

Automated Requirements Validation for ATP Software via Specification Review and Testing

Weikai Miao¹(✉), Geguang Pu¹, Yinbo Yao¹, Ting Su¹,
Danzhu Bao¹, Yang Liu¹, Shuohao Chen², and Kunpeng Xiong²

¹ Shanghai Key Lab for Trustworthy Computing,
School of Computer Science and Software Engineering,
East China Normal University, No. 3663 North Zhongshan Rd, Shanghai, China
wkmiao@sei.ecnu.edu.cn

² Testing Departement, Casco Signal Ltd., Mingde International Plaza,
No. 158 Minde Road, Shanghai, China

Abstract. Complete and correct requirements specification is the foundation for developing high-quality Automatic Train Protection (ATP) software. Requirements validation aims at facilitating the completeness and correctness of the specification. In this paper, we propose a novel requirements validation approach combining diagram-guided specification review and scenario-based specification testing for ATP software. The specification is transformed into an executable prototype. Diagrams are generated from the prototype to visualize the interactions between variables for an effective review. To check whether the specification conforms to the user's concerned scenarios of train operation, the scenarios are specified as test cases for testing the prototype. The conformance is then determined via test analysis. Through the review and the testing, the requirements specification is validated. The case study and experiments show that the approach achieves a higher error detection rate and while it reduces the time costs comparing to the traditional review method used by our industrial partner.

1 Introduction

ATP (Automatic Train Protection) software is one of the kernel components of railway transportation system, which performs safety-critical functionalities of a train. The validation of the ATP software requirements must be considered for ensuring the quality of the ultimate software systems, since the requirements specification act as the foundation of the ATP software development.

Requirements validation focuses on checking the completeness and correctness of the requirements specification [1–3]. That is, the requirements specification needs to cover the user's expected functions as complete as possible. Potential scenarios of the target system should be satisfied by the specification. Meanwhile, the requirements specification should not contain logic errors (e.g., inconsistency). Research efforts have been devoted to the requirements validation from both the academic and the industrial communities, including specification review/inspection [4–7], specification testing [18–20] and animation [13, 17]. However, effective requirements validation for the industrial ATP software practitioners is still a challenge.

One major problem is how to validate whether the user's (e.g., the train driver) concerned safety-critical scenarios (e.g., accelerate and then run at a designated velocity) of the train operation are completely and correctly satisfied by the requirements specification. Usually there is a gap between the concerned scenarios and the requirements specification. The specification is established by the requirements analyst and the ATP domain expert, which describes the ATP software from the functional perspective in terms of individual functions. However, such a specification does not explicitly specify the scenarios of running a train in the real world. The practitioner demands effective methods for validating whether the user's concerned scenarios can be satisfied by the specification.

In this paper, we propose a scenario-based specification testing approach to requirements validation of ATP software. The ATP analyst constructs the requirements document using a particular modeling language. To facilitate the automation of requirements validation, the specification is transformed into an equivalent executable ATP prototype. Since ATP software is a large-scale system that involves complex interactions among variables, directly reviewing the specification or the code of the prototype is tedious and error-prone. To tackle this problem, we provide an intuitive diagram-guided specification review technique. The variable dependency diagrams and the state transition diagrams are derived from the prototype. Potential errors of variable interactions are detected via the diagram-guided review. To validate whether the specification satisfies the user's concerned scenarios, we adopt the idea of "model-based testing". Specifically, concerned scenarios of train operations are defined in the scenario document using a designated scenario notation. Then the scenario document is transformed into an executable test script for running the prototype. The satisfaction of the prototype with respect to the scenarios can be evaluated via analyzing the test results. Through the static review and the dynamic testing, the requirements specification is rigorously validated.

To support the automation of the approach, we have developed a supporting tool for applying this approach in practice. We have also carried out a systematic case study for validating the efficacy of the approach in a real ATP software development project of our industrial partner, the CASCO Signal Ltd. of China. The feedback provided by the practitioner and the experimental results demonstrate that the approach can improve the efficacy of the requirements validation and lead to higher productivity of the ATP software development.

The rest of the paper is organized as follows. Section 2 overviews the state-of-art in the area of requirements validation. Section 3 presents the technical details of the approach. In Sect. 4, we present the case study and experiments for demonstrating the efficacy of the approach in real ATP software development. Section 5 summarizes the paper and points out our future research plans.

