Integrating graphical and natural language specifications to support analysis and testing

Christopher L. Robinson-Mallett
Mobility Division, Siemens AG, Berlin
c.robinson-mallett@siemens.com

Robert M. Hierons
Brunel University London,
Uxbridge, United Kingdom
rob.hierons@brunel.ac.uk

***Abstract*—The ongoing trend towards distributed development activities causes a growing need for specification activities and techniques. Each component leads to a large number of specification documents being exchanged, change managed and committed. The quality of the specifications influences the timing, costs and success of the development task. However, the quality of such specifications is often far from optimal, exhibiting gaps, inconsistencies, redundancies, and unbalanced structures. At every release or delivery milestone, acceptance and integration testing take place. Therefore, test-cases have to be created from the requirements exchanged. This paper presents a model-based approach for improving the quality of comprehensive requirements sets. The presented solution is based on a combination of a graphical notation and natural language and can be used to drive model-based testing. The approach has been implemented using state-of-the-art tools. We present experience from field application in the automotive industry.**

I. Introduction

The ongoing trend towards distributed development activities amongst world-wide teams and the integration of the respective results causes a growing need for specification activities and techniques. Each component leads to a large number of specifications being exchanged, change managed as well as integration and tests committed between the leading and the supplying development teams.

While distributed development projects are not the topic of this paper, our experience with them motivated the work described. The quality of specifications has a significant impact on the efficiency and risks of distributed development activities [1] such as the quality of development results, change management, coordination activities, and testing. Furthermore, the development of dependable software demands traceable processes and provably complete and correct results.

This paper aims at reducing risks of distributed development activities, presenting a method to create and maintain product specifications that are precise, formally structured, and that product engineers find equally easy to understand and to use.

Despite their potential benefits, formal languages (FL) are not widely used because the required expertise is often not present. Instead, natural language (NL) in combination with graphical notations (GN) is a familiar choice for the engineer, with the use of GN helping reduce the size of the NL specification and providing the potential to automate analysis, transformation and testing. An additional benefit is that often there is domain specific technical NL and GN that is common to all groups. Thus a combination of NL and GN is practically the de facto standard specification language in distributed development scenarios, although significant drawbacks exist:

1) NL can leave much room for interpretation;
2) NL is local, e.g. a specification written in German is only useful for German speaking groups (this can also affect models);
3) It is difficult to maintain NL and keep it precise, consistent and free of redundancies;
4) Automated processing of NL is difficult;
5) Over time and changes, NL and embedded or referred GN tend to diverge in meaning;
6) GN alone may contribute precise, but only partial information, which demand NL requirements to tell the reader how to interpret it.

Our experience in industrial projects indicates that structure has a significant impact on requirements quality. An unclear structure may contribute to redundancy and so to overly long specifications. Additionally, inconsistencies may be created, if additional requirements deviate from those already in the set.

In the context of this paper a specification is a set of statements (requirements) describing aspects of the product under development. Such statements may possess a type, for instance to denote priority or use of certain information, e.g. differentiate requirements from heading or secondary information. A specification targets a stage of the development process, e.g. product definition, analysis, design, implementation, testing, integration, delivery. Furthermore, specifications may have a certain scope, describing the product as a whole or only parts.

The proposed approach aims to create a model-integrated specification that closes the gap between NL and GN through formalising the structure of the GN specification, while preserving the benefits of NL. We have found that a major benefit is that the resulting specifications provide almost optimal cohesion. In addition, it is possible to apply formal analysis and test generation techniques to the resultant specifications. This paper extends previous work [6] by providing additional information about the formalisation used and by describing a semi-automatic approach to test generation along with associated coverage criteria.

This paper is structured as follows. Section II gives the running example and Section III describes the proposed approach. Section IV explains how the resultant combination of GN and NL can drive testing. Section V describes our experiences of using the proposed approach and associated tool. Section VI

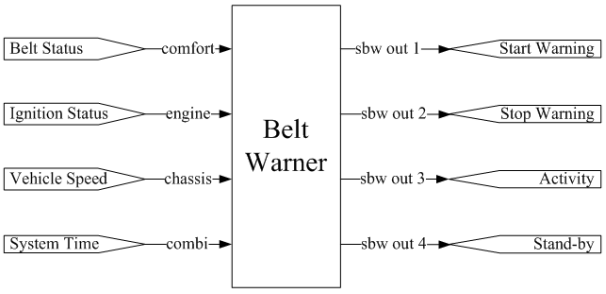

Fig. 1. Block diagram belt-warner example

then describes related work and Section VII summarises the work.

II. Belt-warner example

The belt-warner example is a simplified extract of the specification of a real driver assistance function that warns the driver at critical speeds that the safety belt is not tightened. The architecture of the system is described using a subset of Block Diagrams [2] [3]. The system behaviour is specified using a sub-set of statecharts [4].

