Block-Based Platform for Defining Adaptation Rules for Automotive Systems

Luigi Altamirano
Eindhoven University of Technology
Eindhoven, Netherlands
luigialtamirano@gmail.com

Mauricio Verano Merino
Vrije Universiteit Amsterdam
Amsterdam, Netherlands
Eindhoven University of Technology
Eindhoven, Netherlands
m.verano.merino@vu.nl

Ion Barosan
Eindhoven University of Technology
Eindhoven, Netherlands
i.barosan@TUE.nl

Abstract

Adaptive human-machine interfaces (HMIs) enhance driver safety and comfort by tailoring information presentation. While existing research identifies key adaptation parameters, their practical application remains challenging due to the complexity of rule-based decision making and the limitations of current authoring tools. To address these issues, this paper introduces AHSL, a domain-specific language for efficiently specifying message adaptation logic. By unifying established adaptation parameters within a hierarchical structure, AHSL simplifies rule creation and improves readability. Usability testing demonstrates AHSL's effectiveness in supporting intuitive and efficient decision logic development.

CCS Concepts: • **Software and its engineering** → **Visual languages**; **Domain specific languages**; • **Human-centered computing** → *User interface programming*.

Keywords: rule-based systems, HMI, block-based languages, automotive

ACM Reference Format:
Luigi Altamirano, Mauricio Verano Merino, and Ion Barosan. 2024. Block-Based Platform for Defining Adaptation Rules for Automotive Systems. In *Proceedings of the 3rd ACM SIGPLAN International Workshop on Programming Abstractions and Interactive Notations, Tools, and Environments (PAINT '24), October 22, 2024, Pasadena, CA, USA.* ACM, New York, NY, USA, 16 pages. https://doi.org/10.1145/3689488.3689989

1 Introduction

In recent years, the number of driver assistance and infotainment systems such as Advanced Driver Assistance Systems (ADAS) and In-Vehicle Information Systems (IVIS) [2] have increased. ADAS and IVIS use Human-Machine Interface (HMI) devices (e.g., instrument clusters, touchscreens, and steering wheel controls) to interact with drivers. Traditionally, the information displayed by onboard systems is predetermined by message type (e.g., visual and auditory warnings may communicate a lane deviation event).

Adaptive interaction [13] emerged as a research field to improve driver experience and traffic safety. Existing implementations of adaptive interfaces [5, 6, 8, 23], rely on *rule-based reasoning* to adapt the interaction with the driver in real-time. These projects also include the definition of *parameters* (e.g., driving situations, driver-vehicle interactions, and adaptation decisions). These *parameters* are hard to reuse because they are scattered across multiple documents and projects; therefore, the need for a unified model is evident so that further research can be built upon this model.

Current systems require programming expertise to define adaptive behavior, hindering domain experts from specifying automotive adaptation rules. This paper addresses the following research questions:

How can we ease the specification of rules that describe onboard messages to the driver?

From the main research question we derive the following sub questions.

1. *What parameters are necessary to make adaptation decisions?*
2. *How can a tool provide non-technical users with a user-friendly mechanism to specify executable rules for* HMI *interaction adaptation?*

This paper presents a review of existing adaptive HMI interaction tools (Section 2.1), introduces a DSL for describing automotive HMI interfaces (Section 3), and proposes block-based and text-based editors for domain experts to specify adaptation rules (Section 3.4). The paper concludes with an evaluation (Section 4), testing (Section 7), discussion (Section 6), and final remarks with future directions (Section 7).

2 Background & Related Work

Onboard vehicle systems, such as In-Vehicle Information Systems (IVIS) and Advanced Driver Assistance Systems

PAINT '24, October 22, 2024, Pasadena, CA, USA
© 2024 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-1212-8/24/10
https://doi.org/10.1145/3689488.3689989

(ADAS), allow drivers to interact with the system via Human-Machine Interfaces (HMI), and the way messages and warnings are presented to the driver follow certain rules.

To the best of our knowledge, in practice, Original Equipment Manufacturers (OEMs) have not yet implemented adaptive HMI in production vehicles. However, rule based systems have been used in research projects where adaptive HMI interaction were implemented [5, 6, 8]. Rule-based systems are computer systems that use rules to mechanize the decision-making process of a human expert. To implement an adaptive HMI, a rule-based reasoner is required [5, 6, 8]. In rule-based reasoning, the decision logic is represented as production rules, consisting of premises and actions [15]. For example, in the following sentence, if *it rains* (premise), then *the road gets wet* (action). In this case, rules are atomic if-then statements without secondary paths of execution, i.e., defining an *else* branch is impossible. Thus, the number of rules quickly grows as the decision complexity increases, making it harder to maintain [9]. In practice, different languages are used to specify these rules (e.g., Drools Rule Language, IBM ILOG Rule Language). Nevertheless, each language requires programming experience and knowledge of its particular syntax and semantics. As the complexity of rules increases, the task of maintaining and understanding these languages becomes significantly more challenging, underscoring the intricacy of rule-based systems.

2.1 Adaptive Automotive Interfaces

This section presents an overview of projects that developed adaptive HMIs to improve comfort or safety, and presents the parameters used and their decision categories. Table 1 summarizes the adaptation features implemented in the previous projects.

***COMUNICAR* [5, 7]** integrates multimedia HMI to harmonize messages from ADAS and IVIS systems (e.g., adaptive cruise control, collision warning, navigation, and multimedia). It improves safety, comfort, and driver alertness by choosing when and how system messages are displayed to the driver according to driving conditions and driver workload.

***SAVE-IT* [23]** targets HMI adaptation in two ways. First, it improves the effectiveness of ADAS safety warnings by shortening the reaction time and reducing nuisance without degrading the crash reduction potential. Second, it mitigates IVIS-related distraction by disabling some features and blocking incoming phone calls. Conversely to *COMUNICAR*, *SAVE-IT*'s adaptations decisions and parameters are available, but details about their reasoning system are not.

***AIDE* [2, 6]** focuses on the integration and adaptation of HMIs. They generated the required knowledge and HMI technologies to safely integrate *ADAS*, *IVIS*, and nomad devices in the driving environment.

```
rule "microfinance"
when
   The loan with loan_amount is smaller than or equal to 5000
   and duration_year is no more than 3
   The customer with monthly_income is greater than 2000
   and credit_report is "good"
then
   The loan will be approved
   Set the interest_rate of the loan to 0.025
end
```

Figure 1. Example of a rule written using Drools DSL [16].

ADAS & Me [8] develops an adaptive ADAS that automatically transfer control between the vehicle and driver to ensure safer road usage. A holistic approach that considers automated driving with information on driver state in the environment is applied.

2.2 Rule Definition Software

Rule-based reasoning is a common approach to implementing adaptive HMIs. This section analyses the out-of-the-box Drools rule creation mechanisms [19], which is a popular Java-based Business Rule Management System (*BRMS*).

Drools has a native rule language called *drl*. This language, although powerful, might be complex for end-users. Its complexity lies in the language's syntax because writing a rule requires several lines of code and previous programming knowledge. Nevertheless, Drools provides user-friendly manners for defining rules by supporting different abstractions over *drl* such as the *Drools DSL* and *Decision Tables* [18]. *Drools DSL* allows business users to employ domain-oriented concepts to write simple rules, which is helpful for non-technical users because they do not have the technical jargon of *drl*. Figure 1 shows an example of a *drl* rule suitable for non-programmers.

However, this requires the manual definition of a dictionary containing the mapping between business and *drl* concepts. Moreover, Drools does not provide a mechanism to edit the rules after they are created [16]. An alternative is to create a decision table with two sections: (1) *RuleSet* contains information that affects all rules, (2) the *RuleTable* contains rule templates, where at least each column represents a condition or an action, and each row in the *RuleTable* becomes a *drl* rule. Drools abstractions have a common limitation. Each rule in Drools DSL maps to one in *drl*. The same happens with each row in the *RuleTable* section of a decision table. Rules are essentially atomic *if-then* statements without secondary execution paths. Consequently, these abstractions do not solve the rapid growth of rules when the complexity of the decision logic increases [9].

3 AHSL: DSL for HMI Rule Definition

AHSL [1] is a DSL for specifying HMI adaptation rules with a user-friendly, compact, and consistent syntax for domain experts. Also, it allows users to execute adaptation rules for

Table 1. *HMI* Adaptations in different research projects [5, 6, 8, 23].