2 Related Work

Specification review is a classic technique for requirements validation, especially in the industry. To validate whether requirements specification really satisfies the given requirements, the authors of work [7] propose an approach based on the notion of

querying a model, which is built from the requirements specification. Then scenario questions are raised and the results are analyzed for validating whether the derived model's behavior satisfies the given requirements. This work inherits the advantages of the review/inspection process. In the work [5], a mental model is introduced as the foundation for requirements validation. The requirements are aligned with the model and be reviews following certain criteria. A specification review approach based on virtual prototype is proposed in the work [22]. Specification in natural language is transformed into virtual prototype for structural and functional review. In practice, however, when the specification is in large-scale and the logics is complex, directly reviewing or inspecting textual requirements specification may be difficult. The efficacy, to a large extent, relies on the practitioner's experience. To facilitate the specification review, our approach recommends a diagram-based review strategy.

Animation of specification is also used for requirements validation. Specification animation is usually done through model checking and specification execution. ProB is a validation toolset for B formal specification that animates counter-examples by displaying its state transition paths [11, 12]. UPPAAL also animates dynamic behaviors of the target system by model checking [13, 14]. It allows system modeling with states and transitions and explores the state space automatically. In [15], the authors propose an approach to animating tabular specification. In [17], an animation-based inspection approach is proposed where animation is used to guide the inspection process. By demonstrating the relationship between input and output of the selected functional scenarios, the efficiency of the inspection activity can be improved. In fact, the effectiveness of animation relies on the understanding of the intended functions by human.

Specification testing is a promising technique for requirements validation. VDMTools is developed to support the analysis of VDM specifications [16]. It is able to execute a large subset of VDM notations. With test cases generated by certain criteria, VDMTools will show system behaviors by executing the concerned specifications. In [18], the authors propose a tool suite for testing software design specifications using dynamic slicing technology. In [19], the authors use testing modules to test formal specifications. The testing modules are described in the same formalism as the formal specification and can be automatically generated. Liu proposes a specification testing method for reviewing task trees of the target formal specifications [20]. It includes different strategies for generating test cases for different kinds of review task trees. Our approach differs from them in the way of test case generation and the automated test result analysis. The test cases in our approach are generated from dedicated scenarios. The scenario also acts as the foundation of the test oracle of specification testing.

Some researches adopt the formal methods for requirement validation. In the work [8, 9], the authors present the VDM++ specification validation of ATP software. [10] introduces a specification verification technique based on the SCADE platform. However, these approaches require that the practitioner construct the proof obligations using complex mathematics notations. Although in recent years formal methods have been reorganized by the industrial practitioners, exploiting these formal techniques including formal modeling and verification in industry is still challenging. In particular, how the formal methods can benefit traditional industrial software engineering activities such as

specification review and inspection is still a problem. In our approach, we try to support the traditional specification review and testing using the formal specification written in a light-weight formal language for the ATP software. In this way, the precision of formal methods can help the requirements validation without change the practitioner's engineering processes radically.

3 The Approach

3.1 Main Framework of the Approach

Before a detailed illustration of the techniques involved in the approach, we first introduce the framework of the approach, which is described in Fig. 1.

In our approach, requirements are documented using the CASDL (Casco Accurate Description Language) language developed for ATP software modeling. The CASDL is a light-weight formal language for specifying the functions of an ATP system. In the textual specification, each function is specified in terms of the relations between the input and the output variables. In addition, there are some descriptions written in natural language in the specification.

