Integrating Pattern-based Formal Requirements Specification in an Industrial Tool-chain

Predrag Filipovikj*, Trevor Jagerfield*, Mattias Nyberg†, Guillermo Rodriguez-Navas*, Cristina Seceleanu*
*Mälardalen University, Västerås, Sweden
{predrag.filipovikj, guillermo.rodriguez-navas, cristina.seceleanu}@mdh.se, trevor.jagerfield@gmail.com
†Scania, Södertälje, Sweden, mattias.nyberg@scania.com

***Abstract*—The lack of formal system specifications is a major obstacle to the widespread adoption of formal verification techniques in industrial settings. Specification patterns represent a promising approach that can fill this gap by enabling non-expert practitioners to write formal specifications based on reusing solutions to commonly occurring problems. Despite the fact that the specification patterns have been proven suitable for specification of industrial systems, there is no engineer-friendly tool support adequate for industrial adoption. In this paper, we present a tool called SESAMM Specifier in which we integrate a subset of the specification patterns for formal requirements specification, called SPS, into an existing industrial tool-chain. The tool provides the necessary means for the formal specification of system requirements and the later validation of the formally expressed behavior.**

I. Introduction

The potential benefits of using formal techniques for specification and verification of systems, especially in what concerns product quality and reduction of verification effort, have been reported in several studies [1]. Formal specification/verification is strongly recommended by some safety standards, particularly for high-integrity safety levels [2].

Nowadays, the lack of formal requirement specifications constitutes a major obstacle to the widespread adoption of formal verification. In most industrial settings, engineers use only informal documents, written in natural language or including graphs with no well-defined semantics, until very late in the development process. The use of formal notation is typically not well understood by engineers, hence rarely employed by current practices.

Pattern-based approaches have been proposed in order to facilitate the task of formalizing requirement specifications and make it amenable to the average engineer. The approach is based on the fact that requirements specifications are written in reoccurring solutions (or *patterns*) that can be expressed in structured English language with automatic translation into corresponding formal counterparts. The patterns have been standardized and organized into catalog for enabling the knowledge transfer among the practitioners, thus enabling future reuse. Some first experiences show that this approach has potential for industrial application, but choosing the appropriate pattern for each requirement is still challenging and error-prone due to the lack of familiarity with formal logics [3] [4].

Tools originating from academia have focused on the formal aspects of the pattern-based methods and have paid little attention to issues that are vital to industry, such as: ease of use, extensiveness of the pattern catalog and possibility for integration within existing industrial tool-chains. In this paper we try to fill this gap, by showing how tool support for pattern-based formalization of requirements can be integrated with an actual tool-chain for designing vehicular control systems. The resulting tool is called SESAMM Specifier, which we have designed and developed according to the engineers' needs from the heavy-truck industry.

The rest of the paper is organized as follows. In Section II, we present an overview of the specification patterns and visual notations for formalized requirements, which we implement in this work. In Section III, we describe the existing industrial tool-chain that we integrate the SESAMM Specifier with. The description of SESAMM Specifier and the implementation details are given in Section IV. Section V compares to related work, whereas in Section VI we discuss the approach, before concluding the paper in Section VII.

II. Background

In this section we present the preliminary concepts used in the paper. Section II-A overviews the pattern-based approach for formal system specification, whereas Section II-B overviews the graphical notations used for visualizing the behavior of the pattern-based system specification.

A. Specification Patterns

The specification pattern system (SPS) [5] is a catalog of specification patterns used for specifying concurrent and reactive systems. A specification pattern represents a generalized description of a commonly occurring solution when specifying requirements. It captures some aspect of the system's behavior and provides mechanisms to express this behavior in different formalisms. A pattern is characterized by *name*, *intent*, and *scope*, where the intent represents the structure of the described behavior and scope denotes the extent of program execution to which the pattern must hold. The intent of a pattern can be expressed in various specification formalisms, which can be event-based (ex: quantified regular expressions (QRE) [6]) or state-based (ex: linear temporal logic (LTL) [7], computation tree logic (CTL) [8]). The five basic pattern scopes [5] are: *globally* (the entire program execution), *before* (before the first occurrence of a given state/event), *after* (after the occurrence of given state/event), *between* (any part of program execution between two states/events), and *after-until* (similar to