Project	HMI Adaptation Feature	Decision Category
[5]	Postpone message to avoid distracting the driver	Change Timing
	Select message modality to avoid information overload	Select Modalities
[6]	Postpone message to avoid distracting the driver	Change Timing
	Increase intensity of an auditory warning/message to be noticeable	Set Auditory Salience
	Increase intensity/repetition of warning/message to be noticeable	Set Auditory Salience
	Non-Visual Presentation of a message to avoid distraction	Select Modalities
	Present the warning earlier to elicit a timely response	Change Timing
	Use an enhanced modality to elicit a timely response	Select Modalities
	Suppress a warning to avoid annoying the driver	Suppress
[23]	Suppress a warning to avoid annoying the driver	Suppress
	Present the warning later to reduce annoyance to the driver	Change Timing
	Present the warning earlier to elicit a timely response	Change Timing
	Present warning only visually to avoid annoying the driver	Select Modalities
	Present visual-auditory warning to elicit timely response	Select Modalities
	Present warning also in the center console to improve reaction time	Select Visual Device(s)
	Limit the functionality in the GUI to prevent driver distraction	GUI Functionality
	Suppress incoming calls to prevent distraction	Suppress
[8]	Use louder sound notifications to be noticeable	Change Salience
	Present visual-haptic feedback to be noticeable	Select Modalities
	Change GUI colors and sounds to improve user experience	GUI Display
	Change GUI content and format to reduce information overload	GUI Functionality

automotive HMIs. Their syntax uses domain concepts rather than programming concepts, which might help domain experts in the rule definition process. In addition, AHSL offers two editors, text and block-based editors, and uses *Drools* as the rule execution engine.

3.1 AHSL Syntax

AHSL's syntax consists of compact structures called *assign statements* that define the conditions to assign values to each parameter. Listing 1 shows an example of the conditions to determine the priority of an onboard message. We assume that if the "safety criticality" of the message is *high*, then the message's *priority* is *high*. If the "safety criticality" is not high but the "time criticality" is high, then the *priority* of the message is *moderate*. Otherwise, the *priority* is *low*.

The `AssignStmt` is a control structure that provides multiple execution paths. A *production rule*, on the other hand, is essentially an *if-then* statement that provides a single execution path. Listing 2 shows that three *if-then* statements (production rules) are needed to describe the same example.

AHSL provides alternative execution paths, resembling *if-then-else* structures but with simpler syntax. By prioritizing parameter visualization and minimizing code redundancy, it enhances readability and maintainability for domain experts.

3.2 Grammar

Listing 3 shows AHSL's grammar. A program (Program) consists of one or more assign statements (AssignStmt). Each assign statement begins with keyword *assign* followed by the

Listing 1. Example of a message priority using `AssignStmt`.

```
assign priority
  high when
    safety-criticality is high
  moderate when
    time-criticality is high
  default low

assign traffic-risk
  high when
    vulnerable-road-user
  default low

assign driving-demand
  default moderate

assign suppress-message
  true when
    priority is not high
    traffic-risk is high
    driving-demand is not low
  default false

assign use-visual
  false when
    traffic-risk is high
  default true

assign display-hud
  true when
    driving-relevance
  default false
```

Listing 2. Specification of the message priority using *if-then* statements.

```
if(safetyCriticality == "high"){
   priority = "high";}

if(safetyCriticality != "high" && timeCriticality == "high")
    {
   priority = "moderate";}

if(safetyCriticality != "high" && timeCriticality != "high")
    {
   priority = "low";}
```

target parameter's name, zero or more conditional statements (ConditionalStmt), and one default statement (DefaultStmt). Each conditional statement consists of a value (Value) and one or more condition lines (ConditionLine). A condition line has one or more conditions (Condition) that are combined using an `or` operator. To illustrate this, Listing 1 shows an example of the usage of multiple conditions in the assign suppress-message parameter. The syntax definitions of non-terminals Parameter and Value are not shown.
To make AHSL simpler for end-user, we give it a natural language appearance, and reduces as much as possible the usage of special characters commonly found in GPLs such as the curly brackets (`{}`), parenthesis(`()`), quotation marks (`?`), semicolons (`;`), among others. The only exception is the usage of hyphens (`-`); it is used to separate words in parameter and value names. Table 2 shows a summary of AHSL operators.

Conditional Operators. An AssignStmt contains conditional statements, therefore, *Operators* are needed to describe the *condition* and *action* of a conditional statement. Additionally, there are three types of operators (*Assignment Operator*, *Equality Operators*, and Conditional Operators) that are used to combine conditional statements.

- *Assignment Operator.* An assignment operator is the operator used to assign a new value to a variable. They are needed to define the *action* part of conditional statements. Support for the simple *assignment operator* is implemented using the keyword *assign*.
- *Equality Operators.* Equality operators are used to test the values of parameters individually. Support for the equality operator `equal to` and the inequality operator `not equal to` is required. The equality operator is represented using `is` and the inequality operator using `is not`. In the case of Boolean parameters, `is` is omitted.
- *Conditional Operators.* The *decision nodes* in the *unified decision hierarchy* depend on the values of other parameters. Driving demand, for example, depends on the values of up to 15 parameters. Each condition tests the value of a parameter. Conditional operators provide the capability to combine conditions. Thus, we include support for the `or` and `and` operators. Condition(s) in a ConditionLine are combined using `or`. while a ConditionLine(s) in a ConditionalStmt are combined implicitly using `and`.

Table 2. Operators used in AHSL

Operator	Syntax	Example Code
Assignment	**assign** Parameter Value	**assign** driving-demand high when (...)
Bool Equality*	**when** Parameter	high **when** bend
Bool Inequality	**not** Parameter	low **when not** intersection
Enum Equality	Parameter **is** Value	low **when** situation **is** following
Enum Inequality	Parameter **is not** Value	high when traffic-density **is not** low
Conditional AND*	ConditionLine ConditionLine	high when situation **is** overtaking fog
Conditional OR	Condition **or** Condition	high **when** rain **or** night

(*): Implicit Operator

3.3 Code Generator& Execution

This section describes the *core features* and *support features* of the code generation, as well as the requirements related to the code generator, are derived. Table 4 shows a summary of the five code generator requirements. The DSL shall generate source code that can be executed on a rule engine. A complete transformation from AHSL source code to DRL is implemented. The generated source code is executable *as is* and conforms to the concrete syntax of DRL.

3.3.1 Core Features.

The core features are derived from the code generator requirements (Table 4) and are explained as follows.

Code generation. At a high level, this is achieved by converting each AssignStmt into one or more rules. Each ConditionalStmt and DefaultStmt becomes a rule. The conditions and actions specified in the elements of an AssignStmt become the LHS and RHS of each rule.

Assignment Integrity. The rules generated from the same AssignStmt preserve their atomic behavior. This means that ConditionalStmt(s) are evaluated in descending order, and that the DefaultStmt is evaluated last. Additionally, a maximum of one rule per AssignStmt should fire. This is achieved by using the *activation-group* and *salience* rule attributes [19]. The *activation-group* guarantees that only one rule within a group fires. The rules generated from one AssignStmt have the same *activation-group* name. The *salience* prioritizes the execution of a rule within an *activation-group*. The rule generated from the first ConditionalStmt has a higher salience. The rule generated from the DefaultStmt has the lowest salience.

Decision sequence. Rule engines handle the firing of related rules in what is called *rule chaining*. Listing 4 shows an example of chaining. The first two AssignStmt(s) change the values of *priority* and *traffic-risk*, which are included in

Table 4. Code generator requirements.

ID	Name	Specification	Type
1.3.1	AssignStmtIntegrity	When the rules execute, the DSL shall enforce that rules from a single AssignStmt fire abiding by its control flow.	*Func.*
1.3.2	Decision Sequence	When the rules execute, the DSL shall enforce that rules fire in the decision sequence described in the *u.d.h.*	*Func.*
1.3.3	Gate Conditions	When the rules execute, the DSL shall enforce that rules fire abiding by the gate conditions derived from the *u.d.h.*	*Func.*
1.3.4	Code Tracing	The DSL shall provide information about the mapping between DSL code and the generated rules.	*Func.*
1.3.5	Firing Log	After the generated rules execute in Drools, the DSL shall provide information about the rule firing order.	*Func.*

the condition part of the third AssignStmt. The *agenda-group* rule attribute is used to categorize rules in groups (i.e., *SEQ1*, *SEQ2*, *SEQ3*, and *SEQ4*) that are evaluated sequentially.

