there are at least two responsibilities in the impact graph, the tool determines the relationships and the probabilities of change propagation thereof. It parses the functional model, determines all external connections, and maps them to the relationship. Using a graph traversal algorithm, it determines all blocks that are affected by a relationship and then computes the pcp based on their complexity and the total complexity of the responsibility. The relationships are represented as arrows in the impact graph and are annotated with the pcp values.

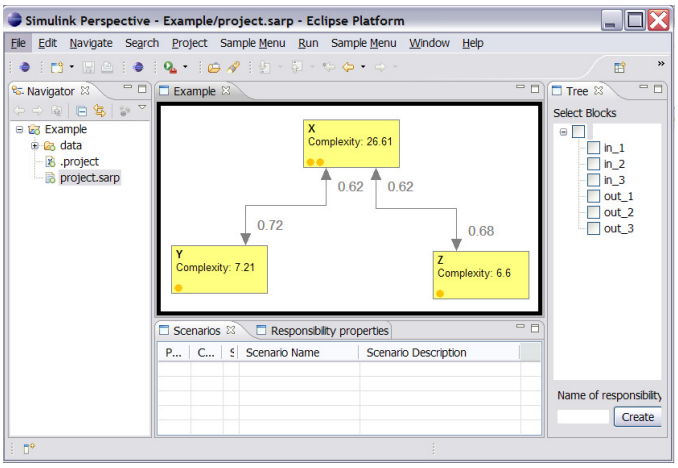

Figure 14: The user interface of the tool. The center shows the final impact graph of the example functional model.

CREATING SCENARIOS

After the modifiability model has been built, the user can provide the non-functional requirements that are to be checked on the models. These non-functional requirements are written in the form of quality attribute scenarios. The responsibilities on which the modifiability is to be evaluated is input to the tool. A window then opens up in which other scenario properties can be specified. The modifiability value is mandatory.

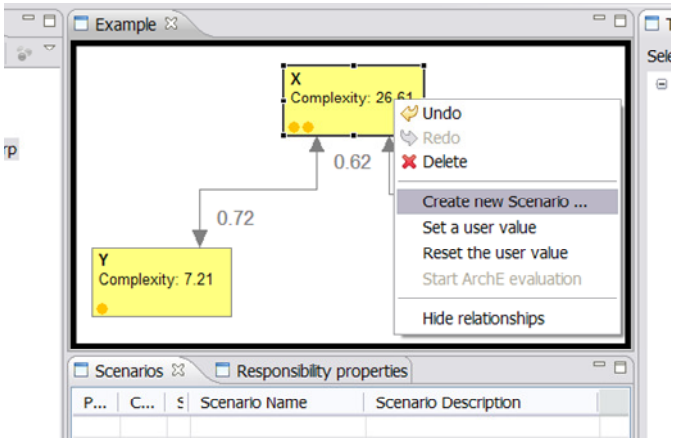

Figure 15: The user selects one or more responsibilities and creates a scenario.

The actual verification of the modifiability scenarios is conducted by ArchE. The user starts the verification process by clicking a button. The tool sends the impact graph and scenarios information to the ArchE back-end and awaits a response. Upon finishing the evaluation, ArchE sends the results back and the tool visualizes violations in the user interface. It displays an estimate for the modifiability of the current scenario, i.e. the number of man-days needed to implement the scenario. An icon indicates whether the quality attribute scenario holds. In the example, the scenario (the actual scenario is provided in the Section) is violated since the change can not be implemented within 7 days as specified.

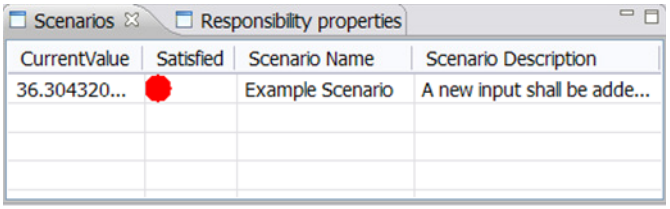

Figure 16: Visualization of the evaluation results in the tool.

If ArchE has determined that a scenario is violated, the user has several options for resolving the problem. Two of them are lowering the complexity value of the affected responsibility and lowering the probability of change propagation. Of course, in order to achieve either one, refactoring is necessary, which can potentially be time consuming. The tool allows the user to simulate the refactoring by allowing the user to provide a custom value for both the responsibility complexities and the pcp's. The impact graph and the scenarios can then be evaluated with the custom values. This is a quick solution for finding a set up that satisfies the quality attribute requirements without actually implementing the change.