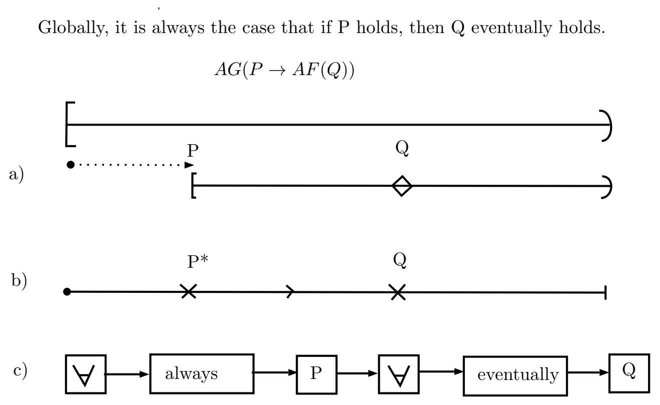

Globally, it is always the case that if P holds, then Q eventually holds.
$AG(P \to AF(Q))$
a)
P
Q
b)
P*
Q
c)
∀
always
P
∀
eventually
Q

Fig. 1. Response pattern encoded in graphical notations. a) RTGIL b) UCMPPS c) L_R

between except that the occurrence of the second event is not mandatory).

To facilitate easier browsing, the patterns in the SPS catalog are organized into hierarchies according to their semantics, such that the similar patterns are connected to each other. On the top level, the catalog is composed of two main categories: *Occurrence* and *Order*. The *Occurrence* category contains the Absence, Universality, Existence and Bound Existence patterns that capture the occurrence/absence of a given state/event. The patterns in *Order* category including: *Precedence*, *Response*, *Chain Precedence* and *Chain Response* describe the relative order in which multiple events/states occur.

Subsequently, Konrad and Cheng [9] extended the original SPS catalog by including a new class of specification patterns for real-time system properties. The formal encoding of real-time patterns is given in metric temporal logic (MTL) [10], timed CTL (TCTL) [11] and real-time graphical interval logic (RTGIL) [12]. To facilitate the adoption of specification patterns by non-expert practitioners, Konrad and Cheng proposed a pattern representation in structured English. Such representation enables practitioners to understand the meaning of a property without knowledge of its temporal logic representation.

B. Methods for Visual Representation of Semantics of SPS patterns

This section presents an overview of three different approaches that provide support for the graphical representation of behavior captured in formal notation, which we implement in our tool. Each of the notations will be discussed via a simple scenario of a Response pattern with global scope, corresponding to the CTL formula $AG(P \to AF\,Q)$ [5], which reads: "Globally, it is always the case that if P holds, then Q eventually holds".

a) Real-time GIL (RTGIL) [13] is an extension of graphics interval logic (GIL) that provides support for specification of real-time properties. Fig. 1a shows the RTGIL representation of the Response pattern. Formulas in RTGIL are read from top to bottom and from left to right, starting from the topmost interval. The progression of the computation in RTGIL is represented as a time-line (top line in Fig. 1a). An interval represents a segment of the time-line closed to the left and open to right. Each interval in RTGIL starts and ends with a search pattern, which is a sequence of states that starts from the beginning of the context or at the state located by the previous search pattern. The example contains two search patterns, one for each of the boolean terms involved (P, Q). The first search interval for P starts at the beginning of the context and it is represented by a dotted line, denoting that the search interval can be empty, that is, P can also be missing in the context. The search interval terminates when the designated term evaluates to true. The second search interval starts at the end of the previous interval and is valid until the end of the time-line. The interval has been decorated with a diamond symbol, meaning that Q will eventually become true in the interval. The representation of formalized behavior using intervals can be useful to capture the temporal ordering of events, which is not easy to comprehend by reading the logical formula only.

b) Hassine [14] proposes a system called *Use Case Map Property Pattern System* (UCMPPS) that provides graphical representation for all the specification patterns of the quantitative [5] and real-time [9] categories based on the Use Case Map (UCM) notation. UCM is a high-level design language that enables reasoning about the systems' behavioral patterns. Fig. 1b depicts the UCMPPS representation of the Response pattern with global scope. The complete system execution is represented with a horizontal line called scenario path, in which the time progresses from left to right. The beginning of the scenario is denoted with a filled circle, and the end with a vertical bar. The scenario contains *responsibilities* that represent abstract activities that can be refined in terms of functions, events, etc. The pattern in Fig. 1b has two responsibilities denoted by "x", which correspond to events P and Q. The star in the name of event P denotes that its occurrence is optional. The arrow between the two events is added to distinguish the general response property from the restricted response (Q follows P immediately, which is denoted without an arrow on the scenario line). Other elements in the UCM notation that are not used in the given example include: components, forks, joins, stubs and timers.

c) L_R [15] is a specification language based on graphical notation intended to facilitate the formal specification of system properties described in temporal logic. The language is intended to be flexible, thus has no fixed list of constructs. The intended use of the language is through a two-level approach, namely expert and user level. On the expert level, a set of patterns is defined, which is later used to specify system properties on the user level represented as directed acyclic graphs. Nodes in the graph represent predicates, logical connectives, modal operators and quantified temporal operators. Each of the nodes has an associated graphical construct that denotes its semantics. Fig. 1c contains the L_R graph corresponding to the Response pattern. The graph is composed of nodes, and the directed arrows show the order of execution. The first node in the graph is the universal path quantifier ("$\forall$") followed by a keyword "*always*" denoting the global temporal operator. Next, there comes the event/state P that for all paths will "*eventually*" (that is, in finite time) be followed by event/state Q.