Gate conditions. The rules that modify parameters constrained by the value of other parameters shall not fire. These *gate conditions* are specified in Figure 2. For example, when *suppress-message* is true, *use-visual* is never true. This is achieved with *gate conditions*. A gate condition is added to the LHS of the affected rules.

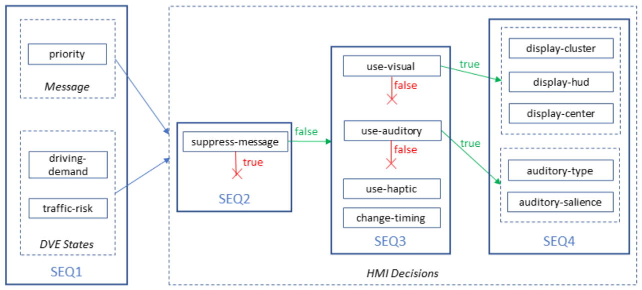


Figure 2. Decision sequence and gate conditions derived from the *unified decision hierarchy* .

3.3.2 Support Features.

To enhance user understanding, we implemented *code tracing* and *firing log* functionalities. *Code tracing* maps rules to their corresponding source code elements (`AssignStmt`, `ConditionalStmt`, or `DefaultStmt`) via rule names. The *firing log* records rule activations, including modified parameters and new values. Figure 3 outlines the code generation process, which iteratively processes `AssignStmts` in a `Program` while incorporating these support features.

Execution Engine. To execute the rules written in AHSL, it is necessary to generate a Drools project as shown in Appendix C; we used *Eclipse IDE v2021-06* with the *Drools plug-in*

3.4 Language Editors

This section presents AHSL's built-in editors. The language design and tooling are essential for a language's success and adoption. A DSL should offer a user-friendly editing environment. AHSL offers two editing environments: a text-based editor and a block-based environment. The first one is

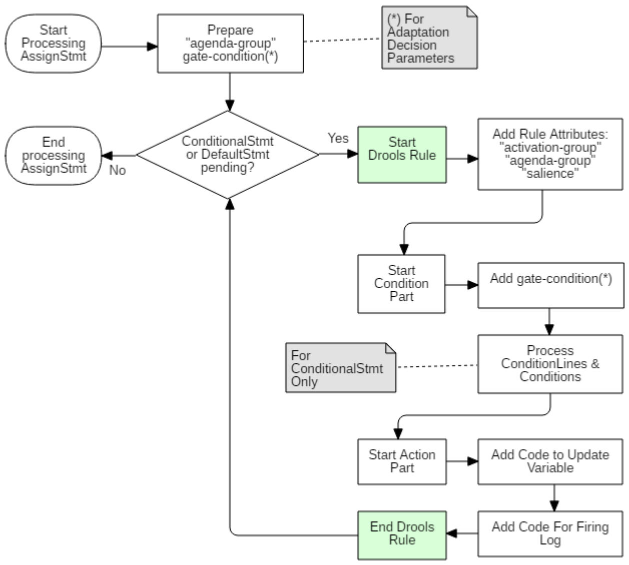

Start Processing AssignStmt
Prepare "agenda-group" gate-condition(*)
(*) For Adaptation Decision Parameters
End processing AssignStmt
No
ConditionalStmt or DefaultStmt pending?
Yes
Start Drools Rule
Add Rule Attributes: "activation-group" "agenda-group" "salience"
Start Condition Part
Add gate-condition(*)
For ConditionalStmt Only
Process ConditionLines & Conditions
Start Action Part
Add Code to Update Variable
End Drools Rule
Add Code For Firing Log

Figure 3. Code generation algorithm of one AssignStmt.

built-in Rascal, which brings a text-based editor almost for free and is used to create and edit AHSL programs textually. It allows connecting directly with the language's parser to check for syntax correctness while the program is being typed.

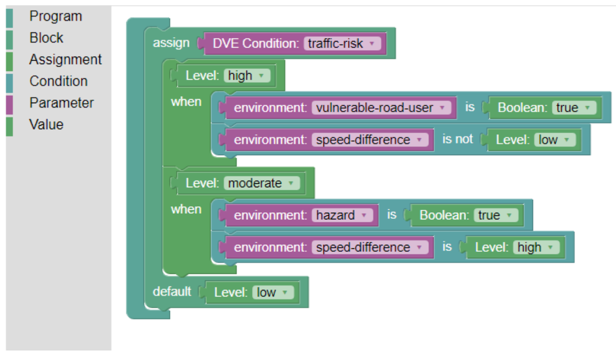
The block-based environment is obtained using Kogi [4, 20–22], which is a tool that allows language engineers to generate a Google Blockly [17] block-based editor from a context-free grammar. In this case, we used AHSL's grammar as input, and the generated block-based environment is shown in Figure 4.

4 Evaluation

This section presents an evaluation of AHSL using two techniques, usability and functional testing. For the first, we designed a questionnaire and recruited three participants to measure the usability of the proposed language.

4.1 Usability Testing

Usability is the question of how well users can use the functionality [14]. DSL usability by domain users is a key factor for DSL adoption [3]. Usability testing involves test users performing a specified set of tasks. Two requirements are



```
assign DVE Condition: traffic-risk
  Level: high
  when environment: vulnerable-road-user is Boolean: true
       environment: speed-difference is not Level: low
  Level: moderate
  when environment: hazard is Boolean: true
       environment: speed-difference is Level: high
default Level: low
```

Figure 4. Sample AHSL program containing one AssignStmt in the block-based environment.

related to usability: (1) The DSL shall provide a user-friendly syntax designed for domain users. (2) The DSL shall provide a user-friendly editor.

4.2 Tools

Participants answer a questionnaire that comprises Likert scale questions to assess the editors using *usability heuristics* and open-ended questions to identify characteristics of the DSL along the *cognitive dimensions* [10]. In addition, a *Thematic analysis* is used to analyze the responses to open-ended questions.

Cognitive Dimensions. The cognitive dimensions (CDS) are a tool to evaluate the usability of information-based artefacts using a broad-brush treatment rather than a lengthy analysis [10]. The target artifact is evaluated along 13 different cognitively relevant dimensions. The 13 cognitive dimensions are available in Appendix B.2. The CDS can be used with non-interactive artifacts such as *computer languages*, and they assess the manipulation of computer programs by distinguishing four types of activities. (1) *Incrementation* refers to adding a new element to a program. (2) *Transcription* refers to converting statements to program elements. (3) *Modification* refers to changing existing program elements. (4) *Exploratory design* refers to programming on the fly or "hacking".

Thematic Analysis. The thematic analysis process described in [11] is adapted to analyze the open-ended question responses of the questionnaire. The objective is to find the features of the AHSL syntax and editors that affect end-users usability. The adapted methodology of the thematic analysis are:

- The questionnaire responses are printed to familiarize with the data. Paper is a convenient medium to detect reoccurring topics in the participants' answers.
- Relevant phrases in user responses are marked. Relevance is determined by a feature that positively or negatively affects usability. These phrases are condensed without altering their meaning and written in a column. These phrases become *Codes*. The cognitive dimension that each question is assessing is written down.
- Patterns in the codes are identified to come up with initial themes. *Themes* are broader than codes. Usually, several codes are combined into a single theme.
- In an iterative process, themes are redefined to accommodate for more codes or split into more themes if the codes contained are not cohesive enough.
- Additional information is identified for each code. The valence, which indicates whether the user is stating something "good" or "bad" about AHSL, is written down. The editor, the participant referred to in the response, is identified.
- A report which includes the AHSL characteristics (i.e., themes) that affect usability and the number of times a characteristic is mentioned is produced.

4.3 Methodology and Materials

Usability testing involves test users. This type of testing is essential because it lets us understand how other users besides the developer interact with our DSL. The testing materials can be found in Appendix B.

Participants. Participants were healthy adults with programming experience who visited Eindhoven University of Technology to complete a 60-minute evaluation of our software prototype and questionnaire. Due to prototype limitations and the need for a coding-familiar user base to assess our language, we recruited four participants with over two years of programming experience. All participants were male, aged 23-35.