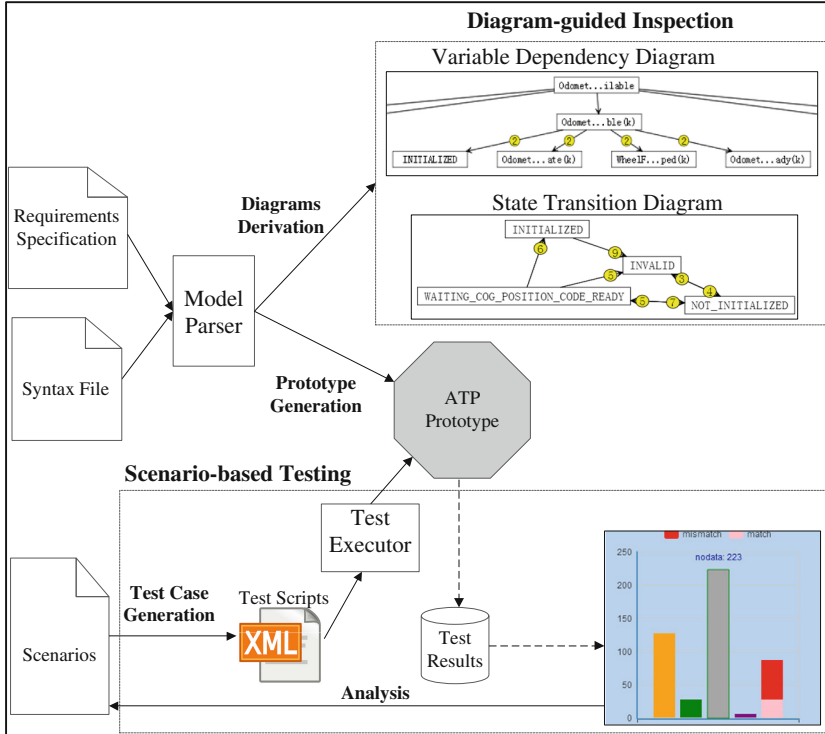


Fig. 1. Framework of the requirements validation approach

Requirements validation is performed via the stages of specification review and testing. To facilitate the automation of requirements validation, the specification needs to be processed into an executable model by removing the natural language comments for effective diagram generation and testing. To this end, we first transform the requirements specification into an equivalent executable model. Firstly, the analyst defines the syntax file for the specification language. Since the specification is basically written using the CASDL, the syntax file here defines the CASDL syntax. The syntax file and the requirements specification are accepted by a dedicated Model Parser. The parser can generate an executable CASDL requirement model, i.e., the prototype of the ultimate ATP software through analyzing the syntax file and the specification.

To facilitate a more effective specification review, we provide a diagram-guided review technique for specification validation. Two diagrams, variable dependency diagram (VDD) and state transition diagram (STD), are generated from the prototype to precisely visualize the interactions among variables and state transitions of individual variables, respectively. Through reviewing the VDDs, the analyst determines whether the relations among the involved variables conform to the expectations. Similarly, whether the state transitions of each concerned variable are correct can be checked via reviewing the STDs.

Specification review only checks the defined requirements statically. To validate whether the requirements specification satisfies the concerned operational scenarios of the trains, we propose a scenario-based specification testing technique. User's concerned operational scenarios are first specified in the scenario document using the CASSL (CASco Scenario Language) language that is designed for describing the train operations in the real world. The scenario documents are then automatically transformed into test scripts for running the ATP prototype. Subsequently, test report is automatically generated. The report visualizes detailed information of the test results (e.g., coverage of requirements) and explicitly describes the consistency between the scenarios and the prototype. Therefore, whether the specification satisfies the expected scenarios can be finally determined based on the test results analysis.

3.2 The Specification and the Prototype Construction

Our approach is language independent. In practice, due to the different characteristics of the target system, various requirements description notations can be used. In our approach, we deliberately design the CASDL modeling language for our industrial partner. The CASDL language is a domain specific requirements specification language for accurately describing the requirements of ATP software. Figure 2 is a simple example of the CASDL requirements specification.

The expected functions of an ATP system are represented by individual requirements items. Each item consists of two parts. The first part is a section of natural language to concisely describe the expected functions. The other part is a section of the CASDL formal description for precisely defining the relations between the input and output variables. Such combination of both natural language and formal notations keeps the balance of readability and precision of the specification. Note that the ATP software is a periodic system (i.e., computation tasks are driven according to the time

```

[ITC_CC_ATP-SwRS-0176]
//If OdometerState is NOT_INITIALIZED at period k, and if wheel stops at k,
then WheelMaximumMovement shall be set to zero.*/

if (OdometerState == NOT_INITIALIZED)
  if (WheelFilteredStopped(k) == True)
    WheelMaximumMovement = 0
  else:
    ...
if ((OdometerState(k-1) = NOT_INITIALIZED)
  and (not WheelFilteredStopped(k) and not UnconsistentSensorTest(k)
  and (WheelFilteredStopped(k-1))
  OdometerState = WAITING_COG_POSITION_CODE_READY
  ...
if ((OdometerState(k-1) = NOT_INITIALIZED) and UnconsistentSensorTest(k))
  OdometerState = INVALID

```

Fig. 2. The sample CASDL specification

period). In Fig. 2, the parameter k explicitly specifies the time period. The ATP software just runs in a sequential non-terminating execution cycle when the power is on. This specification describes the functionalities of the odometer monitoring component of the ATP software. The function is responsible for monitoring the speed of the train and the distance the train has passed. The odometer can switch between certain states. The state transitions of the variable *OdometerState* is regarded as a function in the specification. For instance, it can change from uninitialized state (i.e., *NOT INITIALIZED*) to the state that the meter is disabled or idled (i.e., *INVALID*). In this example, these functionalities are defined in terms of if-else control blocks.