The approaches a) and b) use time progression to depict the time-line of the system execution. The events are presented on a time-line that clearly captures their temporal ordering. The approach described as c), on the other hand, represents the constructs of a logical formula, that is, propositions, temporal operators and branch quantifiers, as graphical elements that can be easily understood and manipulated by engineers who are used to notations such as UML.

III. Description of the Existing Industrial Tool-chain

In this section we describe the industrial tool-chain, called *SESAMM*, which represents a collection of tools for visualizing architectural models, requirements specification and failure propagation modeling. The SESAMM tool-chain comprises research prototype tools, intended to continuously evolve until reaching production quality.

SESAMM has been originally developed as a tool-chain for the extraction of architectural models from implementation, that is, the deployed source code. The initial implementation, as described in previous work [16], has four main parts: (i) a back-end C-code parser and parser coordinator, (ii) a graph database, and two front-end applications: (iii) *Architecture Browser* and (iv) *CAN Verifier*. The goal is to build tools for architecture recovery intended for mitigating the problem of architectural drift that occurs when the system implementation and the documentation evolve in inconsistent ways.

The basic functionality of the code parser and the respective coordinator is to retrieve the architectural information based on the source code. For extracting the architectural information, the parser parses the source files from the application layer software and CAN communication layer, and stores that information into a graph database. The Architecture Browser front-end tool is used for the interactive visualization of the extracted software architecture as such. The tool provides an improved overall understanding of the system's architecture, based on two views, namely: the network diagram that represents the computational elements and main networks of the heavy-truck's design, and the software/hardware diagram, which shows the architectural break-down of the software and hardware architectures of the computational elements. The second front-end tool, called CAN Verifier, is used for verifying the content of the CAN communication layer, by comparing the parsed data with the data in other external databases. The aim is to find potential inconsistencies between the design and implementation, which as reported by the authors has been fulfilled. In subsequent research endeavors, the SESAMM tool-chain has been extended with a tool for failure propagation modeling [17]. The idea of the approach is to exploit the requirements specification structured using contracts theory for creating models of failure propagation. The tool provides interface features for: i) requirements specification in natural language and basic traceability options based on the assumption-guarantee contracts theory [18], and ii) specification of failure modes and generation of logical relations of faults/failures linked to the assumptions of each requirement specified as a guarantee.

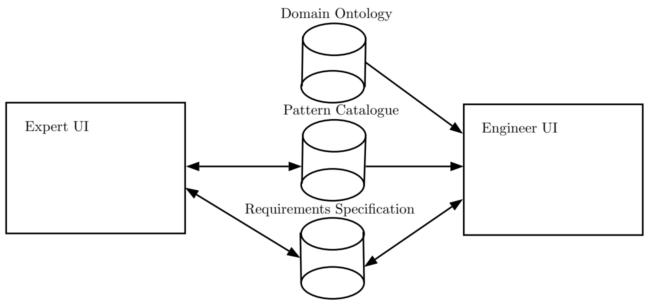


Fig. 2. SESAMM Specifier Architecture

IV. SESAMM Specifier

In this section, we describe our main contribution, the SESAMM Specifier tool, which supports (industrial) practitioners not experts in formal notations, in the process of formal requirements specification. The requirements specification tool has been integrated into the SESAMM tool-chain introduced in Section III.

The SESAMM Specifier has been designed to satisfy demands such as *flexibility* and *extensiveness*. Flexibility is reflected by the fact that the tool can be integrated into existing tool-chains used in industrial settings instead of being developed as standalone tool. On the other hand, extensiveness means that the tool has been developed around the pattern-based approach for requirements specification without being bound to a specific pattern catalog, thus it can be extended and adapted further. This feature facilitates the tool's customization to satisfy the needs of different stakeholders. To obtain a consistent integration with the rest of the tools within the tool-chain, the SESAMM Specifier has been implemented as a desktop application using .NET framework v4.5, with the back-end code written in C# and interface implemented using the Windows Presentation Foundation (WPF). In the remainder of the section, we present the tool by two important aspects: its architecture and functionality. The functionality of the tool is described via a workflow diagram, and user interfaces (UIs), where each step of the workflow is mirrored by the part of the respective UI that realizes it.