Testing Procedure. Each participant is scheduled at a separate time. Participants sign a consent form and receive a ten-minute presentation of AHSL. They are provided with a computer containing the Rascal text-based editor and a block-based environment rendered as an HTML file on a web browser. Participants are requested to perform four **user tasks** and to ask questions to the facilitator at any time. When they complete the tasks, participants answer the **questionnaire**. Finally, the user testing session ends. The user tasks and questionnaire are described below.

User Tasks. A summary of the four user tasks is presented below.

- *Understanding a program in the text-based editor.* Participants are provided with a text-based editor, file *Task01.ahsl*, and input parameter values. They are requested to manually calculate the resulting parameter values according to the code and input values. This task is presented first to familiarize participants with AHSL.

- *Modifying a program in the text-based editor.* Participants are given instructions to change some statements in file *Task01.ahsl* using the text-based editor.
- *Creating a program in the text-based editor.* Participants are given instructions to create statements that assign the values of four parameters using the text-based editor.
- *Creating a program in the block-based environment.* Participants are provided with instructions to create one AssignStmt using the block-based environment.

Questionnaire. The questionnaire has 27 questions (see Table 9) and consists of two parts:

1. Questions 1-6 request the respondent to assess the text-based editor and the block-based environment using six usability heuristics. Heuristics 1, 2, 9, and 10 are not assessed because they are not relevant for code editors or because the editors do not have features related to these heuristics. Instead, we use one Likert scale question with a scale from 1 (bad) to 5 (excellent) to assess each heuristic [25].
2. Questions 7-27 encourage the respondent to identify characteristics of AHSL that are relevant to each cognitive dimension. Ten cognitive dimensions are selected. Cognitive dimensions 1, 11, and 12 were not assessed because the language lacks these features. There are two questions per dimension and one asks the respondent to suggest ways to improve the design of AHSL, regardless of the dimension.

User Study Approval. The usability testing was approved according to Eindhoven University of Technology guidelines for user studies with minimal risk.

4.4 Results

This section presents and analyzes the participant's responses to the usability questionnaire.

4.4.1 Usability Heuristics (Q. 1-6).

The mean of the Likert scale responses assessing the editors' compliance with the usability heuristics is shown in Figure 5. Below we present the main findings.

- Both editors scored well in heuristics *user control and freedom*, *recognition rather than recall*, and *aesthetic and minimalist design*.
- The highest rating overall was given to the text-based editor in terms of *Consistency*. We believe this is due to the consistency of the AssignStmt(s) and language operators' usage.
- The block-based environment was rated much better in *prevent errors* than the text-based editor. This is probably because the block-based environment ensures by design that only syntactically compatible blocks are connected. The text-based editor, on the other hand, does not prevent syntax errors but warns after they occur.
- Users rated the text-based editor higher than the block-based environment in *flexibility and efficiency of use.* This can be explained by the fact that participants had programming experience and were used to coding in plain text.

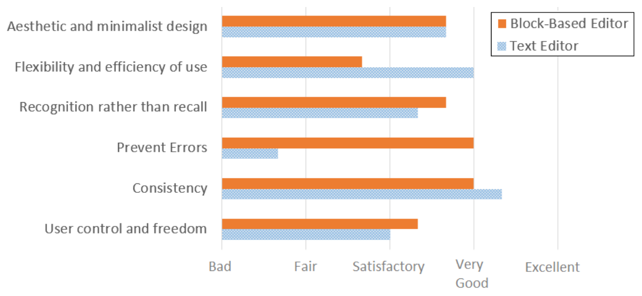


Figure 5. User assessment of editor compliance with usability heuristics. Source: mean of responses to questions 1-6.

4.4.2 Cognitive Dimensions (Q. 7-27).

The responses of the three participants gave a total of 1,520 words of unstructured text. The responses were analyzed using a *thematic analysis.* The 71 codes identified are found in Table 10. Each code is assigned a theme—themes groups characteristics of the text-based editor or the block-based environment that affect usability. The themes identified and the number of times they were mentioned for the text-based editor and block-based environment are shown in Table 5 and Table 6 respectively.

Themes related to text-based editor identified in Table 5 are described below. Usability issues have a negative symbol (−) on the left.

1. *Easy notation.* This theme groups comments related to the notation being clear, easy, simple, and consistent.
2. *Natural language.* Groups comments mention the similarity between the notation and natural language.
3. *Assign cohesiveness.* Groups responses that are related to the fact that the elements of the AssignStmt form a united whole.
4. *Minimalistic.* Groups responses that mention the briefness or minimalism of the notation.
5. *Easy to change.* Related to comments that mention the easiness of editing the internal elements of an AssignStmt.
6. (−) *Using parameters.* Groups comments related to the difficulty of using the parameters.
7. (−) *Large conditions.* Related to the fact that writing and comprehending conditions becomes more difficult as condition complexity or size grows.
8. (−) *Action before condition.* Related to the inverted order of actions and conditions.

Table 5. Assessment of text-based editor characteristics that affect usability, including examples of user responses.

	Theme	Examples of User Responses	Freq.
Positive	Easy notation	*notation is clear, easy, simple, consistent*	10
	Natural language	*sounds like pure English, reads as natural language, can become documentation*	5
	Assign cohesiveness	*closely related elements, conditions and results in neat blocks, dependencies visible*	4
	Minimalistic	*can express briefly, notation minimalistic*	3
	Easy to change	*easy to change value assigned and condition, target parameter*	2
	Using parameters	*can't see parameter type when typing, typos in names, forgetting parameter and value names*	6
Negative	Large conditions	*many "or"s create wider code, comprehending long conditions*	5
	Action before condition	*think first about target parameter, difficult to start with assignment instead of condition*	3
	Assign inter-dependency	*dependencies between related parameters, must think to assign parameters used in other blocks*	2
	Unfamiliar notation	*different from experience, using "is" instead of "="*	2
	Implicit and operator	*remembering that new line is "and", strange that "and" is not expressed in words*	2

Table 6. Assessment of block-based environment characteristics that affect usability, including examples of user responses.

	Theme	Examples of User Responses	Freq.
Positive	Drop-down menu	*changing parts of the notation is more or less easy because there is a dropdown*	3
	Copy-paste blocks	*the structures can be copied and pasted*	2
Negative	Using parameters	*difficult to navigate the left bar and find the blocks and variables you are looking for, there is no searching feature in the block-based interface*	11
	Frequent drag and drop	*feels long winded, a lot of dragging and dropping when first starting a project*	4

9. (−) *Assign interdependency.* Groups the phrases pointing to the lack of visibility of dependencies between blocks related by chaining.
10. (−) *Unfamiliar notation.* Groups comments related to the lack of familiarity with the operators used in AHSL.
11. (−) *Implicit and.* Related to the difficulties caused by the *and* operator not being explicitly expressed in the code.

The themes related to *block-based environment* identified in Table 6 are described below.

1. *Drop-down menu.* Groups comments related to the benefit of using the drop-down menus in blocks to select a parameter or value without typing it down.
2. *Copy-paste blocks.* Related to the feature that enables reusing blocks by copying and pasting them in the canvas.
3. *Using parameters.* Related to the difficulty in navigating the left pane (i.e., toolbar) to find the parameters and values needed to build statements.
4. *Frequent drag and drop.* Related to the repeated need to drag and drop blocks from the toolbar to the canvas to write a program.

This section presents a thematic analysis of open-ended responses assessing cognitive dimensions. Notably, themes often encompassed codes from multiple cognitive dimensions. For instance, the *easy notation* theme included codes from questions on *viscosity, diffuseness, closeness of mapping, role expressiveness*, and *consistency.*

text-based editor feedback primarily focused on syntax-related usability. Participants appreciated the clear, concise, and natural language-like syntax. The cohesive *assign* block structure was also praised. However, implicit *and* syntax, inconsistency, and the inability to visualize `AssignStmt` interdependence were criticized. Additionally, unfamiliar notation and the action-before-condition structure posed challenges.

Both text-based editor and block-based environment lacked features like suggestions, auto-complete, and parameter search. Consequently, parameter usage was a prevalent usability issue.

The themes identified block-based environment feedback centered on editor functionality. While drag-and-drop was often frustrating, drop-down menus and copy-paste features were beneficial.

5 Functional Testing

To assess AHSL functionality, we conducted manual black-box testing by generating AHSL code. This system testing involved crafting a AHSL script, creating the rule file, and executing the resulting rules using the Drools rule engine.

