Guaranteeing the validity of requirements in a safety-critical context like in automobiles or airplanes is of utmost importance as failures of such systems often lead to life-threatening situations. Checking that a system under test meets a set of requirements can be performed at each level of the production, i.e. from the very beginning where just the natural-language requirements are known, over the stages where a system model, a system implementation, and a hardware virtualization exist, up to the final product. Since the process of requirements checking is time-consuming and expensive, in particular if malfunction is detected at some very late point, there is an industrial trend to find system failures very early and preferably in an automatic way. There exist a variety of solutions pioneered by academia, commonly referred to as formal methods, being able to support this trend from a technological point of view. The off-the-shelf application of formal methods in industry however is at an early stage due to the lack of (de facto) standards and the rareness easy-to-use engineering tools.

There are two main challenges to advise non-expert industrial users to deploy the power of formal methods in their projects and applications: first, providing intuitive description languages to formalize requirements given in natural language, and, second, efficient methods to achieve requirements integrity at each stage of the production, preferably just at the touch of a button. In this talk, we present the approach to formalize and verify system requirements as being implemented the upcoming product suite BTC EmbeddedPlatform®, a platform for specification, testing, and verification of requirements for Simulink® and TargetLink® models and production code.

Simplified Universal Pattern. By long-standing experience and cooperation with engineers from prestigious automobile and aircraft manufacturers and suppliers, description languages for formalizing natural-language requirements should be as intuitive as possible and easy to understand, and preferably presented in a graphical way such that formalization of human-readable to machine-readable requirements becomes a common engineering task without being very prone to errors. The universal pattern approach is based on the observation that the vast majority of real-life safety-critical requirements can be expressed by temporal trigger-action relationships like in the textual requirement “If the driver up or passenger up switch is pressed then the window has to start moving up in less than 50 ms”. A universal pattern explicitly introduces artifacts like trigger and action to close the gap between human intuition of a requirement and its formalized description,
Figure 1: Example of a simplified universal pattern

A trigger or action is started by consuming its start event and successfully passed by accepting its end event in the specified time interval, while its condition must hold in between. A trigger or action fails during processing if its condition became false or its end event was not observed in the time interval. An SUP is successfully passed by an execution if its trigger and action is successfully passed by this execution and their temporal relation is met. An SUP is violated if there is some execution passing the trigger, for which the action does not start in the specified time interval or for which the action fails after entering it.

An SUP has three different kinds of interpretation. In addition to progress, meaning that “trigger implies action” as described above, we provide the interpretations invariant, a special case of progress meaning that “trigger implies action at the same time”, and ordering, meaning that “action implies previous occurrence of trigger”. Moreover, an SUP has three different activation modes: init forces immediate occurrence of the trigger, first in contrast waits for the first occurrence of a trigger, while cyclic restarts the SUP observation again and again. The interpretation and activation mode have an effect on the semantics, e.g., for activation mode init, a failing trigger stops the SUP observation (with an “inconclusive” result, i.e., the SUP was neither violated nor successfully passed), while in the same situation the SUP observation is restarted for the other activation modes. We remark that an SUP may have further artifacts like a start-up phase, i.e., SUP observation starts only if some start-up condition is fulfilled, trigger and action exit conditions, i.e., trigger or action can be left by a “good exit”, or a global time scope, i.e., SUP observation is limited in time.

For a small example, consider the SUP from Figure 1. Please note that whenever a trigger or action is an instantaneous event then their respective start event, condition, and end event are equal. In such cases, only the condition is depicted in the graphical SUP description for the sake of clarity, compactness, and usability. One possible SUP execution is shown in Figure 2: in step 2 the expression of the trigger condition driver_up || passenger_up holds as driver_up is true, and thus the trigger is passed. The SUP is then ready to observe the action which happens...
immediately as $\text{Sa1\_move\_up}$ is also true in step 2. The SUP is successfully passed and, due to activation mode cyclic, waits for a new trigger. The next trigger is consumed in step 4 due to $\text{passenger\_up}$. Since the expression of the action condition $\text{Sa1\_move\_up}$ does not hold in the following 5 steps/50 ms (where one step corresponds to 10 ms), the SUP is violated in step 9.