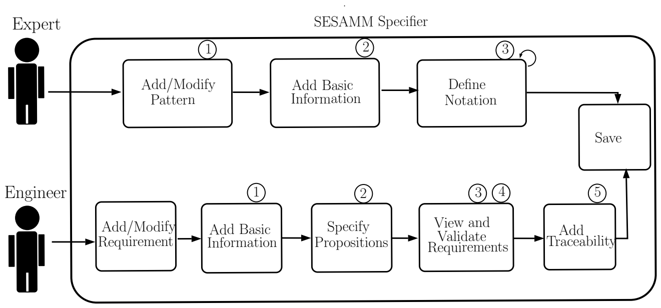


Fig. 3. Expert and Engineer work-flows of SESAMM Specifier

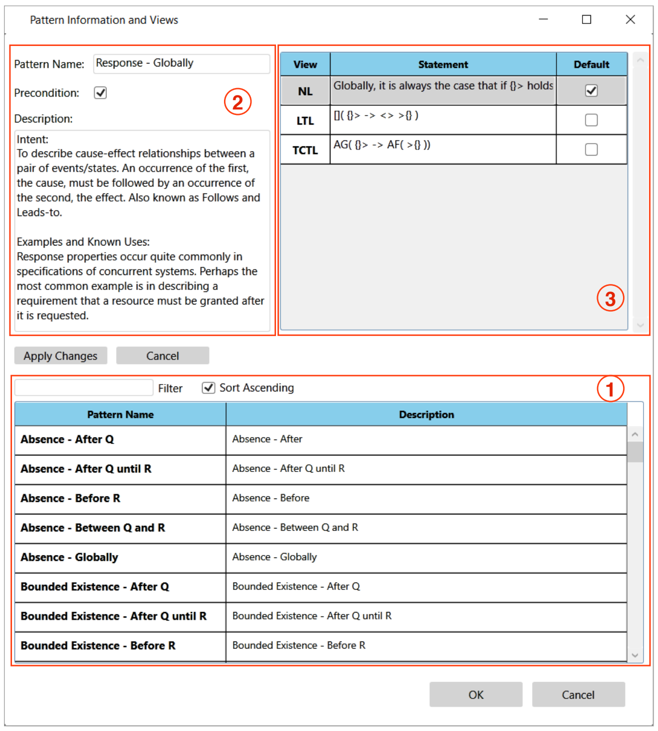
The architecture of the SESAMM Specifier (Fig. 2) consists of three data sources for storing data and two UIs for managing it. The arrows between the architectural elements indicate the data flow within the tool. An arrow from a data source to an UI denotes read-only data access, while a two way arrow indicates that a given UI can both read from, and write data to that particular data source. The *Domain Ontology* stores information about various concepts (function and signal names, software variables, etc.) from the system architectural design, managed by the C-code parser from the SESAMM tool chain (see Section III). This data source is accessible only by the engineers through the Engineer UI. The *Pattern Catalog* data source contains all the patterns that can be used for requirements specification, managed by a formal methods expert through the designated Expert UI. The catalog stores information relevant to each specification pattern, including the pattern's name and different notations used for representing the pattern. The specified requirements using our SESAMM Specifier tool are stored in *Requirements Specification*.

The intended workflows for both experts and engineers are given in Fig. 3. In the following, we present both workflows and the respective UIs by which they are realized.

A. *Expert Workflow and UI*

All the features of the tool intended for the experts revolve around creating, modifying and deleting patterns in the pattern catalog. The Expert UI through which these functionalities can be exercised is given in Fig. 4.

The Expert UI has been divided into three sections, each corresponding to an action in the workflow denoted by the same number. The first step in the expert workflow is the initialization of the process for adding, modifying or removing a pattern from the catalog. This action is realized by "Section 1" of the Expert UI. It contains a list of all available patterns in the catalog. The actions for adding, modifying and removing patterns are given in a context menu that can be opened by a right-click on the list of patterns. Due to space limitation, in the following we describe only the scenario for creating a new pattern. The modification of an existing pattern contains the same steps, whereas removing a pattern is done in a single step.



Pattern Name: Response - Globally

Precondition: ✓

Description:

Intent:
To describe cause-effect relationships between a pair of events/states. An occurrence of the first, the cause, must be followed by an occurrence of the second, the effect. Also known as Follows and Leads-to.

Examples and Known Uses:
Response properties occur quite commonly in specifications of concurrent systems. Perhaps the most common example is in describing a requirement that a resource must be granted after it is requested.