5.1 System Testing

System tests verify both correct code generation and the accurate execution of generated rules in the target environment. Unlike unit testing, our focus is on "happy path" scenarios without anticipated errors.

We tested features aligned with code generator requirements, using an AHSL program (Listing 4), two input data scenarios (Table 8), and their corresponding manually calculated expected outputs. Successful test cases matched rule execution results in the target environment with expected outputs. Table 7 details two test cases, each with a unique scenario and expected result.

The testing steps and results are described below.

1. Create the *st.ahsl* file using the text-based editor.
2. In Rascal, execute the code generator to generate rule file *st.drl*.
3. Manually copy the *st.drl* file to the Drools Project.
4. In class *ExecuteRules*, assign the initial parameter values and fire the rules. This step is repeated again for the second scenario.

The input values and rule firing log are shown in Table 8. Using this set of rules, the presence of a *vulnerable road user* (e.g., a cyclist) caused the evaluation of *traffic risk* to be high. This, in turn, led to the system-initiated message being suppressed by the adaptive HMI system. The behavior was as expected; as a result, both tests passed.

5.2 Comparing Code Size

We compare the program size between a source file written in AHSL (Listing 4) and the generated Drools rule file (Listing 5). We use the source lines of code (SLOC), which is one of the most commonly used metrics for comparing [24] code. For the same functionality, the source *.ahsl* uses 31 SLOC while the generated *.drl* uses 161 SLOC. Based on this metric, AHSL has 5.19 times less complexity than the generated DRL code.

We also compare the number of statements used in both languages. In AHSL, we count AssignStmt(s) and in DRL we count *rules*. In the same example, AHSL needed 6 statements while DRL needed 12 rules. In this case, AHSL requires half the number of statements to specify the same conditions.

In summary, we evaluated AHSL's usability and functionality. On one hand, the main findings of usability testing are:

- The simplicity of the syntax, its closeness to natural language, and its minimalist design were frequently rated positively by the users. These characteristics were particularly evident when using the text-based editor.
- However, some issues related to the internal structure of the *assign statement* were reported. The implicit use of the *and* operator and having to write the action before the condition was sometimes confusing for participants. On the other hand, the potential growth of condition complexity concerned participants. Additionally, the lack of visibility of the dependencies between assigned statements was reported as an issue.
- The main advantage of the block-based environment over the text-based editor was its ability to prevent syntax errors. Its primary disadvantage, however, was its inefficiency. Navigating the toolbar to find language elements and dragging and dropping them was too repetitive.
- The most reported usability issue for both editors was the use of parameters. In the text-based editor, participants had to locate the parameter name and values in a separate list of parameters and type character by character without assistance (e.g., auto-complete, suggestions). In the block-based environment, participants had to find the parameters and values by navigating through categories in the left pane or toolbox.

Functional testing yielded results consistent with AHSL's design. We assessed the parser, well-formedness checker, and code generator, emphasizing the behavior of generated rules within the Drools environment. Notably, AHSL code is over five times less complex than DRL code in terms of SLOCs.

6 Discussion

Usability testing with users revealed that the cohesiveness and clarity of the textual notation improved usability. All three users reported that the notation was close to natural language, minimalist, and simple. These results show that the language design decisions contributed to the usability of AHSL. On the other hand, the use of implicit `Conditional-AND` and the hidden dependencies between assigned statements were mentioned as issues, although less frequently. Overall, user responses indicate that the language syntax contributed to usability. However, the limited number of users and the background of the participants who took part in the user study is a limitation of these results.

The editors used for testing were bare-bone versions of a text-based editor and a block-based environment. The text-based editor provided by Rascal was used *as-is* and not fine-tuned to improve the users' experience. Although it provided syntax checking, it did not have standard text-editor features such as auto-completion or suggestions. Likewise, the block-based environment has several usability limitations as well. The editors, unrelated to the syntax, negatively affected the usability assessment of the users. The lack of assistance provided by the editors was compounded by the fact that AHSL only uses the parameters in the *unified decision hierarchy*. These parameters, unfamiliar to the users, must also be typed in the text-based editor or located in the block-based environment.

Usability testing was not performed with DRL or Drools DSL. Thus, it is impossible to state that AHSL is easier to use than these languages. However, using an example, we found that AHSL required five times fewer lines of code than DRL to express equivalent logic. Additionally, AHSL required half the number of statements to express equivalent logic. However, a more in-depth analysis of the code complexity between both languages is required to draw concrete answers.

Table 7. System test cases.

ID	Name	Description	Expected Result
st01	Message suppressed	Rules and scenario 1 lead to message suppression	*Rule firing log* is generated. It shows that order of rule firing followed *decision sequence*, and that adaptation decisions beyond *suppress-message=true* are not executed due to *gate conditions*. Generated code is *traceable*.
st02	Message Displayed	Rules and scenario 2 lead to message displayed	*Rule firing log* is generated. It shows that order of rule firing followed *decision sequence*, and that adaptation decisions beyond *suppress-message=false* are executed due to *gate conditions*. Generated code is *traceable*.

Table 8. Execution of Drools using *st.drl* in Listing 5 with two scenarios.

Scenario	Initial Parameter Values	Rule Firing Log

Empty road	safety-criticality = low
	time-criticality = high
	driving-relevance = true
	vulnerable-road-user = false
	speed = high
	low-visibility = true
		priority = moderate [AS.1 CS.2]
		trafficRisk = low [AS.2 DS]
		drivingDemand = moderate [AS.3 DS]
		suppressMessage = false [AS.4 DS]
		useVisual = true [AS.5 DS]
		displayHud = true [AS.6 CS.1]

cyclist on the road	safety-criticality = low
	time-criticality = high
	driving-relevance = true
	vulnerable-road-user = true
	speed = high
	low-visibility = true
		trafficRisk = high [AS.2 CS.1]
		priority = moderate [AS.1 CS.2]
		drivingDemand = moderate [AS.3 DS]
		suppressMessage = true [AS.4 CS.1]

7 Conclusions & Future Work

Given the increasing prevalence of ADAS and IVIS systems in the automotive industry, this paper proposes a novel approach to facilitate their development by addressing the following research questions:

How can the HMI be adapted to improve driving comfort and traffic safety? What parameters are needed to make these adaptation decisions?

To address this question, we analyzed existing literature on adaptive HMIs (Section 2.1). These HMIs enhance traffic safety and comfort by adapting ADAS and IVIS messages and warnings through suppression, modality selection, timing adjustments, and signal property modifications. Different parameter sets describing the driver, vehicle, and environment are used to achieve various adaptation goals.

How can a tool provide non-technical users with a simple and compact mechanism to specify executable rules for HMI adaptation?

We developed a DSL named AHSL, offering a concise, natural language-like notation for rule definition. AHSL includes both text-based and block-based editors for program creation, which are subsequently converted into executable DRL rules via a code generator. AHSL code demonstrates a five-fold reduction in complexity compared to DRL code in terms of lines of code, and it expresses decision logic using approximately half the number of statements. User testing confirmed that the clear and concise syntax enhances usability.

The current work offers several extension points. While AHSL currently employs predefined, domain-specific parameters, future iterations could incorporate variable declaration and decision hierarchies for broader applicability. Moreover, enhancing editor functionalities like auto-completion and suggestions can significantly improve user experience and productivity. To address the limitations of the *assign statement*, AHSL could be extended to support multiple parameter assignments within a single block or the reuse of conditions across statements. Additionally, simulating rule execution based on automatically generated input values can aid handling complex scenarios. Additionally, integrating HMI guidelines as constraints within AHSL would promote adherence to best practices. Furthermore, incorporating regional regulations and consumer rating requirements for ADAS warnings would enhance the tool's adaptability. Finally, exploring alternative rule creation mechanisms in Drools, such as those offered by Drools DSL, guided rules, and decision tables, could potentially provide valuable insights for AHSL's evolution.

Acknowledgments

To David A. Manrique Negrin for his insightful suggestions and thorough proofreading, both of which contributed to the improvement of this paper.