**Formal Test.** Validating requirements by means of testing is a well-established industrial process and always applicable if the system under test can be simulated. BTC EmbeddedPlatform® supports automatic validation of SUPs wrt test executions resulting from Simulink®, TargetLink®, or production code as well as for imported test executions from external architectures, with dedicated test reporting for all considered system architectures. Though extensive and sophisticated tests passed on model and code level are a convincing argument for system safety, the system behavior still is uncertain in its intended embedding (hardware) environment. To achieve more confidence in a more realistic setting, dSPACE offers a wide range of hardware-in-the-loop (HIL) simulators for processing large-scale Simulink® models in real-time and in a multi/many-core environment. BTC EmbeddedPlatform® provides a technical solution of integrating SUP specifications into such HIL systems with the objective of automatic online testing of formalized requirements. It is worth to mention that in contrast to the “ideal” model view, within real-time testing new aspects play an important role like timing tolerances for handling permissible computation and communication delays being easily expressible in SUP by its nature.

We finally mention the use case of self-monitoring which goes one step further: a code implementation of the specification probes the final product, e.g. a car, by monitoring the correctness of the system during lifetime. BTC EmbeddedPlatform® provides source-code export of SUP specifications to support this use case in principle.

**Debugging.** Counterexamples, i.e. system executions violating the specification, are very precious in order to understand and eliminate the malfunction. Owing to its temporal nature, counterexamples of SUPs can be illustrated in a very lucid and comprehensible way, linking each execution step to the current status of the SUP as shown in Figure 2.

**Requirement Coverage.** If formal testing of all requirements was successful, i.e. all SUPs are not violated by the existing test suite, it often remains the question of how comprehensive the requirements are covered by the test suite. The system execution from Figure 2 covers two test scenarios, namely the one where $\text{driver\_up}$ holds but not $\text{passenger\_up}$ and where $\text{passenger\_up}$ holds but not $\text{driver\_up}$ (the latter actually revealing a requirement violation). The test where both signals are true however is not covered. Note that a test scenario where the trigger condition is false is pointless in the sense that the entire SUP is not touched. Thus, a useful notion of requirement coverage should deal with such “interesting” test cases and give a reliable measure of how good the requirements are covered by the test suite.
BTC Embedded Platform offers the notion of \textit{trigger coverage}. Intuitively, we want to measure how many of the different ways a trigger can be processed were actually seen by the formal test. A bit more precisely, from the expressions of the trigger events and condition, new goal expressions are derived that encode “interesting” waypoints through the trigger as indicated in Figure 3, e.g. from the trigger start event $a \lor b$ the goal expressions $a \land \neg b$, $\neg a \land b$, and $a \land b$ are derived. \textit{Trigger coverage} then aims at visiting each of the waypoints of the trigger while afterwards successfully passing (or violating) the SUP. An example of a covering execution is shown in Figure 3. An analogous approach is conceivable to establish the notion of \textit{action coverage}, which usefulness however needs some further investigation together with our industrial partners and customers.

\textbf{Formal Verification.} A significant test suite that fully covers the safety-critical requirements both on model and code level is an important achievement and inspires confidence in quality of the product. It is however clear that testing cannot be exhaustive with the consequence that system errors may remain undetected even if only on rare occasions. To detect such remaining erroneous behavior the system can be exhaustively checked against specification by using so-called \textit{model checking} techniques. In case an error was found by model checking, a counterexample is generated supporting the debugging process. Though model checking is a very active research area in academia and also successfully applied in industry with increasing frequency, it must be often tailored to the corresponding application, domain, or use case for the sake of efficacy and efficiency. BTC Embedded Platform provides model checking technology that is particularly tailored to check TargetLink-generated production C code against SUP specifications. As backend tools, we mainly employ \textit{symbolic} model checkers based on \textit{satisfiability} (SAT) and \textit{satisfiability modulo theories} (SMT) solving as well as \textit{binary decision diagrams} (BDD).

\textbf{Test Case Generation.} Finding test cases that yield a convincing requirement coverage most often is a \textit{manual} and thus very time-consuming task. Moreover, if some test goals are not covered by the test suite then it is not always clear whether the test goals cannot be reached at all or whether some test cases are missing. BTC Embedded Platform \textit{automatically} generates test cases for requirements with the objective of achieving full requirement coverage. Due to the fact that model checking techniques are used, test goals can also be certified as \textit{unreachable}. Test case generation from requirements can be performed at a very early level, namely when just the formalized SUP specifications are known, or at a level where the system implementation is already present.