View	Statement	Default
NL	Globally, it is always the case that if {}> holds	✓
LTL	[]({}> -> <> >{})	☐
TCTL	AG({}> -> AF(>{}))	☐

✓ Sort Ascending

Pattern Name	Description
Absence - After Q	Absence - After
Absence - After Q until R	Absence - After Q until R
Absence - Before R	Absence - Before
Absence - Between Q and R	Absence - Between Q and R
Absence - Globally	Absence - Globally
Bounded Existence - After Q	Bounded Existence - After Q
Bounded Existence - After Q until R	Bounded Existence - After Q until R
Bounded Existence - Before R	Bounded Existence - Before R

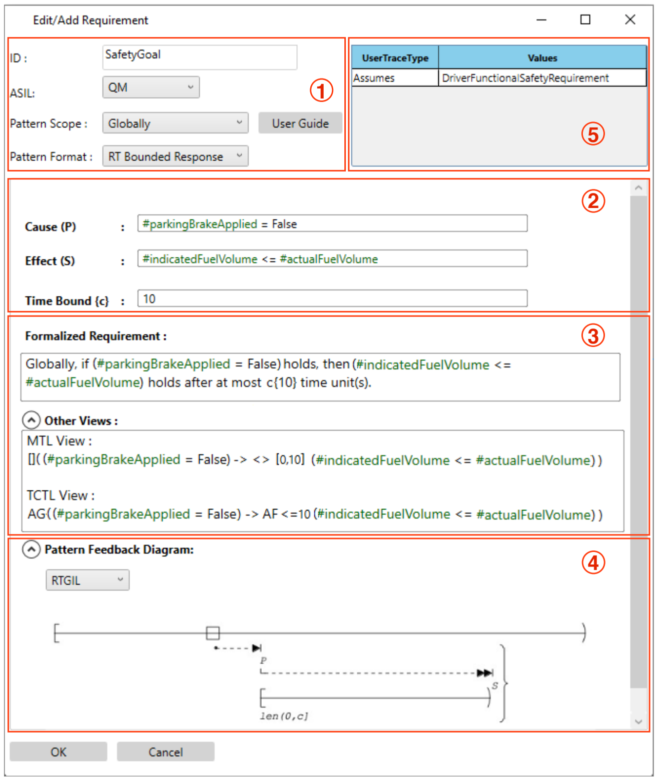
Fig. 4. SESAMM Specifier: Expert User Interface

After selecting the option of adding a new pattern, during the second step of the workflow the expert enters the basic data for the pattern, which includes its name and description. To enter this data into the system, the expert uses "Section 2" of the Expert UI as presented in Fig. 4. The last expert action in the workflow is to define the different notations (also called views) in which the given pattern can be encoded. In "Section 3" of the Expert UI one can see that for the given pattern the expert has defined its representation in natural language (NL), LTL, and CTL. Additionally, the expert has selected the NL to be the default view of the pattern.

B. *Engineer Workflow and UI*

The requirements specification workflow intended for the engineers' use is realized by the Engineer UI given in Fig. 5. Again, each of the actions in the workflow has a designated part in the Engineer UI through which it is realized. To illustrate the features of the engineers' workflow, we chose to formalize one requirement of an operational industrial system, as follows: ***"If the parking brake is not applied, then the fuel volume shown by the gauge shall be less or equal than the fuel volume in the tank, within 10 ms"***. The behavior of the requirement can be captured via the Bounded Response pattern, valid during the entire program execution, that is, bearing a Global scope.

After launching the Engineer UI, during the first step of the workflow the engineer is required to enter the basic information for the requirement, by using the "Section 1" of the UI given in Fig. 5. The basic data includes the



UserTraceType	Values
Assumes	DriverFunctionalSafetyRequirement

ID : SafetyGoal

ASIL: QM

Pattern Scope : Globally

Pattern Format : RT Bounded Response

Cause (P) : #parkingBrakeApplied = False

Effect (S) : #indicatedFuelVolume <= #actualFuelVolume

Time Bound {c} : 10

Formalized Requirement :

Globally, if (#parkingBrakeApplied = False) holds, then (#indicatedFuelVolume <= #actualFuelVolume) holds after at most c{10} time unit(s).

Other Views :

MTL View :
```
[]((#parkingBrakeApplied = False) -> <> [0,10] (#indicatedFuelVolume <= #actualFuelVolume))
```

TCTL View :
```
AG((#parkingBrakeApplied = False) -> AF <=10 (#indicatedFuelVolume <= #actualFuelVolume))
```

Pattern Feedback Diagram:

RTGIL

Fig. 5. SESAMM Specifier: Engineer User Interface

requirement ID, an automotive safety integrity level (ASIL) [2], as well as the intent and scope of the requirement. In cases when the user is not sure which intent and scope capture the semantics of the requirements most accurately, the tool provides a multi-step process that leads the users to determining patterning information through a series of steps. For this purpose we have implemented a custom version of a Question Tree [19] mechanism, which uses a set of predefined questions meant to guide the engineers towards a more accurate selection.