A. Architecture of the belt-warner example

In the belt-warner example a block diagram specifies the system architecture. Block diagrams are graphical structures consisting of hierarchical blocks, signal flows, signal entry points and signal exit points. A block, represented as a rectangle, may contain a behaviour. Hierarchical blocks may possess architecture and behaviour. Signal flows, represented as edges in the block diagram, specify message routing between blocks. Signal sources and sinks, represented as wedged rectangles, are special blocks used to define system interfaces.

The example architecture of a simple belt-warner in Figure 1 contains a single block, 4 signal sources and 4 signal sinks. The sources define the input interface, which consists of signals Belt Status, Ignition Status, Vehicle Speed, and System Time. The sinks define the output interface, which consists of signals Start Warning, Stop Warning, Activity, and Stand-by.

B. Behaviour of the belt-warner example

Statecharts is a graphical notation for extended, hierarchical finite state-machines that is applicable to the specification of state-based system behaviour. In the context of the presented approach, a sub-set of statecharts has proved to be useful and will be explained in this section. A complete and detailed description of statecharts is provided in [4].

The nodes of a statechart represent a finite set of system states, while the edges represent transitions between these states. A statechart may process a finite set of inputs and produces a finite set of outputs. A statechart may be extended with data of arbitrary number and types.

A state may contain a finite set of sub-states, one being the initial state, i.e. when changing into the state the initial sub-state is entered. A transition into a hierarchical state (a state containing sub-states) ends either implicitly in the initial or explicitly in one of the sub-states. An atomic state does not contain sub-states. A state-invariant specifies conditions that must hold when the system is in the corresponding state.

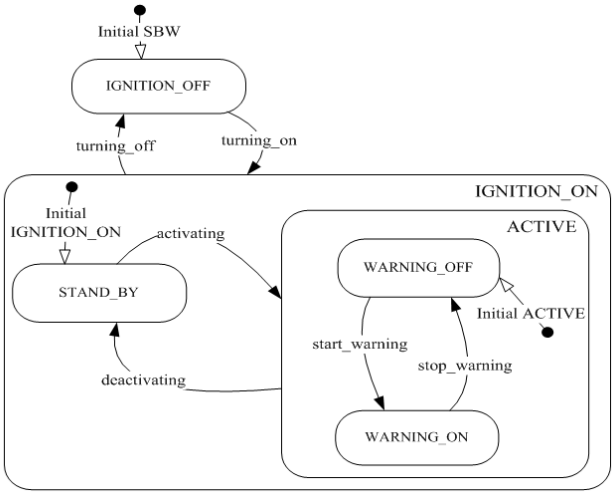

Fig. 2. Statechart for the belt-warner

A state transition has an initial and a final state. A group transition starts from a hierarchical state S_h; it can be triggered from any sub-state of S_h. The trigger of transition t specifies conditions on inputs and data under which t executes. Furthermore, in a transition it is possible to specify data operations and outputs. We use predicate logic to specify the above.

The statechart in Figure 2 contains 4 atomic states, 2 hierarchical states and 9 transitions, of which 3 are initial transitions and 4 are group transitions. It describes the system behaviour in use, when turning it on/of, activating, deactivating the function and situations in which the belt warning is started or stopped.

III. Model-Integrated Requirements Specification

As previously discussed, the proposed approach is based on integrating GN and NL, which is achieved by mapping each 'basic model element' of the graphical model to a corresponding textual section (called a *chapter*).

This process is applicable to a variety of modeling notations, if these can be mapped into finite structures, such as graphs. Given an NL specification of a system (for the ease of explanation initially without GN parts), we create a model-integrated specification as follows.

1) Break down the specification into atomic statements.
2) Give each atomic statement a type of either information or requirement.
3) Create a GN of the system's high-level architecture.
4) Create a NL chapter structure based on the architecture:
   - each node in the GN relates to a chapter,
   - each edge in the GN relates to a chapter.
5) Relocate atomic statements into the chapter structure.
6) Analyse residual atomic statements and either discard or redesign GN and go back to 3.

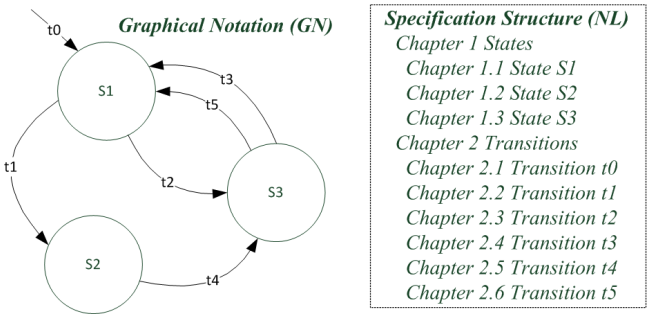

Fig. 3. Example model-integrated finite state machine

7) Review each chapter and eliminate incompleteness, inconsistencies and redundancies.
8) Optionally, complement NL specification with conditional statements defining feasible paths and further system properties, e.g. state invariants.

This results in a semi-formal specification in the form of a chapter structure; specification statements remain in NL. For instance, if the GN is a finite state machine, then the model elements are states and transitions, and a chapter is created for each of these. Figure 3 presents the example of an FSM and the NL structure generated.