Since the ATP software is a typical large-scale control system, the requirements validation should be automated. To this end, the textual requirements specification is transformed into an executable prototype for automated analysis and specification testing. The transformation focuses on removing all the natural language descriptions; meanwhile, the syntax errors in the specification are detected. We have implemented a Model Parser that uses the ANTLR tool [21] as the foundation for the transformation. Aided by the ANTRL (Another Tool for Language Recognition) tool, the parser can generate an executable CASDL requirement model. Given a language and its syntax, ANTLR can build the ASTs (Abstract Syntax Tree) for analyzing the target language. Our parser derives the prototype through analyzing the ASTs of the CASDL specification.

When the specification is processed, syntax errors of the CASDL specification can be detected. In particular, the circular dependency errors of variables can be automatically detected. Circular dependency error of variables is a typical logical error in ATP software, which refers to the situation that the relations among certain variables constitute a cycle. For instance, the value of variable A relies on the value of variable B . B relies on the value of variable C and C relies on the value of A . In this case, the variables A , B and C constitute a circular relation. From the perspective of data flow, a circular relation refers to a cycle of the data flows between certain variables. Such circular relations need to be detected. Otherwise, the system will be stuck in certain states. We have the additional criterion for detecting such errors.

Criterion 1. *Each variable V specified in the specification (i.e., the prototype) should be checked whether V is involved in any circular dependency relation.*

Let's consider the CASDL statement: $ImmediateNb = ImmediateNb + 1$. The value of $ImmediateNb$ depends on itself and the value 1, which is obviously a circular dependency error. The intention of this statement is to add value 1 to the previous value of $ImmediateNb$ and get the new value. The correct statement should be $ImmediateNb(k) = ImmediateNb(k-1) + 1$ in which the state of $ImmediateNb$ is explicitly differentiated. This is a special case that the circular relation occurs on a single variable.

3.3 Diagram-Based Specification Review

An ATP specification describes the train protection functions by defining the interactions between the variables. Some variables are crucial since they correspond to the kernel functions such as the speed monitoring. Therefore, the analyst needs to validate whether the interactions of these variables with other variables correctly represents the intended relations. Similarly, the state transitions of the individual variables also need to be considered. To this end, a careful and rigorous specification review is demanded.

The VDDs and the STDs are generated from the prototype for precisely visualizing the interactions among the concerned variables and the state transitions of each individual variable. The developer checks the diagrams and determines whether they conform to the expectation. Meanwhile, logic errors in the prototype are also detected.

Definition 1. *A variable dependency diagram of a variable V is a tree structure in which the root node represents V . The child nodes of V is a set of variables ranging from V_child_1 to V_child_n ($n > 0$), which represents that the value of V depends on the n variables.*

The state transition diagrams of each concerned variable can also be derived from the prototype. For the sake of space, we omit the definition of the STD here since the definition is the same to the traditional definition of STD. The following criteria can be applied for the validation.

Criterion 2. *For each concerned variable V , generate the VDD for validating the intended relations between the involved variables.*

Criterion 3. *For each concerned variable V , generate the STD for validating the state transitions.*

Following the criteria, the analyst can validate the requirements from the perspectives of the relations among variables and the state transitions. We use the sample specification shown in Fig. 2 as an example to illustrate the review process. Figure 3 shows a part of the generated variable dependency diagram of the variable *OdometerState*.

The VDD shows that the value of this variable is determined by various variables such as *INITIALIZED*. If some relations conflict with the analyst's understanding of the requirements or some relations are missing, then the analyst can make decision that the

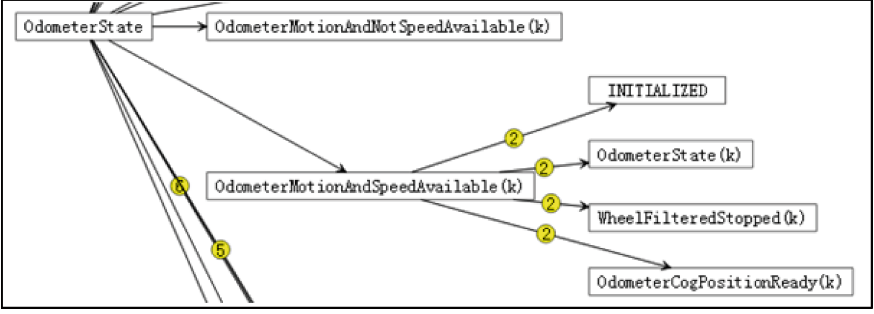


Fig. 3. The VDD of variable *OdometerState*

requirements are incorrect or incomplete. The VDD in this figure is in horizontal manner, since the structure is relatively too large to be displayed in vertical manner. Our tool provides both the horizontal and vertical views of the tree structures.