In the second step of the workflow, the engineer specifies the terms describing the states/events over which the behavior of the pattern is defined. There are two classes of terms that should be specified: the first contains the states/events defining the behavior of the pattern, whereas to the second class belong those that define the scope. The data concerning the terms is entered via "Section 2" of the Engineer UI. Since the number of states/events for different combinations of intent and scope is not constant, the interface is automatically adjusted to represent the information relevant to the currently selected intent and scope. For the given example, there are two terms describing the behavior and no term for the scope since the Global option is used. The representation of the requirement in TCTL is given as: $AG(\#parkingBrakeApplied = False \rightarrow AF_{\leq 10}(\#indicatedFuelVolume \leq \#actualFuelVolume))$. The phrases starting with ("#") denote concepts from the domain ontology. The $\#parkingBrakeApplied$ represents a domain concept that models the status of the parking brake of the vehicle using a Boolean variable. Similarly, $\#indicatedFuelVolume$ models the fuel volume displayed to the driver, while the $\#actualFuelVolume$ models the fuel that is in the fuel tank of the vehicle. The input of domain concepts is supported by auto-complete functionality built into the tool, hence preventing typing errors and supporting the engineers in typing complex domain concepts. Encoding concepts from the domain ontology into the requirements specification is important from two points of view. First, it assures the traceability between the requirements and the architectural elements to which given requirements have been assigned; by encoding concepts from the domain ontology, the tool establishes such traceability, automatically. This traceability information is exploited by the SESAMM tool-chain to help the designers trace the requirements down to architectural artifacts, at various design levels, and ultimately to the code itself. Second, the meta information for each concept, such as type (enum, integer, string, etc.), range of values etc., is used by the tool to perform an automatic on-the-fly validation of the specified requirement. For instance, based on the meta information, the run-time validation engine prevents assigning or comparing the concept $\#parkingBrakeApplied$ of type Boolean to values other than "True" or "False".

In the steps 3 and 4 of the workflow, the engineer performs the validation of the specified requirement by browsing through the different available notations. In "Section 3" of the Engineer UI one can notice that the requirement is represented in notations that include: controlled natural language, MTL and TCTL. These representations are dynamically created by the tool based on the information provided in "Sections 1 and 2". If the information in "Sections 1 and 2" is additionally updated, the information in "Section 3" is automatically updated also, to ensure consistency between the user input and the presentation. One can visualize the given requirement in different graphical notations in "Section 4". Currently, the tool supports visualizing requirements in RTGIL, UCMPPS or L_R, and the navigation between the visual notations is provided by using the drop-down list. In the example, we have selected RTGIL to visualize the behavior of the example requirement (see Fig. 5). The diagram reads as follows: within the complete program execution (denoted by a square on the main time-line), the occurrence of event/state P leads to the occurrence of event/state S, within the right-closed interval given as (0, c]. The constant "c" is used to keep the reusability of the RTGIL diagrams, with its actual value specified in "Section 2" as a "Time Bound" property.

The last step in the workflow requires entering the traceability information between requirements, using the traceability matrix of "Section 5" of the Engineer UI. The tool provides two types of traceability links, namely *decomposes* and *assumes*, as proposed by the assume-guarantee contract-based system requirements specification approach [18]. For the example given in Fig. 5, the fulfillment of the newly created "SafetyGoal" requirement depends on the fulfillment of the "DriverFunctionalSafetyRerquirement" by the particular system. Based on this traceability links, the decomposition of the system's requirements specification can be visualized in form of decomposition graphs.

V. Related Work

A number of research endeavors have shown the fact that the adoption of any pattern-based approach depends on an adequate tool support. In the following, we present an overview of five academic tools for formal requirements specification that rely on the pattern-based approach that we have considered.

The "PROPerty ELucidation" system (PROPEL) [19] and Property Specification (Prospect) [20] are among the first tools for formal specification of system properties based on specification patterns. The tools provide interfaces for requirements specification using disciplined natural language and finite-state automata (FSA). Subsequently, PROPEL has been extended to encompass guidance mechanisms for the users during the process of selecting the correct template [21], in form of Behavior Question Tree techniques. Both of the tools use only state-base notations to formally represent requirements behavior. Due to the limitations of the state-based formalisms, a second version of the Prospect tool has been designed to include the future interval logic (FIL) notation as a complementary notation for the specification of intervals. The Property ASSistant (PASS) [22] tool has been implemented to facilitate the specification of event-based systems. The PASS tool provides three notations for the requirements specification, in three complementary languages: natural language summary, μ-calculus and UML-like notations. There are other tools such as CHARMY [23] that support the design and validation of architectural specifications captured in UML. The latest tool called PSPWizard [24] is very similar to PROPEL and Prospect, yet it has a more comprehensive catalog of specification patterns.