In this paper we outline how GN in the form of labelled directed graphs (LDGs) can be used. The GN part may have a specific (separate) semantics associated with the language originally used. We need to retain this information to correctly interpret different languages such as statecharts and block diagrams. We have developed a set of semantics for automotive controller functions but anticipate that different semantics might be used in other domains. In practice, we have found LDGs to cover a large and useful variety of modelling notations. However, there might be notations, e.g. sequence diagrams and use-case diagrams, where there is no natural mapping to LDGs. In such cases we suggest that different GN-notations are used, e.g. multi-graphs.

In practical software and systems development, the variety of GN used often depends on the domain (e.g. statecharts, activity diagrams, Petri nets, block diagrams, or fault trees). To make the model-integration approach applicable to a maximum number of GNs, it splits a GN into an abstract graphical structure, whose elements will be given semantics via NL, and a semantics that comes from the original graphical modelling language used. For the example in Figure 3, the semantics includes the information that the graph represents a finite state machine: nodes represent states; edges represent transitions. The graph of a GN thus represents the topology. Mapping a GN to abstract graphs provides the benefit that we can have a common approach and toolset for a range of GNs.

Conditional statements form an important part of many modelling notations. For example, statecharts has state invariants and trigger conditions on edges. In the presented approach, predicate logic may be used to specify conditional structures in the NL part of the model-integrated specification; we will see that this facilitates semi-automated test generation. It is possible to structure this NL to form logical trees, where leaves are atomic logical statements in NL and the other nodes represent the logical operators OR, AND and XOR.

A. An abstract model

A labelled directed graph G is defined by a tuple (L, V, V_0, E, f_V) in which L is the set of labels; V is the finite set of nodes; $V_0 \subset V$ is the set of initial nodes; $E \subseteq V \times L \times V$ is the finite set of edges (edge (v, l, v') has source node v, destination node v' and label l); and $f_V : V \to L$ is the node label function that maps each node to a corresponding label.

The nodes and edges of graph G provide the structure of the model. The labels allow us to add semantics to these.

L can be any suitably large set since the individual labels can be separately mapped to the required information (such as invariants). In practice, we have found it simplest to use strings, with a label denoting, for example, the name of a state or a function. Information regarding the semantics associated with a label is supplied in a separate table. Tables I and II give such information for the belt-warner system.

TABLE I
LDG OF BELT-WARNER BLOCK DIAGRAM

$$
\begin{aligned}
G_{bd} = (\\
V = \{\ & (n_0, Belt\,Status, source),\\
& (n_1, Ignition\,Status, source),\\
& (n_2, Vehicle\,Speed, source),\\
& (n_3, System\,Time, source),\\
& (n_4, Simple\,Belt\,Warner, node),\\
& (n_5, Start\,Warning, sink),\\
& (n_6, Stop\,Warning, sink),\\
& (n_7, Belt\,Warner\,Activity, sink),\\
& (n_8, Belt\,Warner\,Stand-By, sink)\},\\
E = \{\ & ((n_0, n_4), comfort),\\
& ((n_1, n_4), engine),\\
& ((n_2, n_4), chassis),\\
& ((n_3, n_4), combi),\\
& ((n_4, n_5), sbwout1),\\
& ((n_4, n_6), sbwout2),\\
& ((n_4, n_7), sbwout3),\\
& ((n4, n_8), sbwout4)\})
\end{aligned}
$$

TABLE II
LDG OF BELT-WARNER STATECHART

$$
\begin{aligned}
G_{sc} = (\\
V = \{\ & (n_0, IGNITION\_OFF, node),\\
& (n_1, IGNITION\_ON, hierarchical\,node),\\
& (n_2, STAND\_BY, node),\\
& (n_3, ACTIVE, hierarchical\,node),\\
& (n_4, WARNING\_OFF, node),\\
& (n_5, WARNING\_ON, node)\},\\
E = \{\ & ((-, n_0), Initial\,SBW),\\
& ((n_0, n_1), turning\_on),\\
& ((n_1, n_0), turning\_off),\\
& ((-, n_2), Initial\,IGNITION\_ON),\\
& ((n_2, n_3), activating),\\
& ((-, n_4), Initial\,ACTIVE),\\
& ((n_4, n_5), start\_warning),\\
& ((n_5, n_4), stop\_warning)\})
\end{aligned}
$$

B. Semantics

The process of mapping a graphical model into LDGs can lead to some loss of information that corresponds to elements of the semantics of the GN used. For example, if we map a statechart into an LDG then the resultant graph does not contain information such as the fact that a node represents a state and an edge represents a transition from one state to another. This additional semantic information must be retained; in the proposed approach the identity of the original GN is retained so that it is possible to apply analysis techniques that depend on this semantics. Furthermore, it is possible to attach specific properties $p_0, ..., p_n$ for each model element type. For instance, properties may be used to define a specific chapter structure for each model element type, e.g. a state chapter must contain an invariant definition and a transition chapter must contain a trigger definition.