A AHSL Grammar

Listing 3. Snippet of AHSL's grammar definition using Rascal [12].

```
start syntax Program
      = program: AssignStmt+ assignstmts;

syntax AssignStmt
      = assignStmt: "assign" Parameter parameter
          ConditionalStmt* conditionalstmts DefaultStmt
          defaultstmt;

syntax ConditionalStmt
      = conditionalStmt: Value value "when" ConditionLine+
          conditionlines;

syntax DefaultStmt
      = defaultStmt: "default" Value value;

syntax ConditionLine
      = conditionline: {Condition "or"}+ conditions;

syntax Condition
      = boolean: Negation? negation Parameter parameter
      | enumeration: Parameter parameter "is" Negation?
          Value value;

syntax Negation
      = not: "not";
```

B Usability Testing

B.1 Usability Heuristics

Jakob Nielsen [14] suggested the following ten general heuristics for measuring the usability of user interfaces:

1. *Visibility of System Status.* Systems should provide immediate feedback on the interaction.
2. *Match between System and Real World.* Systems should use terms and concepts that are familiar to the user.
3. *User Control and Freedom.* Systems should provide clear "emergency exit" mechanisms to leave unwanted states quickly.
4. *Consistency and Standards.* Users should not have to wonder whether different words, visuals, or actions mean the same. Follow conventions.
5. *Error Prevention.* Systems should have a careful design to prevent problems from occurring.
6. *Recognition rather than Recall.* The user should not have to remember information. Instructions should be easily retrievable whenever appropriate.
7. *Flexibility and Efficiency of Use.* The system can cater to both inexperienced and experienced users.
8. *Aesthetic and Minimalist Design.* Dialogues should not contain information which is irrelevant or rarely needed.
9. *Recognize, Diagnose and Recover from Errors.* Error messages should be expressed in plain language and suggest a solution.
10. *Help and Documentation.* Such information should be easy to search, focused on the user's task, list concrete steps, and not be too large.

B.2 Cognitive Dimensions Framework

The Cognitive Dimensions (CD) framework [10] defines the following 13 dimensions:

1. *Abstraction.* Types and availability of abstraction mechanisms.
2. *Hidden dependencies.* Important links between entities are not visible.
3. *Premature commitment.* Constraints on the order of doing things.
4. *Viscosity.* Resistance to change.
5. *Visibility.* Ability to view components easily.
6. *Closeness of mapping.* Closeness of representation to domain.
7. *Consistency.* Similar semantics are expressed in similar syntactic forms.
8. *Diffuseness.* Verbosity of language.
9. *Error-proneness.* Notation invites mistakes.
10. *Hard mental operations.* High demand on cognitive resources.
11. *Progressive evaluation.* Work-to-date can be checked at any time.
12. *Provisionality.* Degree of commitment to actions or marks.
13. *Role-expressiveness.* The purpose of a component is readily inferred.

C Drools Project

The generated Drools project contains three packages:

- Package *com.sample* contains class *ExecuteRules.* Upon execution, it defines initial parameter values, initializes the Drools environment, and fires the rules.
- Package *com.sample.rules* contains the generated rule file.
- Package *com.sample.ahsl* contains the classes that declare the variables used by the rules. An additional class *Constants*) declares the enumerations.

D AHSL: Example

Listing 4. Source *st.ahsl* with six AssignStmt(s).

```
assign priority
  high when
    safety-criticality is high
  moderate when
    time-criticality is high
  default low

assign traffic-risk
  high when
    vulnerable-road-user
  default low

assign driving-demand
  default moderate

assign suppress-message
  true when
    priority is not high
    traffic-risk is high
    driving-demand is not low
  default false

assign use-visual
  false when
    traffic-risk is high
  default true

assign display-hud
  true when
    driving-relevance
  default false
```

E Code Generation Example

Listing 5. Target *st.drl* file.

```
package com.sample.rules

import com.sample.ahsl .*;

rule  "AS.1 CS.1"
        activation -group "AS.1"
        agenda-group "SEQ1"
        salience  99
        when
                $MessageDecision :MessageDecision()
                MessageData( safetyCriticality   == Constants.
                        enumLevel.high)
        then
                $MessageDecision.priority  = Constants.enumLevel.
                        high;
                update($MessageDecision);
                System.out. println ( " priority =high  [" + drools .
                        getRule () .getName() + "]" ) ;
end

rule  "AS.1 CS.2"
        activation -group "AS.1"
        agenda-group "SEQ1"
        salience  98
        when
                $MessageDecision :MessageDecision()
                MessageData( timeCriticality   == Constants.
                        enumLevel.high)
        then
                $MessageDecision.priority  = Constants.enumLevel.
                        moderate;
                update($MessageDecision);
                System.out. println ( " priority =moderate [" + drools .
                        getRule () .getName() + "]" ) ;
end

rule  "AS.1 DS"
        activation -group "AS.1"
        agenda-group "SEQ1"
        salience  -1
        when
                $MessageDecision :MessageDecision()
        then
                $MessageDecision.priority  = Constants.enumLevel.low
                        ;
                update($MessageDecision);
                System.out. println ( " priority =low  [" + drools .
                        getRule () .getName() + "]" ) ;
end

rule  "AS.2 CS.1"
        activation -group "AS.2"
        agenda-group "SEQ1"
        salience  99
        when
                $DVEState :DVEState()
                TrafficRiskData (vulnerableRoadUser ==   true )
        then
                $DVEState. trafficRisk  = Constants.enumLevel.high;
                update($DVEState);
                System.out. println ( "  trafficRisk  =high  [" + drools .
                        getRule () .getName() + "]" ) ;
end

rule  "AS.2 DS"
        activation -group "AS.2"
        agenda-group "SEQ1"
        salience  -1
        when
                $DVEState :DVEState()
        then
                $DVEState. trafficRisk  = Constants.enumLevel.low;
                update($DVEState);
                System.out. println ( "  trafficRisk  =low  [" + drools .
                        getRule () .getName() + "]" ) ;
end

rule  "AS.3 DS"
        activation -group "AS.3"
        agenda-group "SEQ1"
        salience  -1
        when
                $DVEState :DVEState()
        then
                $DVEState.drivingDemand = Constants.enumLevel.
                        moderate;
                update($DVEState);
        System.out. println ( "drivingDemand=moderate [" +
            drools . getRule () . getName() +  "]") ;
end

rule  "AS.4  CS.1"
    activation -group "AS.4"
    agenda-group "SEQ2"
    salience  99
    when
        MessageDecision( priority   !=  null )
        $HMIDecision :HMIDecision()
        MessageDecision( priority   !=  Constants . enumLevel.
            high)
        DVEState( trafficRisk   == Constants . enumLevel.high)
        DVEState(drivingDemand != Constants.enumLevel.low)
    then
        $HMIDecision.suppressMessage = true;
        update($HMIDecision);
        System.out. println ( "suppressMessage=true  [" +
            drools . getRule () . getName() +  "]") ;
end

rule  "AS.4  DS"
    activation -group "AS.4"
    agenda-group "SEQ2"
    salience  -1
    when
        MessageDecision( priority   !=  null )
        $HMIDecision :HMIDecision()
    then
        $HMIDecision.suppressMessage = false;
        update($HMIDecision);
        System.out. println ( "suppressMessage= false  [" +
            drools . getRule () . getName() +  "]") ;
end

rule  "AS.5  CS.1"
    activation -group "AS.5"
    agenda-group "SEQ3"
    salience  99
    when
        HMIDecision(suppressMessage == false)
        $HMIDecision :HMIDecision()
        DVEState( trafficRisk   == Constants . enumLevel.high)
    then
        $HMIDecision.useVisual = false ;
        update($HMIDecision);
        System.out. println ( " useVisual = false  [" +  drools .
            getRule () . getName() +  "]") ;
end

rule  "AS.5  DS"
    activation -group "AS.5"
    agenda-group "SEQ3"
    salience  -1
    when
        HMIDecision(suppressMessage == false)
        $HMIDecision :HMIDecision()
    then
        $HMIDecision.useVisual = true;
        update($HMIDecision);
        System.out. println ( " useVisual = true  [" +  drools .
            getRule () . getName() +  "]") ;
end
```

```
rule  "AS.6  CS.1"
    activation -group "AS.6"
    agenda-group "SEQ4"
    salience  99
    when
        HMIDecision(useVisual == true )
        $HMIDecision :HMIDecision()
        MessageData(drivingRelevance ==   true )
    then
        $HMIDecision.displayHud = true;
        update($HMIDecision);
        System.out. println ( "displayHud=true  [" +  drools .
            getRule () . getName() +  "]") ;
end

rule  "AS.6  DS"
    activation -group "AS.6"
    agenda-group "SEQ4"
    salience  -1
    when
        HMIDecision(useVisual == true )
        $HMIDecision :HMIDecision()
    then
        $HMIDecision.displayHud = false;
        update($HMIDecision);
        System.out. println ( "displayHud= false  [" +  drools .
            getRule () . getName() +  "]") ;
end
```