Some of the listed tools also provide formal analysis and verification engines integrated into the tool set. The Prospect tool provides means for the run-time verification of properties specified in FIL. CHARMY can be used for the verification of architectural designs using SPIN, whereas the PASS tool provides run-time verification of event-based properties specified in μ-calculus, by using the underlying mCRL2 engine.

If compared to the tools listed in this section, the SESAMM Specifier has several advantages. First, all previously mentioned tools rely on a specific pattern catalog. SESAMM Specifier does not include a predefined catalog, but it is a more general tool built on top of the pattern-based approach. Consequently, the SESAMM Specifier has more expressive power than the rest of the tools. Secondly, the SESAMM Specifier provides a mechanism for providing visual feedback to the users. Although not unique with respect to this feature, our tool provides more options than most tools, and we are currently working on extending the set of available graphical notations for visualizing behavior. What is truly unique about the SESAMM Specifier is the fact that it has been developed with practitioners in the loop. This has resulted in a tool that engineers can associate themselves with, which can have a positive effect on the adoption of the tool in their everyday work.

Table I
Results from the preliminary interviews

Interviewee	Preffered visual notation
1	RTGIL
2	RMF
3	UCMPPS
4	UCMPPS
5	FSA
6	L_R
7	RTGIL

VI. Discussion

Prior to the implementation of the SESAMM Specifier, we have conducted a two tier structured process for determining the features that such tool should encompass. During the first phase, we have performed a literature survey to identify the already existing tools and methodologies based on graphical notations suitable for visualizing formalized behavior. The identified tools and their features have been discussed in Section V. The survey for identifying notations based on graphical elements has yielded the following results: RTGIL, UCMPPS, L_R, FSA and Run-time Monitoring framework (RMF) [25].

In the second phase, we have performed a preliminary study in which seven different stakeholders involved in the requirements engineering process in the company have been asked to use the proposed methods and select one that, in their opinion, captures the formal requirement behavior in the most intuitive way. The results of the preliminary study are given in Table I, showing that the RTGIL and UCMPPS have been selected by two participants, while RMF, FSA and L_R have been found useful by only one interviewee. Based on these results and additional discussions with the engineers, the RTGIL, UCMPPS and L_R notations have been marked as top priority for the implementation in the first version of the tool. The rest of the notations are expected to be implemented in subsequent versions of the tool.

VII. Conclusions

In this paper, we have proposed a tool called SESAMM Specifier for the formal requirements specification in industrial settings, using a pattern-based approach.

The tool has been developed with three main features in focus: engineer friendliness, extensiveness and flexibility. To make the tool usable by industrial practitioners, we have incorporated the feedback obtained by interviewing different stakeholders involved in the requirements engineering process in the chosen company from the automotive domain. The engineers were involved in the design decisions regarding the selection of the features to be implemented in the tool and the ways in which the chosen features were going to be implemented.

Unlike any other existing tool for pattern-based requirements specification, our tool is highly extensible. This has been achieved by providing an expert UI through which an expert practitioner can create a pattern catalog that can be used by different stakeholders, during the process of formal requirements specification. This opens a window of possibilities for customizing the approach to fit the needs of a specific industrial domain (ex: automotive, avionics, railway), or even a sector within a company with specific needs for requirements specification. In addition, our tool supports the "on demand" definition of multiple views expressed in different notations in which the requirements can be specified.

To enable engineers to understand the semantics of the specified behavior better, the tool provides a visualization of the patterns in a number of different notations based on graphical elements. The initial feedback from the engineers confirms our intuition with respect to the positive impact of such a feature on the understanding of the behavior of formally specified requirements. To gather more concrete data, we plan to conduct a study on a larger scale, in which different stakeholders from different sectors of the company will be using the tool to specify selected requirements.

The tool described in this paper represents a first step towards a more comprehensive engineer-friendly framework for formal specification, analysis and verification of industrial system models. To be able to achieve this ambitious goal, the tool will most probably evolve via more design phases in which the specification and visualization features will be improved, as well as new features for the formal analysis of system requirements added.

Acknowledgement

This work was funded by the Swedish Governmental Agency for Innovation Systems (VINNOVA) under the VeriSpec project 2013-01299.

References

[1] J. P. Bowen and M. G. Hinchey, "Ten commandments of formal methods... ten years later," *IEEE Computer*, pp. 40–48, 2006.