Similarly, the analyst can review the state transitions of a variable. Figure 4 describes the state transition of variable *OdometerState*.

The diagram shows that the variable cannot directly switch from the invalid state to the initialized state. The analyst and the ATP domain engineer found this missing transition after carefully reviewing the diagram.

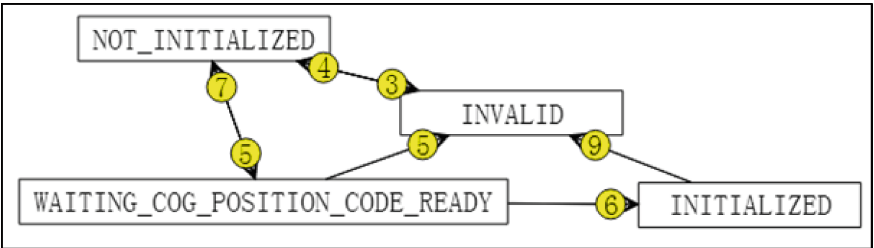


Fig. 4. The STD of variable *OdometerState*

3.4 Scenario-Based Specification Testing

The goal of the scenario-based specification testing is to dynamically check whether the user's concerned operational scenarios of the train are correctly and completely reflected by the specification. In other words, we try to check whether the specification satisfies the expected scenarios for running a train in the real world. The scenario-based testing is carried out in three steps. The first step is to describe the expected scenarios for operating the train. The second step is to transform the scenarios into executable test scripts for running the prototype. Finally, the test results are analyzed for checking the conformance of the prototype with respect to the scenarios. Therefore, whether the specification satisfies the scenario can be determined.

Scenario Representation. The scenarios represent how the train works in the real world. For example, the train driver may concern whether the train can stop at certain position after an acceleration. The scenarios are defined by the user based on domain knowledge. The domain engineer or the user may not familiar with software requirements modeling languages. Therefore, the scenario representation notation should be both precise and easy to understand. Our group and the industrial partner develop a domain specific scenario notation called CASSL for defining the expected scenarios of the train.

Definition 2. *An ATP scenario is a description of train operations, which consists of the train settings, the train actions and the event triggers.*

The train settings refer to the environment information for running the train, such as the routes and length of the train. The train actions are the primitive commands (e.g., stop or accelerate) for running a train. The event triggers refer to the configuration information for describing the reactions of the train with its external hardware (e.g., the signal equipment on the tracks).

The user can freely define the expected scenarios for running a train using the CASSL by setting the parameters values of each involved command. Figure 5 describes a scenario document in the CASSL language.

The train settings section defines the routes of the train using a sequence of blocks on the tracks. The “start” command in the train settings section represents that the train starts from the 0 position of the 247 block. In the train action section, the command *start(@0,30)* indicates that the train starts from the 0 position 30 s later after receiving the command. The train accelerates to the 4 speed units by the acceleration of 1 unit and then stops at position @94.

Test Execution. The user focuses on defining the physical variables such as the speed and the position information for the concerned scenarios. Then the scenarios are transformed into XML test scripts for testing the prototype. For each variable x in the

```
train{
  route: 247, 91, 92, 93, 94, 95, 96, 97, 98, 99, 100
  start: #(247, 0)
  length: 116066
  cog_dir: 1
  direction: 1
}

-> start(@0, 30)
-> acc(1, 4)
-> run(@84)
-> when( #(247, 116066) ):
  ccnv.CCNVRuleEnable=1;
-> stop(@94)
```

Fig. 5. A scenario document for operating a train

scenario, there should be a set of corresponding variables denoted by $R(x)$ in the prototype. Function R represents the associations between the scenario variables and the prototype variables, which is pre-defined by the domain expert and saved as a file in the Test Executor of the tool. For each scenario, the Test Executor deduces the input values for running the prototype by referring to the association.

A scenario variable may correspond to several variables in the prototype. For instance, as shown by Fig. 5, the “run” action takes a variable as input for setting the speed of the train. This variable actually associates to the variables *TrainMaxSpeed*, *WheelMinSpeed* and other variables in the prototype. To run the prototype, we have implemented a CASDL execution engine as a component in the Test Executor.