VISUALIZATION

Depending on the size of the functional model, the impact graph might become quite large and the diagram might be too crowded with responsibilities and relationships. A few graphical features are supposed to address this issue. First, to allow the engineer to focus on certain parts of the model, the user has the option to hide all relationships. She can then click on one of the responsibilities to show all of its incoming and outgoing relationships. Second, colored dots indicate the number of relationships of each responsibility and their probability of change propagation. Each relationship that is connected to the responsibility is represented as a dot. The dots function as traffic lights that indicate if the pcp of the respective relationship is low (green), medium (orange), or high (red). This highlights responsibilities that are affected to a large extent by other responsibilities and should support the user in quickly identifying the areas of the model that might cause problems when evaluating a modifiability scenario.

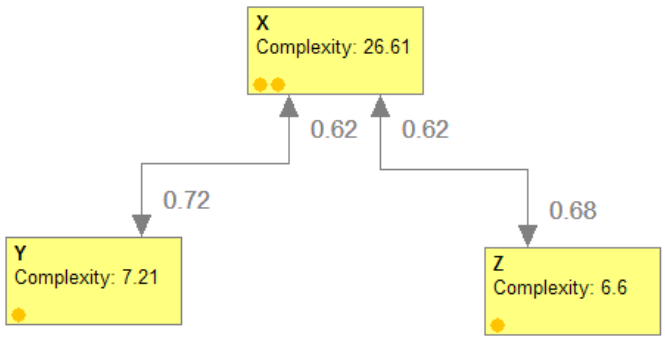

Figure 17: Impact graph of the example functional model as produced by the tool.

Figure 17 shows the final impact graph that was created based on mapping the subsystems X, Y, and Z to a responsibility. The relationships and all quantitative characteristics were computed automatically.

CONCLUSIONS AND FUTURE WORK

This paper has shown how non-functional modifiability models can be extracted from functional models given in Simulink. The modifiability models can be used to estimate the effort needed subsequently to modify these models and are thus intended for use in assessing how "modifiable" the functional model is. The extraction procedure relies on viewing subsystems in the functional model as components, or responsibilities (the term used in the paper), and connections between subsystems as dependencies. The information about blocks and connections with subsystems are then used to quantify the complexity of modifying individual subsystems and the degree to which modifying one subsystem induces changes in a neighboring subsystem. Effort data gleaned from creating 74 Simulink models as part of a body-electronic modeling effort was used to calibrate the procedure and validate the results.

Future work will involve further fine-tuning of the calibrations in the extraction process, and further development of a combined functional / non-functional design-time verification workflow. Experimenting with the tool on ongoing modeling efforts will also provide insight into the utility of non-functional design verification.

REFERENCES

1. Bakshi, A. MILAN: A Model based integrated simulation framework for design of embedded systems. *ACM SIGPLAN 2001 Workshop on Languages, Compilers, and Tools for Embedded Systems*.

2. Takahashi, J. and Kakuda, Y. 2002. Extended Model-Based Testing toward High Code Coverage Rate. *Proceedings of the 7th international Conference on Software Quality* (June 09 - 13, 2002). J. Kontio and R. Conradi, Eds. Lecture Notes In Computer Science, vol. 2349. Springer-Verlag, London, 310-320.

3. Clarke E. Grumberg O, Pereld D. Model Checking. The MIT Press.

4. Ackermann C., Ray A., Cleaveland R., Heit J., Shelton C., Martin C.. Model-Based Design Verification: A Monitor Based Approach. *Society of Automotive Engineers (SAE) World Congress 2008*, Detroit, USA.

5. Bachmann, Felix; Bass, Len; & Klein, Mark. Preliminary Design of ArchE: A Software Architecture Design Assistant (CMU/SEI-2003-TR-021).

6. Bass, L., Ivers, J., Klein, M., Merson, P., and Wallnau, K. 2005. Encapsulating Quality Attribute Knowledge. *Proceedings of the 5th Working IEEE/IFIP Conference on Software Architecture* (November 06 - 10, 2005).

7. C. Shelton and C. Martin, "Using Models to Improve the Availability of Automotive Software Architectures", *ICSE Workshops SEAS '07*, IEEE, Location, 20-26 May 2007, pp. 9-19.

8. Lev Vitkin, Susan Dong, Rick Searcey and Manjunath BC. "Effort Estimation in Model-Based Software Development", *Society for Automotive Engineers World Congress 2006.*

9. Bachmann, Felix; Bass, Len; Klein, Mark; & Shelton, Charles. "Designing Software Architectures to Achieve Quality Attribute Requirements" (153-165). IEEE Proceedings on Software, August 2005.