[2] "ISO 26262: Road vehicles Functional safety," Geneva, Switzerland, Tech. Rep., July 2011.

[3] A. Post, I. Menzel, J. Hoenicke, and A. Podelski, "Automotive behavioral requirements expressed in a specification pattern system: A case study at bosch," *Requir. Eng.*, pp. 19–33, 2012.

[4] P. Filipovikj, M. Nyberg, and G. Rodriguez-Navas, "Reassessing the pattern-based approach for formalizing requirements in the automotive domain," in *RE'14*, 2014.

[5] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett, "Patterns in property specifications for finite-state verification," in *ICSE '99*. ACM, 1999, pp. 411–420.

[6] K. M. Olender and L. J. Osterweil, "Cecil: A sequencing constraint language for automatic static analysis generation," *IEEE Trans. Softw. Eng.*, pp. 268–280, 1990.

[7] Z. Manna and A. Pnueli, *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag New York, Inc., 1992.

[8] E. M. Clarke, E. A. Emerson, and A. P. Sistla, "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, no. 2, pp. 244–263, Apr. 1986.

[9] S. Konrad and B. H. C. Cheng, "Real-time specification patterns," in *ICSE '05*. ACM, 2005, pp. 372–381.

[10] R. Koymans, "Specifying real-time properties with metric temporal logic," *Real-Time Syst.*, pp. 255–299, Oct. 1990.

[11] R. Alur, C. Courcoubetis, and D. Dill, "Model-checking in dense real-time," *Information and Computation*, pp. 2–34, 1993.

[12] Y. Ramakrishna, P. Melliar-Smith, L. Moser, L. Dillon, and G. Kutty, "Interval logics and their decision procedures: Part i: An interval logic," *Theoretical Computer Science*, pp. 1–47, 1996.

[13] L. E. Moser, Y. S. Ramakrishna, G. Kutty, P. M. Melliar-Smith, and L. K. Dillon, "A graphical environment for the design of concurrent real-time systems," *ACM Trans. Softw. Eng. Methodol.*, pp. 31–79, 1997.

[14] J. Hassine, "Formal semantics and verification of use case maps," Ph.D. dissertation, Montreal, P.Q., Canada, 2008.

[15] I. Lee and O. Sokolsky, "A graphical property specification language," in *In 2nd IEEE Workshop on High-Assurance Systems Engineering*. Society Press, 1997, pp. 42–47.

[16] X. Zhang, M. Persson, M. Nyberg, B. Mokhtari, A. Einarson, H. Linder, J. Westman, D. Chen, and M. Törngren, "Experience on applying software architecture recovery to automotive embedded systems," in *CSMR-WCRE*, 2014, pp. 379–382.

[17] M. Nyberg and J. Westman, "Failure propagation modeling based on contracts theory," in *EDCC*, 2015, pp. 108–119.

[18] J. Westman and M. Nyberg, "Environment-centric contracts for design of cyber-physical systems," in *MODELS*, 2014, pp. 218–234.

[19] R. L. Cobleigh, G. S. Avrunin, and L. A. Clarke, "User guidance for creating precise and accessible property specifications," in *SIGSOFT '06/FSE-14*. ACM, 2006, pp. 208–218.

[20] O. Mondragn, A. Q. Gates, and S. Roach, "Prospec: Support for elicitation and formal specification of software properties," *Electronic Notes in Theoretical Computer Science*, pp. 67 – 88, 2003.

[21] R. L. Smith, G. S. Avrunin, L. A. Clarke, and L. J. Osterweil, "Propel: An approach supporting property elucidation," in *ICSE '02*. ACM, 2002, pp. 11–21.

[22] D. Remenska, T. A. C. Willemse, J. Templon, K. Verstoep, and H. Bal, *Property Specification Made Easy: Harnessing the Power of Model Checking in UML Designs*. Springer Verlag, 2014, pp. 17–32.

[23] P. Inverardi, H. Muccini, and P. Pelliccione, "Charmy: An extensible tool for architectural analysis," in *ESEC/FSE-13*. ACM, 2005, pp. 111–114.

[24] M. Autili, L. Grunske, M. Lumpe, P. Pelliccione, and A. Tang, "Aligning qualitative, real-time, and probabilistic property specification patterns using a structured english grammar," *Software Engineering, IEEE Transactions*, pp. 620–638, 2015.

[25] J. Simmonds, M. Chechik, S. Nejati, E. Litani, and B. O'Farrell, "Runtime verification," M. Leucker, Ed. Springer-Verlag, 2008, ch. Property Patterns for Runtime Monitoring of Web Service Conversations, pp. 137–157.