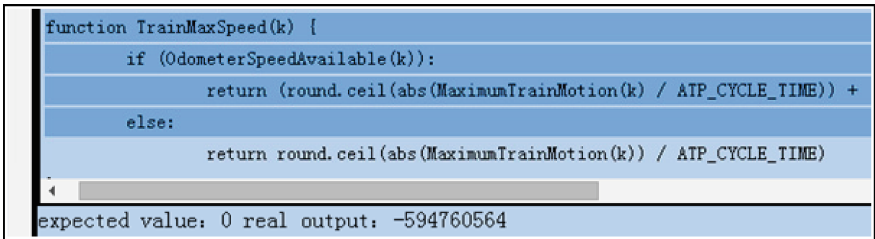
Test Analysis. One major problem in automated test analysis is the construction of test oracle. That is, the expected outputs of the scenario need to be derived. By comparing the expected outputs of the scenario and the real outputs of the prototype, the consistency between the scenario and the prototype can be determined.

The expected values of the variables in the scenario are computed based the physics formulas stored in the Test Executor. The corresponding expected results of prototype can be deduced by referring to the association file. Formally, for each scenario, if the following condition holds, we say that the ATP prototype does not conforms to the scenario.

Condition 1. $\exists x \in S \bullet R(E(x)) \neq P(x)$

The above condition serves as the test oracle. For each variable x of the scenario S , its expected value is evaluated as $E(x)$. Function R represents the associations defined in the association file of the Test Executor. By referring to the associations, the expected values $R(E(x))$ of the corresponding variables of the prototype can be computed. $P(x)$ denotes the execution results produced by the prototype after running the test script. If $R(E(x))$ does not equal to $P(x)$, we say that the prototype does not satisfy the scenario S . Note that we linearise the computation of the kinematic behaviors such as the braking curves in our tool. That is, the engineer focuses on the concerned values at each time cycle.

For the previous example scenario, the expected speed of the train at the destination should be 0 based on the computation formulas. The expected value of its corresponding variable *TrainMaxSpeed* in the prototype should be 0 according to the association file. Figure 6 is the test results.



```
function TrainMaxSpeed(k) {
    if (OdometerSpeedAvailable(k)):
        return (round.ceil(abs(MaximumTrainMotion(k) / ATP_CYCLE_TIME)) +
    else:
        return round.ceil(abs(MaximumTrainMotion(k)) / ATP_CYCLE_TIME)
}
expected value: 0 real output: -594760564
```

Fig. 6. Test result of the scenario

The output value -594760564 after testing obviously violates the expected value 0. That is, the prototype does not implement the scenarios correctly.

4 Experiments

To validate the feasibility and demonstrate the efficacy of our approach, we have applied our approach and the tool in a real ATP software project. An ATP specification is established by the requirements analyst of our industrial partner. The specification is a Microsoft Word file which includes 455 requirements items. Two well-trained engineers participated in the experiments.

4.1 Specification Processing

Taking the syntax file and the textual specification as the inputs, the supporting tool performed the specification processing. The natural language descriptions were removed and the specification was transformed into an executable prototype. The average time for the tool to finish the overall specification (a 450 page document) transformation was about 3 min. During this transformation, 127 syntax errors were detected. Moreover, the tool identified 11 circular relations.

4.2 Diagram-Guided Specification Review

Since an ATP specification contains a huge amount of variables, in practice, the practitioner may only focus on the variables that correspond to the most important functions. In our case study, the practitioner selected 300 variables and generates the VDDs. The tool produced a 2 MB Excel file for storing the generated VDDs. The file contained 46174 rows to record the dependencies of the 300 concerned variables. That is, 300 tree structures were saved in the file. The degree of the deepest trees is 23. The generation of the STDs is similar to the VDDs generation. The engineers were more interested in those variables that associate to kernel functions of the ATP. In our case study, the analyst selected 8 variables to review their state transitions. The most complex state transition relation of a variable includes 5 states and 15 transitions.

4.3 Specification Testing

To sufficiently test the prototype derived from the specification, 317 scenario files were constructed by the ATP engineer. To sufficiently cover the potential operating scenarios of the train, we used real track maps in the case study. All the scenarios were automatically transformed into executable XML test scripts.

4.4 Experiment Results and Analysis

Errors detected from the requirements validation are categorized into four types: syntax error, variable circular error, functional error, and exceptions.