Tuples are used to define the semantics associated with elements of the LDG; a tuple relates a graph element to a GN element type and additional information, e.g. a template for the chapter title. Chapter title templates are string expression, possibly with placeholders, e.g. for a system name or a model element name. An example of a modeling notation semantics for block diagrams is presented in Table III. The graph of the modeling notation consists of the modeling elements hierarchical node, node, edge, source, and sink. Each modeling element is mapped to the appropriate element of block diagrams, e.g. a node is a block, an edge is a signal flow.

TABLE III
SEMANTICS OF BLOCK DIAGRAM

$$
\begin{aligned}
S_{bd} = \{ \quad & (hierarchical\ node, "Requirements\ [sys\ name]"), \\
& (node, "Requirements\ [sys\ name]". \\
& \quad "Functional\ Requirements\ [sys\ name]"), \\
& (edge, "Requirements\ [sys\ name]". \\
& \quad "Signal\ Flows"."Signal\ Flow\ [name]"), \\
& (source, "Requirements\ [sys\ name]". \\
& \quad "Interface"."Inputs"."Input\ [name]"), \\
& (sink, "Requirements\ [sys\ name]". \\
& \quad "Interface"."Outputs"."Output\ [name]")\}
\end{aligned}
$$

An example of a modeling notation semantics for state charts is presented in Table IV. The graph of the modeling notation consists of the modeling elements hierarchical node, node, and edge. Compared to block diagrams there is no need to consider sinks and sources. Each of these modeling elements is mapped to the appropriate element of statecharts, e.g. a node is a state, an edge is an transition.

TABLE IV
SEMANTICS OF STATECHART

$$
\begin{aligned}
S_{sc} = \{ \quad & (node, "Requirements\ [sys\ name]". \\
& \quad "Functional\ Requirements". \\
& \quad "States"."State\ [name]"), \\
& (hierarchical\ node, "Requirements\ [sys\ name]". \\
& \quad "Functional\ Requirements". \\
& \quad "States"."Hierarchical\ State\ [name]"), \\
& (edge, "Requirements\ [sys\ name]". \\
& \quad "Functional\ Requirements". \\
& \quad "Transitions"."Transition\ [name]")\}
\end{aligned}
$$

C. Generating Specification Structures

A specification structure consists of hierarchically structured chapters, where each chapter provides semantics for an element of the GN. The title of a chapter is created from information regarding a graph element and its semantics properties. Identifiers label graph elements; these provide a one-to-one mapping between chapters and graph elements. For example, it is possible to create a specification structure for the belt-warner through two steps: generate chapters from the Block Diagram (bold chapters); generate chapters from the Statechart. Figure 4 gives the resulting structure.

Chapter X Requirements Belt Warner
Chapter X.1 Interface
Chapter X.1.1 Inputs
Chapter X.1.1.1 Input Belt Status
Chapter X.1.1.2 Input Ignition Status
Chapter X.1.1.3 Input Vehicle Speed
Chapter X.1.1.4 Input System State
Chapter X.1.2 Outputs
Chapter X.1.2.1 Output Start Warning
Chapter X.1.2.2 Output Stop Warning
Chapter X.1.2.3 Output Activity
Chapter X.1.2.4 Output Stand-By
Chapter X.2 Functional Requirements
Chapter X.2.1 States
Chapter X.2.1.1 Hierarchical State IGNITION_ON
Chapter X.2.1.2 Hierarchical State ACTIVE
Chapter X.2.1.3 State IGNITION_OFF
Chapter X.2.1.4 State STAND_BY
Chapter X.2.1.5 State NOT_WARNING
Chapter X.2.1.6 State WARNING
Chapter X.2.2 Transitions
Chapter X.2.2.1 Transition Initial_SBW
Chapter X.2.2.2 Transition Initial_IGNITION_ON
Chapter X.2.2.3 Transition Initial_ACTIVE
Chapter X.2.2.4 Transition turning_on
Chapter X.2.2.5 Transition turning_off
Chapter X.2.2.6 Transition activating
Chapter X.2.2.7 Transition deactivating
Chapter X.2.2.8 Transition start_warning
Chapter X.2.2.9 Transition stop_warning

Fig. 4. Model-integrated structure of belt-warner specification

D. Specification Chapters

A specification chapter has a title and a set of specification entries. Each specification entry can have a specific type, e.g. requirement, information, data type, physical constraint.

Each GN element type can have a specific specification chapter structure, e.g. a state chapter may contain an invariant while a transition chapter may contain initial state, final state, and trigger. As a result, for each modelling notation we use specification chapter templates that define a hierarchical structure of specification entries for each model element type. Common attributes for specification entries are types, owner, date of creation and last edit, or a history record. For model-integration a specification type attribute is defined (Table V).

Logical expressions may be used to define conditions that complement GN, e.g. triggers and invariants. These conditions

TABLE V
SPECIFICATION OBJECT TYPES

Type	Description
heading	a structure element
information	explanations and comments
requirement	the specification of a single required product property

may be specified using NL. Naturally, complex conditions can be split into atomic expressions, these being structured into a tree. An example from the belt-warner is given in Table VI.