F Questionnaire

Table 9. Questionnaire for user study

Nr	Question	To Measure
1	User control and freedom	Usability Heuristic 3
2	Consistency	Usability Heuristic 4
3	Prevent Errors	Usability Heuristic 5
4	Recognition rather than recall	Usability Heuristic 6
5	Flexibility and efficiency of use	Usability Heuristic 7
6	Aesthetic and minimalist design	Usability Heuristic 8
7	How easy is it to see or find the various parts of the notation while it is being created or changed? Why?	Cognitive Dimension: Visibility and Juxtaposability
8	What kind of things are more difficult to see or find?	
9	When you need to make changes to previous work, how easy is it to make the change? Why?	Cognitive Dimension: Viscosity
10	Are there particular changes that are more difficult or especially difficult to make? Which ones?	
11	Does the notation a) let you say what you want reasonably briefly, or b) is it long-winded? Why?	Cognitive Dimension: Diffuseness
12	What sorts of things take more space to describe?	
13	What kind of things require the most mental effort with this notation?	Cognitive Dimension: Hard Mental Operations
14	Do some things seem especially complex or difficult to work out in your head (e.g. when combining several things)? What are they?	
15	Do some kinds of mistake seem particularly common or easy to make? Which ones?	Cognitive Dimension: Error Proneness
16	Do you often find yourself making small slips that irritate you or make you feel stupid? What are some examples?	
17	How closely related is the notation to the result that you are describing? Why?	Cognitive Dimension: Closeness of Mapping
18	Which parts seem to be a particularly strange way of doing or describing something?	
19	When reading the notation, is it easy to tell what each part is for in the overall scheme? Why?	Cognitive Dimension: Role Expressiveness
20	Are there some parts that are particularly difficult to interpret? Which ones?	
21	If the structure of the product means some parts are closely related to other parts, and changes to one may affect the other, are those dependencies visible? What kind of dependencies are hidden?	Cognitive Dimension: Hidden Dependencies
22	In what ways can it get worse when you are creating a particularly large description?	
23	When you are working with the notation, can you go about the job in any order you like, or does the system force you to think ahead and make certain decisions first?	Cognitive Dimension: Premature Commitment
24	If so, what decisions do you need to make in advance? What sort of problems can this cause in your work?	
25	Where there are different parts of the notation that mean similar things, is the similarity clear from the way they appear? Please give examples.	Cognitive Dimension: Consistency
26	Are there places where some things ought to be similar, but the notation makes them different? What are they?	
27	After completing this questionnaire, can you think of obvious ways that the design of the system could be improved? What are they?	Other

G Thematic Analysis

Table 10. Codes extracted from the cognitive dimensions and open-ended questionnaire and the themes they became part of.

Nr.	Dimension	Code	Theme	Valence	Editor
1	Visibility	sound like pure English	Natural Language	Positive	Any
2	Visibility	easy to see [value assigned + condition]	Cohesive Assign Statement	Positive	Any
3	Visibility	easy to see [target parameter]	Cohesive Assign Statement	Positive	Any
4	Visibility	reads as natural language	Natural Language	Positive	Any
5	Visibility	somewhat easy with dropdown	Easy Drop-down & Paste	Positive	Blockly
6	Visibility	can be copy pasted	Easy Drop-down & Paste	Positive	Blockly
7	Visibility	navigate left bar for blocks difficult	Left Bar Know Syntax	Negative	Blockly
8	Visibility	navigate left bar for parameters difficult. Suggests search bar	Type/Find Pars & Vals	Negative	Any
9	Visibility	Many parameters hard to find	Type/Find Pars & Vals	Negative	Any
10	Visibility	Can't see parameter type when typing	Type/Find Pars & Vals	Negative	Any
11	Visibility	No auto complete for enum literales. Annoying	Type/Find Pars & Vals	Negative	Any
12	Visibility	Difficult to get used to notation, different from experience	Notation Unfamiliar	Negative	Any
13	Viscosity	More visual to change values (dropdown)	Easy Drop-down & Paste	Positive	Blockly
14	Viscosity	easy to change [value assigned + condition]	Cohesive Assign Statement	Positive	Any
15	Viscosity	easy to change [target parameter]	Cohesive Assign Statement	Positive	Any
16	Viscosity	simplicity of language	Notation Understandable	Positive	Any
17	Viscosity	can be copy pasted	Easy Drop-down & Paste	Positive	Blockly
18	Viscosity	Easy to change with dropdown	Easy Drop-down & Paste	Positive	Blockly
19	Viscosity	navigate left bar for blocks difficult	Type/Find Pars & Vals	Negative	Any
20	Viscosity	navigate left bar for parameters difficult. Suggests search bar	Type/Find Pars & Vals	Negative	Any
21	Viscosity	Easy, syntax intuitive	Notation Understandable	Positive	Any
22	Viscosity	Language clear once you understand it	Notation Understandable	Positive	Any
23	Viscosity	Recalling enum names difficult	Type/Find Pars & Vals	Negative	Any
24	Viscosity	Navigate left bar for parameters difficult	Type/Find Pars & Vals	Negative	Any
25	Viscosity	Navigate left bar for parameters difficult	Type/Find Pars & Vals	Negative	Any
26	Viscosity	Unclear difference of shadow from active block	Shadow Confusing. Add Block Instead	Negative	Blockly
27	Diffuseness	Can express briefly with notation	Notation Concise	Positive	Rascal
28	Diffuseness	Notation minimalistic and unambiguous	Notation Concise	Positive	Rascal
30	Diffuseness	Lots of drag&drop at start	Drag&Drop Repetitive	Negative	Blockly
31	Diffuseness	Notation clear and consistent	Notation Consistent	Positive	Any
33	Diffuseness	Many "or" will create wider code	Long Condition Line "or"	Negative	Any
34	Diffuseness	More space when using "or"s for two values of same parameter.	Long Condition Line "or"	Negative	Any
35	Hard Mental Operations	Remembering that implicit "and" = new line	Implicit "And" Remembering	Negative	Any
36	Hard Mental Operations	Comprehending long and complicated conditions	Large Condition Group Understand	Negative	Any
37	Hard Mental Operations	Getting used to consequence before condition	Action Before Condition Difficult	Negative	Any
39	Error Proneness	Typos in variable names	Type/Find Pars & Vals	Negative	Any
40	Error Proneness	Misplacing block	Drag&Drop Repetitive	Negative	Blockly
41	Error Proneness	Using "is" instead of "="	Notation Unfamiliar	Negative	Any
42	Error Proneness	Forgetting parameter/value names	Type/Find Pars & Vals	Negative	Any
43	Error Proneness	Relies on user knowing syntax (Condition, Assignment, Value)	Left Bar Know Syntax	Negative	Blockly
44	Error Proneness	Error when searching variables	Type/Find Pars & Vals	Negative	Any
45	Error Proneness	Dragging incorrect block	Drag&Drop Repetitive	Negative	Blockly
46	Error Proneness	Lacking: auto-complete	Type/Find Pars & Vals	Negative	Any
47	Error Proneness	lacking: search bar for variables	Type/Find Pars & Vals	Negative	Any
48	Closeness of Mapping	Clear notation	Notation Understandable	Positive	Any
49	Closeness of Mapping	Closely related, conditions and results in neat blocks	Cohesive Assign Statement	Positive	Any
50	Closeness of Mapping	Close because of natural language	Natural Language	Positive	Any
51	Closeness of Mapping	Can become easy documentation	Natural Language	Positive	Rascal
52	Closeness of Mapping	Strange that "and"s are not expressed with words	Implicit "And" Inconsistent	Negative	Any
53	Closeness of Mapping	Finding blocks, suggests search bar	Type/Find Pars & Vals	Negative	Any
54	Closeness of Mapping	Continuously drag and drop	Drag&Drop Repetitive	Negative	Blockly
55	Closeness of Mapping	Some blocks not added automatically, e.g., condition block	Shadow Confusing. Add Block Instead	Negative	Blockly
56	Role Expressiveness	Notation is clear	Notation Understandable	Positive	Any
57	Role Expressiveness	Notation is clear	Notation Understandable	Positive	Any
58	Role Expressiveness	Highlighting elements differently (parameters, value, keyword) would help	Lacks Highlighting	Negative	Rascal
59	Role Expressiveness	Blocks minimalistic	Notation Concise	Positive	Any
60	Role Expressiveness	Clear, natural language	Natural Language	Positive	Any
61	Role Expressiveness	Start with assignment rather than condition, difficult	Action Before Condition Difficult	Negative	Any
62	Hidden Dependencies	Dependencies within block visible	Cohesive Assign Statement	Positive	Any
63	Hidden Dependencies	Comprehending long and complicated conditions	Complicated Conditions Understand	Negative	Any
65	Hidden Dependencies	Many "or" will create wider code	Long Condition Line "or"	Negative	Any
66	Hidden Dependencies	Dependencies between assign pars that depend on each other	Assign Statement Coupling Not Visible	Negative	Any
67	Premature Commitment	Must think to define parameters used as conditions in other blocks to avoid errors	Assign Statement Coupling Not Visible	Negative	Any
68	Premature Commitment	Think first about target parameter. No problems reported	Action Before Condition Difficult	Negative	Any
69	Consistency	Notation consistent	Notation Consistent	Positive	Any
70	Consistency	Notation clear and consistent	Notation Consistent	Positive	Any
71	Consistency	Not confusing	Notation Consistent	Positive	Any