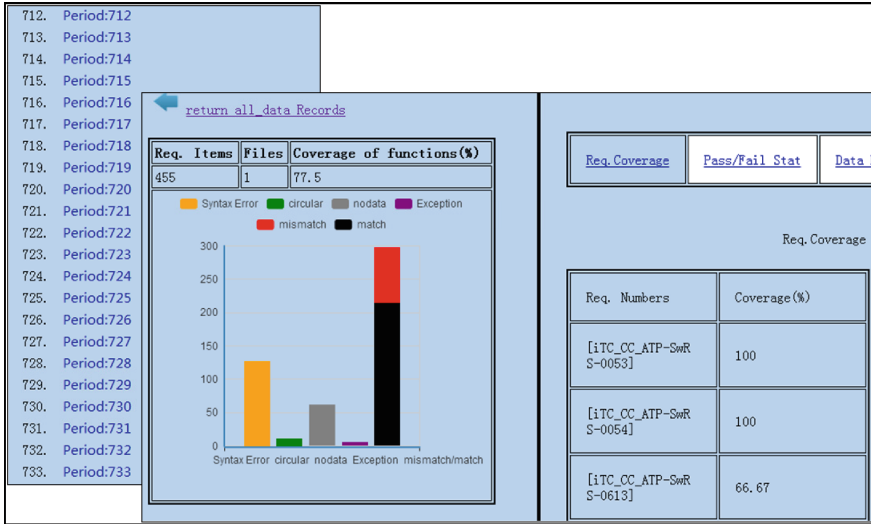


Fig. 7. Test result analysis of the case study

In particular, the functional errors refer to the errors related to the functions that are detected via specification review and testing. The exception errors are basically the unhandled code exception in the prototype. For instance, division by zero is an arithmetic exception. Figure 7 shows the statistics of the case study.

The tool generated both the textual test report and the visualized statistics. The chart shows 127 syntax errors, 11 circular errors, 83 functional errors, 5 exceptions and 61 no data errors were detected through the review and testing phases. Four of the exception errors were arithmetic errors (i.e., division by zero) and one was the out of bound of an array definition. There is a special type of error called *no data*, which is caused by the missing association definitions between the scenario variables and the variables in the prototype.

Our tool estimates the statement coverage of functions in the prototype. This figure shows that the statement coverage of the ATP prototype is 77.5 % after running a certain scenario. For the 317 scenarios used in the case study, the average statement coverage of the functions was approximately 82 %. Table 1 summarizes the comparison of the error detections achieved by the manual specification review and the automated approach.

In general, our proposed approach can reach higher error detection than the manual specification review method applied by the industrial partner. The two engineers reviewed the same specification manually and detected 81 syntax errors. They also detected 2 exceptions in the specification. Similarly, the engineers detect 7 circular errors of variables. For the functional errors, only 27 functional errors were detected through the manual review. Most of the functional errors detected by the engineers were the execution logic errors in individual functions, e.g., the missing of if-else branches.

Table 1. Comparison of the error detection

	Manual review	The automated approach
Syntax errors	81	127
Exceptions	2	5
Circular errors	7	11
Functional errors	27	83

Table 2 compares the time costs of using the two approaches.

Table 2. Comparison of the costs

	Manual review	The automated approach
Syntax and logic errors	0.5 week	12 min
Functional errors	3 weeks	1.5 weeks
Total cost	3.5 weeks	1.5 weeks

To find out the syntax and circular errors, the two engineers spent about 0.5 week in reviewing the specification. By the contrast, the tool only used 12 min to finish the static checking. Most of the time was spent on transforming the Word requirements specification to the executable CASDL prototype and the generation of the variable relations. They spent 3 weeks in identifying the functional errors by manual work while this task was performed within 1.5 weeks aided by the tool. When using the new approach, most of the human efforts were devoted to the construction of the concerned scenarios.

The results of the experiments have convinced us that the automated approach can significantly improve the efficacy and the productivity of the requirements validation for developing the ATP software.

5 Conclusion

Requirements validation is still a challenging problem for the ATP industrial practitioners due to the lack of effective methodologies and powerful tool support. To tackle this problem, in this paper we propose an automated requirements validation approach. Textual requirements specification is transformed into executable ATP prototype. In the static analysis phase, diagram-based review is conducted for validating the requirements from the perspective of variable interactions. The dynamic phase focuses on checking whether the requirements specification can satisfy the user's intended train operation scenarios. We have proposed a particular scenario description language. Test scripts are transformed from the scenarios for running the prototype. The conformance of the specification to the scenarios is evaluated via analyzing the test results. To facilitate the approach in practice, we have also developed a supporting tool for automating the activities involved in the approach.. The case study indicates that our approach is effective in requirements validation and can significantly improve the productivity of ATP software development.

We will continue to develop the approach as a long-term research project. Our future research will be devoted to the aspects including facilitating the association or traceability mechanism between the scenario variables and the prototype variables and the more effective technology for defining high-quality scenarios. The tool support will also be enhanced in our future research.

Acknowledgments. Weikai Miao is supported by NSFCs of China (No. 61402178, No. 61572306 and No. 91418203) and the STCSM Project (No. 14YF1404300). Geguang Pu is supported by China HGJ Project (No. 2014ZX01038-101-001) and STCSM Project No. 14511100400. This work is also partly supported by Japan JSPS KAKENHI (No. 26240008).