TABLE VI
CONDITION SPECIFICATION TYPES

Type	description
condition	atomic condition
XOR	exclusive disjunctive composition of conditions
OR	disjunctive composition of conditions
AND	conjunctive composition of conditions

Since the atomic conditional expressions remain in NL, the graph traversal produces concatenated NL expressions, which only a human reader may check for validity. However, the ability to automatically generate traces from conditional GN was found to be very beneficial.

Returning to the running example, Table VII presents a specification chapter template for states ('tbd' denotes information to be added by the user). Likewise, a chapter template for transitions is presented in Table VIII.

TABLE VII
SPECIFICATION CHAPTER TEMPLATE BELT-WARNER STATE

Type	Short	Text
heading	-	Chapter X.2.1.3 State IGNITION_OFF
information	Context	tbd
AND	Invariant	tbd

TABLE VIII
SPECIFICATION CHAPTER TEMPLATE BELT-WARNER TRANSITION

Type	Short	Text
heading	-	Chapter X.2.2.4 Transition turning_on
information	Context	tbd
requirement	Initial State	tbd
requirement	Final State	tbd
AND	Trigger	tbd

Finally, the specification chapters are complemented manually with details in NL. As an example for the use of specification types, Table IX presents requirements for the activation hysteresis of the simple belt-warner.

Logical conditions like the activation hysteresis are often found in specification documents. The model-integrated specification can be extended with formalised conditional specifications resulting in an extended model-integrated specification; graph operations are still applicable. Again, the formalisation applies only to the structure of the conditional specifications, the conditional statements remain NL.

TABLE IX
BELT-WARNER HYSTERESIS SPECIFICATION

Type	Text
information	The driver must not be distracted through unnecessary belt-warnings at low speeds.
requirement	The belt-warner shall be activated, if vehicle speed exceeds 15 mph.
requirement	The belt-warner shall be deactivated, if vehicle speed drops below 12 mph.
requirement	The belt-warner shall be deactivated, if a relevant malfunction occurs.

TABLE X
BELT-WARNER IGNITION OFF INVARIANT SPECIFICATION

Type	Text $(\bullet - level1, \circ - level2, \star - level3)$
AND	System resides in $IGNITION\_OFF$, if
condition	• Belt-warner ECU not powered.

TABLE XI
BELT-WARNER WARNING OFF INVARIANT SPECIFICATION

Type	Text $(\bullet - level1, \circ - level2, \star - level3)$
AND	System resides in $WARNING\_OFF$, if
condition	• No warning sound emitted.
condition	• Seat-belts fastened.

Examples of specifications of state invariants of the belt-warner are presented in Tables X and XI; others are similar.

Examples of specifications of the triggering conditions of the belt-warner example are presented in Tables XII and XIII.

TABLE XII
BELT-WARNER INITIAL CONDITION SPECIFICATION

Type	Text $(\bullet - level1, \circ - level2, \star - level3)$
AND	Seat-belt warner turns on, if
condition	• System precondition is TRUE.

TABLE XIII
BELT-WARNER START WARNING CONDITION SPECIFICATION

Type	Text $(\bullet - level1, \circ - level2, \star - level3)$
XOR	Seat-belt warning must start, if
condition	• The driver seat-belt is unfastened.
AND	• passenger seat
condition	○ Passenger seat is occupied.
condition	⋆ Passenger seat-belt is unfastened.

The conditional specifications complement the belt-warner model in Figure 2, resulting in a non-deterministic model-integrated specification. This model-integrated specification can be input to further automate analysis and testing steps.

IV. MODEL-BASED TEST-CASE SPECIFICATION

In this section, we present an approach to apply model-based test-case generation to model-integrated specifications.

A. Creating Test-Cases

The presented test-case creation approach combines automated test-case generation with experienced based testing. We

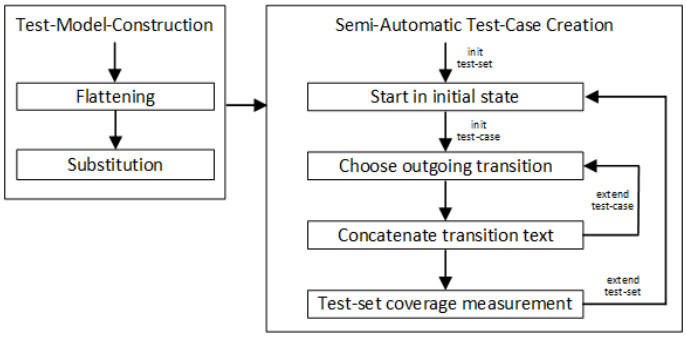

Fig. 5. Model-based testing process

outline this for statecharts. Initially, the statechart is flattened and some modeling elements are substituted. Conditional trigger specifications are transformed into disjunctive normal-form; for each conjunct a corresponding transition is added to the model. The resulting model is a graph to which one can apply common traversal and coverage analysis algorithms.