References

[1] Luigi Altamirano Mollo. 2022. *ahsl 0.1.* https://doi.org/10.5281/zenodo.6303824

[2] Angelos Amditis, Luisa Andreone, Katia Pagle, Gustav Markkula, Enrica Deregibus, Maria Romera Rue, Francesco Bellotti, Andreas Engelsberg, Rino Brouwer, Björn Peters, and Alessandro De Gloria. 2010. Towards the automotive HMI of the future: Overview of the AIDE - Integrated project results. *IEEE Transactions on Intelligent Transportation Systems* 11, 3 (2010), 567–578. https://doi.org/10.1109/TITS.2010.2048751

[3] Ankica Bariic, Vasco Amaral, and Miguel Goulao. 2012. Usability evaluation of domain-specific languages. In *2012 Eighth International Conference on the Quality of Information and Communications Technology*. IEEE, 342–347.

[4] Tom Beckmann and Mauricio Verano Merino. 2021. *maveme/skogi: SKogi 0.1.0.* https://doi.org/10.5281/zenodo.5534113

[5] F. Bellotti, A. De Gloria, R. Montanari, N. Dosio, and D. Morreale. 2005. COMUNICAR: Designing a multimedia, context-aware human-machine interface for cars. *Cognition, Technology and Work* 7, 1 (2005), 36–45. https://doi.org/10.1007/s10111-004-0168-9

[6] Rino F.T. Brouwer, Marieka Hoedemaeker, and Mark A Neerincx. 2009. Adaptive interfaces in driving. In *International Conference on Foundations of Augmented Cognition*. Springer, 13–19.

[7] Centro Ricerche FIAT, CRF, University of Siena, Volvo Car, Daimler-Chrysler, Borg Instruments, Metravib, ICCS/NTUA, DIBE, FHG/IAO, BASt, TNO. 2003. *COMUNICAR Project Final Report.* Technical Report. 58 pages. [Online, accessed 27 February 2022].

[8] Frederik Diederichs, Alessia Knauss, Marc Wilbrink, Yannis Lilis, Evangelia Chrysochoou, Anna Anund, Evangelos Bekiaris, Stella Nikolaou, Svitlana Finér, Luca Zanovello, Pantelis Maroudis, Stas Krupenia, Andreas Absér, Nikos Dimokas, Camilla Apoy, Johan Karlsson, Annika Larsson, Emmanouil Zidianakis, Alexander Efa, Harald Widlroither, Mengnuo Dai, Daniel Teichmann, Hamid Sanatnama, Andreas Wendemuth, and Sven Bischoff. 2020. Adaptive transitions for automation in cars, trucks, buses and motorcycles. *IET Intelligent Transport Systems* 14, 8 (2020), 889–899. https://doi.org/10.1049/iet-its.2018.5342

[9] Martin Fowler. 2009. Should I use a Rules Engine? https://martinfowler.com/bliki/RulesEngine.html. [Online, accessed 27 February 2022].

[10] T. R. G. Green and a. Blackwell. 1998. Cognitive Dimensions of Information Artefacts : a tutorial. *Applied Psychology* October (1998), 75. http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:Cognitive+Dimensions+of+Information+Artefacts+:+a+tutorial#0

[11] Michelle E. Kiger and Lara Varpio. 2020. Thematic analysis of qualitative data: AMEE Guide No. 131. , 846–854 pages. https://doi.org/10.1080/0142159X.2020.1755030

[12] Paul Klint, Tijs Van Der Storm, and Jurgen Vinju. 2009. RASCAL: A domain specific language for source code analysis and manipulation. *9th IEEE International Working Conference on Source Code Analysis and Manipulation, SCAM 2009* (2009), 168–177. https://doi.org/10.1109/SCAM.2009.28

[13] Yannis Lilis, Emmanouil Zidianakis, Nikolaos Partarakis, Margherita Antona, and Constantine Stephanidis. 2017. Personalizing HMI elements in ADAS using ontology meta-models and rule based reasoning. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10277 LNCS (2017), 383–401. https://doi.org/10.1007/978-3-319-58706-6{_}31

[14] Jakob Nielsen. 1994. *Usability engineering.* Morgan Kaufmann.

[15] OMG. 2009. Production Rule Representation (PRR). http://www.omg.org/spec/PRR/1.0/. , 74 pages. [Online, accessed 27 February 2022].

[16] Ludwig Ostermayer, Geng Sun, and Dietmar Seipel. 2013. Simplifying the Development of Rules Using Domain Specific Languages in DROOLS.

[17] Erik Pasternak, Rachel Fenichel, and Andrew N Marshall. 2017. Tips for creating a block language with blockly. In *2017 IEEE Blocks and Beyond Workshop (B&B)*. IEEE, 21–24.

[18] Mauricio Salatino, Mariano De Maio, and Esteban Aliverti. 2016. *Mastering JBoss Drools 6.* Packt Publishing Ltd.

[19] JBoss Drools Team. 2022. Drools Documentation. https://docs.drools.org/7.10.0.Final/drools-docs/html_single/. https://docs.drools.org/7.10.0.Final/drools-docs/html_single/ [Online, accessed 28 July 2022].

[20] Mauricio Verano Merino, Tom Beckmann, Tijs van der Storm, Robert Hirschfeld, and Jurgen Vinju. 2021. Getting Grammars into Shape for Block-Based Editors. https://doi.org/10.1145/3486608.3486908 SIGPLAN International Conference on Software Language Engineering (SLE'21), SLE'21 ; Conference date: 17-10-2021 Through 18-11-2021.

[21] Mauricio Verano Merino and Tijs Van Der Storm. 2020. Block-based syntax from context-free grammars. *SLE 2020 - Proceedings of the 13th ACM SIGPLAN International Conference on Software Language Engineering, Co-located with SPLASH 2020* (2020), 283–295. https://doi.org/10.1145/3426425.3426948

[22] Mauricio Verano Merino and Tijs van der Storn. 2020. cwi-swat/kogi: Kogi 0.1.0. https://doi.org/10.5281/zenodo.4033220.

[23] GJ Witt. 2003. SAfety VEhicle (s) using adaptive interface Technology (SAVE-IT) Program. *US Department of Transportation, The National Transportation Systems Center* (2003).

[24] Yali Wu, Frank Hernandez, Francisco Ortega, Peter J. Clarke, and Robert France. 2010. Measuring the effort for creating and using domain-specific models. *Proceedings of the 10th Workshop on Domain-Specific Modeling, DSM'10* (2010). https://doi.org/10.1145/2060329.2060360

[25] Xulin Zhao, Ying Zou, Jen Hawkins, and Bhadri Madapusi. 2007. A business-process-driven approach for generating e-commerce user interfaces. In *International Conference on Model Driven Engineering Languages and Systems*. Springer, 256–270.

Received 2024-07-15; accepted 2024-08-11