References

1. Kotonya, G., Sommerville, I.: Requirements Engineering. Wiley, Hoboken (1998)
2. Nuseibeh, B., Easterbrook, S.: Requirements engineering: a roadmap. In: Proceedings of International Conference on Software Engineering, pp. 35–41, April 2000
3. Wiegers, K.E.: Software Requirements. Microsoft Press, Redmond (2003)
4. Laitenberger, O., Beil, T., Schwinn, T.: An industrial case study to examine a non-traditional inspection implementation for requirements specifications. In: Proceedings of Eighth IEEE Symposium on Software Metrics, pp. 97–106 (2002)
5. Lee, G.Y.K., In, H.P., Kazman, R.: Customer requirements validation method based on mental models. In: 2014 21st Asia-Pacific Software Engineering Conference (APSEC), pp. 199–206, December 2014
6. Sinha, A., Sutton Jr., S.M., Paradkar, A.: Text2Test: automated inspection of natural language use cases. In: 2010 Third International Conference on Software Testing, Verification and Validation (ICST), pp. 155–164, April 2010
7. Aceituna, D., Do, H., Lee, S.W.: SQ2E: an approach to requirements validation with scenario question. In: 2010 17th Asia Pacific Software Engineering Conference (APSEC), pp. 33–42, November 2010
8. Xie, G., Hei, X., Mochizuki, H., Takahashi, S., Nakamura, H.: Model based specification validation for automatic train protection and block system. In: Proceedings of 7th International Conference on Computing and Convergence Technology, pp. 485–488, December 2012
9. Xie, G., Asano, A., Takahashi, S., Nakamura, H.: Study on formal specification of automatic train protection and block system for local line. In: Proceedings of 5th International Conference on Secure Software Integration Reliability Improvement Companion (SSIRI-C), pp. 35–40, June 2011
10. Wang, H., Liu, S., Gao, C.: Study on model-based safety verification of automatic train protection system. In: Proceedings of Asia-Pacific Conference on Computational Intelligence and Industrial Applications, pp. 467–470, November 2009
11. Leuschel, M., Butler, M.: ProB: a model checker for B. In: Araki, K., Gnesi, S., Mandrioli, D. (eds.) FME 2003. LNCS, vol. 2805, pp. 855–874. Springer, Heidelberg (2003)
12. Leuschel, M., Butler, M.: ProB: an automated analysis toolset for the B method. Int. J. Softw. Tools Technol. Transf. **10**(2), 185–203 (2008)
13. Behrmann, G., David, A., Larsen, K.G.: A tutorial on UPPAAL. In: Bernardo, M., Corradini, F. (eds.) SFM-RT 2004. LNCS, vol. 3185, pp. 200–236. Springer, Heidelberg (2004)
14. Vaandrager, F.: A first introduction to UPPAAL. Deliverable no.: D5. 12 Title of Deliverable: Industrial Handbook (2011)

15. Gargantini, A., Riccobene, E.: Automatic model driven animation of SCR specifications. In: Pezzé, M. (ed.) FASE 2003. LNCS, vol. 2621, pp. 294–309. Springer, Heidelberg (2003)
16. Fitzgerald, J., Larsen, P.G., Sahara, S.: VDMTools: Advances in support for formal modeling in VDM. *ACM Sigplan Not.* **43**(2), 3 (2008)
17. Li, M., Liu, S.: Integrating animation-based inspection into formal design specification construction for reliable software systems. *IEEE Trans. Reliab.* **65**(1), 88–106 (2016)
18. Li, J.J., Horgan, J.R.: A tool suite for diagnosis and testing of software design specifications. In: *Proceedings of International Conference on Dependable Systems and Networks*, New York, USA, pp. 295–304 (2000)
19. Brockmeyer, M.: Using modechart modules for testing formal specifications. In: *Proceedings of 4th IEEE International Symposium on High-Assurance Systems Engineering*, Washington, DC, USA, pp. 20–26 (1999)
20. Liu, S.: Utilizing specification testing in review task trees for rigorous review of formal specifications. In: *Proceedings of Tenth Asia-Pacific Software Engineering Conference*, pp. 510–519 (2003)
21. <http://www.antlr.org/>
22. Aceituna, D., Do, H., Lee, S.W.: Interactive requirements validation for reactive systems through virtual requirements prototype. In: *Model-Driven Requirements Engineering Workshop (MoDRE)*, Trento, 2011, pp. 1–10 (2011)