We have found that it is not always sensible to directly apply graph-traversal algorithms since the GN might not capture all of the relevant information. For example, a specification might contain a cycle that must be repeated a given number of times before it can be left; we could encode this information in the graph by unfolding the cycle but this might lead to a significant increase in the size of the graph. In addition, the GN might not capture important domain knowledge and may contain non-determinism that results from abstraction. Nevertheless, automation remains a key technology to efficiently handle large specifications and reduce the potential for error in carrying out routine tasks. As a result, we have developed an interactive test case generation process that allows the user's domain knowledge to be utilised. After each step, the tool reports the coverage achieved to allow the user to reason about test-case quality. Based on this, the user either decides to terminate or chooses the next transition to take.

The testing process presented in Figure 5 is divided in two phases: automated test-model construction, i.e. flattening and substitution, and semi-automatic test-case generation. A software tool uses flattening and substitution operations to generate a testing model from the model-integrated specification.

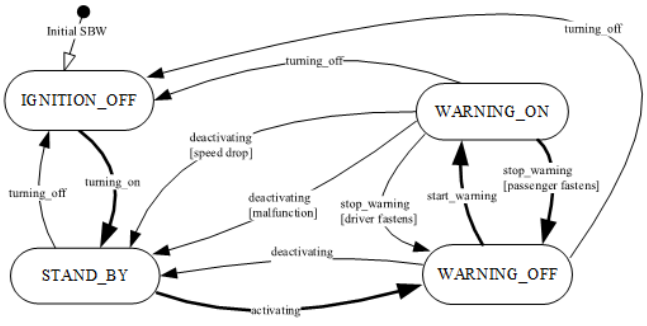

Fig. 6. Test-Model Belt-Warner

The resulting testing model for the belt-warner example is presented in Figure 6. The testing model is used by another software tool to support the test-case specification process by advising the test engineer. Test-case creation starts with an empty sequence in the initial state (the initial transition may add text to the test-case too, but for now we will ignore this). The software tool supports the test-engineer by offering the current state's outgoing transitions and its textual contents, which may be chosen to extend the test-case. When the test-engineer selects a transition its textual content is appended to the test-case and the current state is changed.

Once a test-case is finished, it is added to a test-set (a collection of test-cases). The tool applies coverage metrics to a test-set so that the test engineer can decide whether further test-cases should be added. Table XIV gives a test-case created using the tool from the belt-warner model. The test-case covers the edges marked bold in the model presented in Figure 6.

TABLE XIV
TEST-CASE EXAMPLE

No.	Input	Output
0	*[Initial SBW]* System precondition is TRUE.	*[Invariant IGNITION_OFF]* Belt-warner ECU not powered.
1	*[Trigger turning_on]* Ignition key turned ON. Error memory is empty. *[Trigger Initial IGNITION_ON]* Set power status bit $= 1$.	*[Invariant IGNITION_ON]* Belt-warner ECU powered. *[Invariant STAND_BY]* Belt-warner signals stand-by.
2	*[Trigger activating]* Vehicle speed ≥ 25 km/h. *[Trigger Initial ACTIVE]* Set activity bit $= 1$.	*[Invariant IGNITION_ON]* Belt-warner ECU powered. Error memory is void. *[Invariant ACTIVE]* Belt warner signals activity. Vehicle speed > 25 km/h. *[Invariant WARNING_OFF]* No warning sound emitted. Seat belts fastened.
3	*[Trigger start_warning]* Passenger seat occupied. Passenger seat-belt unfastened.	*[Invariant IGNITION_ON]* Belt-warner ECU powered. Error memory is void. *[Invariant ACTIVE]* Belt warner signals activity. Vehicle speed > 25 km/h. *[Invariant WARNING_ON]* A warning sound emitted. Passenger seat-belt unfastened.
4	*[Trigger stop_warning]* Warning emitted. Driver seat-belt fastened. Passenger seat occupied. Passenger seat-belt fastened.	*[Invariant IGNITION_ON]* Belt-warner ECU powered. Error memory is void. *[Invariant ACTIVE]* Belt warner signals activity. Vehicle speed > 25 km/h. *[Invariant WARNING_OFF]* No warning sound emitted. Seat-belts fastened.

B. Test coverage criteria

Let us suppose that S is a model-integrated specification, G_S is an LDG generated from S, and G_T is a test-model derived from S. Further, suppose that t_S is a test suite. We have several possible coverage metrics.

Edge coverage C_G is the fraction of edges of G_S that are covered when executing t_S. Condition coverage C_S is the fraction of conditional statements (within structured logical

expressions) of S covered when executing t_S. Trigger coverage C_T, is the fraction of edges of the test-model T that are covered when executing t_S. For effort estimation, an upper bound on the number of edges in T is provided by taking the sum, over the edges of G_S, of the number of conditions in the DNF form of the triggers. The test-case presented in Table XIV results in the coverage rates presented in Table XV.

TABLE XV
BELT-WARNER TEST COVERAGE EXAMPLE

Transition	C_G	C_S	C_T
Initial ACTIVE	100%	100%	100%
Initial IGNITION_ON	100%	100%	100%
Initial SBW	100%	100%	100%
activating	100%	100%	100%
deactivating	0%	0%	0%
start_warning	100%	50%	50%
stop_warning	100%	50%	100%
turning_off	0%	0%	0%
turning_on	100%	100%	100%

V. CASE STUDIES

The model-integration approach and tool have been used in a number of development projects in industry: system specification of various parking assistance systems using block diagrams and statecharts; functional specification of emergency braking systems using statecharts; system specification of camera-based surround view systems; system, function and test-case specifications of the control of an automated manual gearbox; architecture, function and user interface specifications of a database management tool; and architecture and function of a software-controlled medical device.

Among the specified driver assistance systems are 4 parking systems, 2 brake-assistance systems, and a camera-based traffic information system, for which it was found that a major benefit lies in the clear structure and transparency of the model-integrated specification. The largest specification included 5 functional variants of a parking system containing more than 2000 requirements, which were structured using 8 statechart models, of which the largest contained 40 states and 69 transitions. Frequently it was seen that the specifications would increase in size. Even though redundancies can be eliminated completely, the model-based approach enforces completeness and the additional, missing, requirements caused the specification to grow.

Functional specifications of an automated gear-box were created using the model-integration tool; experience data is presented in Table XVI. Two requirements engineers took four months to create model-integrated specifications from high-level functional specification. Since the functional specifications were found to be incomplete and inconsistent, workshops were held with the system developers to elicit missing requirements and to remove specification faults. The model-integrated specifications were complemented with testing information, so that the same specifications could be used to generate test-cases for manual and automatic execution.

TABLE XVI
AUTOMATIC GEARBOX APPLICATION STATISTICS

Component	**Requirements**	**Nodes**	**Edges**
Coasting	315	18	36
Creeping	275	15	34
Garage Shift	268	12	34
Launch	262	16	32
Shift Quality	293	14	21
Stalling	144	11	20
Startup&Shutdown	117	17	26

The model-integration has also been used to specify a medical system consisting of 7 major components. Experience data is presented in Table XVII.

TABLE XVII
MEDICAL DEVICE APPLICATION STATISTICS

Component	**Requirements**	**Nodes**	**Edges**
ECU	91	6	8
Sensors	98	8	15
Actuators	92	8	14
Radio Arch.	17	2	2
Radio Func.	83	5	15
Radio Master	92	11	15
Radio Client	71	8	13

The model-integrated specification approach is in use. To specify graphical user interfaces, a specific graphical notation and semantics have been developed that represents the graphical widgets and their placement on the screen. Functional behaviour, in the form of a statechart, can be attached to widgets. The project, including the specification approach, has been assessed and certified for SPICE level 1 [5].

TABLE XVIII
COMPARISON OF APPLICATION STATISTICS IN DEVELOPMENT PROJECTS

System	Comp.	Models	Nodes	Edges	Effort[h]
Parking 1	4	8	40	69	500
Parking 2	4	5	19	38	650
Brake 1	1	2	6	8	700
Brake 2	1	2	15	18	800
Gearbox	7	7	103	203	1100
SW Tool	3	26	133	235	900
Medical Device	4	7	48	82	400

Table XVIII presents statistics regarding specifications created using the approach presented. The rows refer to the specified systems. The columns contain counts of components, models, states and transitions included in the specifications.

Table XVIII shows very different efficiencies (efforts per node, edge, model, or component), where effort is a rough approximation of the number of person hours required to create and maintain the specification during the project. The parking and brake assist system projects were the first applications of the approach and were involved in the development of the method and tool. Thus the comparably low efficiency is caused by less automation and a less mature approach. Furthermore, we found that the number of components and models are individual decisions made during the early phases of each

project. The influence of such decisions on the efficiency of development activities is hard to determine. However, as a rule of thumb, we found during the further projects that specifying and maintaining a chapter takes approximately up to two days.

VI. Related Work

The basic approach described in this paper has previously been presented [6]. This paper provides additional details and formalisation as well as an extended example and experience data. An additional extension is the inclusion of a semi-automatic model-based testing approach and associated coverage criteria. The model-integration approach contributes to the RE reference model presented in [7] as it provides a method to systematically integrate and relate artefacts (as domain knowledge) into artefacts of requirements and specifications.

The presented approach contributes to the concept of viewpoints resolution presented in [8] through abstraction and unification of document structures that significantly eases the comparison and gap analysis of different viewpoints. The extension of the presented approach to methods for the identification of missing and wrong facts over a set of comprehensive model-integrated specifications is a topic of ongoing work and inspired by the viewpoints analysis presented in [8].

In [6] we noted that structure has a significant impact on the quality of requirements sets. Previous work has used clustering to automatically restructure comprehensive requirements sets [9]. In contrast, model-integration takes advantage of the knowledge and experience of the human reader, who is supported by an improved document structure. Similar approaches have been presented on the integration of textual and graphical modeling languages, e.g. [10], [11], although these do not consider natural language as a textual language.

There are many approaches to model-based testing. However, they typically require the existence of a model in a formal language such as a finite state machine [12] or labelled transition system [13], possibly enriched by aspects such as time [14] or probabilities [15], [16]. While NL explanations can complement a model in such a language, the NL is not normally integrated with the model.

VII. Conclusions

The model-integrated approach has been presented, with this making it possible to seemlessly integrate textual specifications and models, resulting in a semi-formal specification. The level of formalisation of the resulting specifications makes it possible to use formal analyses and graph operations, e.g. test-case generation. The benefits include the following:

1) formal definition of requirements completeness;
2) efficiency through automation;
3) fewer manual faults through automation;
4) improved clarity through use of the model as a document structure;
5) fewer redundancies and inconsistencies through formal soundness criteria and checking capabilities;
6) no divergence of the GN and NL parts of a specification;
7) eased and accelerated maintenance and change.

The approach has been implemented in a tool. It has been applied in several projects in industry over a period of 6 years.

Future work will aim at the reimplementation of the tool independent of specific commercial requirements management systems and that utilises the model-integration approach. Furthermore, an extended approach is under development that allows for bidirectional changing of models, i.e. to transfer changes of the model from NL into GN and vice versa. Currently, it is only possible to change the model through manually changing the GN-part and transferring these changes automatically to the NL-part. There may also be scope to automatically derive some structure from NL.

References

[1] T. Gilb, "No cure no pay: How to contract for software services," *Comput. Sci. Inf. Syst.*, vol. 4, no. 1, pp. 29–41, 2007.

[2] J. W. Nilsson, *Electric circuits (4. ed.)*, ser. Addison-Wesley series in electrical and computer engineering. Addison-Wesley, 1993.

[3] M. K. Stojcev, "John p. hayes, computer architecture and organization, third ed., mcgraw-hill book company, inc., boston, 1988, softcover, pp 604, plus xiv, ISBN 0-07-115997-5," *Microelectronics Reliability*, vol. 46, no. 1, pp. 196–197, 2006.

[4] D. Harel, "Statecharts in the making: a personal account," *Commun. ACM*, vol. 52, no. 3, pp. 67–75, 2009.

[5] I. O. for Standardization, "Software process improvement and capability determination (spice)," Tech. Rep., 2012.

[6] C. Robinson-Mallett, "An approach on integrating models and textual specifications," in *Second IEEE International Workshop on Model-Driven Requirements Engineering, MoDRE 2012, Chicago, IL, USA, September 24, 2012*, 2012, pp. 92–96.

[7] C. A. Gunter, E. L. Gunter, M. Jackson, and P. Zave, "A reference model for requirements and specifications," *IEEE Software*, vol. 17, no. 3, pp. 37–43, 2000. [Online]. Available: http://dx.doi.org/10.1109/52.896248

[8] J. C. S. do Prado Leite and P. Freeman, "Requirements validation through viewpoint resolution," *IEEE Trans. Software Eng.*, vol. 17, no. 12, pp. 1253–1269, 1991.

[9] A. Ferrari, S. Gnesi, and G. Tolomei, "Using clustering to improve the structure of natural language requirements documents," in *Requirements Engineering: Foundation for Software Quality - 19th International Working Conference, REFSQ 2013, Essen, Germany, April 8-11, 2013. Proceedings*, 2013, pp. 34–49.

[10] L. Engelen and M. van den Brand, "Integrating textual and graphical modelling languages," *Electr. Notes Theor. Comput. Sci.*, vol. 253, no. 7, pp. 105–120, 2010.

[11] M. Scheidgen, "Textual modelling embedded into graphical modelling," in *Model Driven Architecture - Foundations and Applications, 4th European Conference, ECMDA-FA 2008, Berlin, Germany, June 9-13, 2008. Proceedings*, 2008, pp. 153–168.

[12] R. M. Hierons and U. C. Türker, "Parallel algorithms for testing finite state machines: Generating UIO sequences," *IEEE Transactions on Software Engineering*, vol. 42, no. 11, pp. 1077–1091, 2016.

[13] J. Tretmans, "Model based testing with labelled transition systems," in *Formal Methods and Testing*, ser. Lecture Notes in Computer Science, vol. 4949. Springer, 2008, pp. 1–38.

[14] C. Gaston, R. M. Hierons, and P. L. Gall, "An implementation relation and test framework for timed distributed systems," in *25th IFIP WG 6.1 International Conference on Testing Software and Systems (ICTSS 2013)*, ser. Lecture Notes in Computer Science, vol. 8254. Springer, 2013, pp. 82–97.

[15] M. Gerhold and M. Stoelinga, "Model-based testing of probabilistic systems," in *Fundamental Approaches to Software Engineering - 19th International Conference, FASE 2016*, ser. Lecture Notes in Computer Science, vol. 9633. Springer, 2016, pp. 251–268.

[16] G. H. Walton, J. H. Poore, and C. J. Trammell, "Statistical testing of software based on a usage model," *Softw., Pract. Exper.*, vol. 25, no. 1, pp. 97–108, 1995.