DOI: 10.15439/2023F5469
ISSN 2300-5963 ACSIS, Vol. 35

A Reusability-Oriented Use-Case Model Specification Language

Bogumiła Hnatkowska, Piotr Zabawa
0000-0003-1706-0205
0000-0002-5078-9869
Wroclaw University of Science and Technology
Wyb. Wyspianskiego 27, 50-370 Wroclaw, Poland
Email: {bogumila.hnatkowska, piotr.zabawa}@pwr.edu.pl

***Abstract*—Use-case models play an essential role in software development processes. They are used to specify functional requirements, estimate software development project efforts, and plan iterations. The use-case model is subject to change as requirements are modified, or the model is refactored. Therefore, it is essential that the use-case model is not redundant and its parts are reusable. Existing approaches for use-case model specification support reusability in a limited way. This paper fills the gap. It introduces a new approach to conveniently yet semi-formally specifying the entire use-case model. The paper presents the Use-Case Flow Language metamodel, consisting of its abstract syntax and a description of the semantics of the metamodel elements. A concrete textual syntax of the language is also provided and informally described. An example of a use-case model specified in the proposed notation is presented in the paper.**

I. Introduction

THE SOFTWARE requirements specification (SRS) is one of the most important artifacts documenting the qualities of a software product. It is always produced regardless of the development methodology used. The SRS can take different forms, including use-case models, product backlogs with user stories, or documents written in free natural language. In the case of a use-case model, the SRS consists of a use-case diagram and the associated use-case specification documents, typically documented using structured texts, tables, or graphical notations (e.g., activity diagrams).

Textual specifications are the most widely used because they are easy to understand and quick to define, even for non-technical people. Still, on the other hand, they can be misinterpreted or incomplete [1]. Therefore, many researchers (e.g., [2], [3], [4]) try to define templates, a set of patterns or rules that help to keep use-case specifications complete, coherent, and consistent.

The important aspect of use-case specification, not fully covered by the existing research, is the specification reusability. The same steps, step sequences, flows, or subflows can be applied in many places when the change in one place will influence all their instances. There are already defined some reusability mechanisms in the UML, like «include» and «extend» dependencies and generalization. However, they are defined at the use-case level. To have an advantage of these mechanisms for use-case fragments, one should introduce new use-cases just for the reusability, which would increase the use-case number significantly, making the use-case diagram and the whole use-case model more complicated.

The paper aims to define a general-purpose language for writing textual use-case model specifications, emphasizing reusability. The proposed notation is consistent with existing good practices, and the result of their application, i.e., the textual use-case specification should have the necessary features to allow its further processing, e.g.:

- Checking use-case models' completeness and correctness.
- Bi transformation into diagrammatic notations, e.g., activity diagrams or flow visualization.

The motivation to cover the use-case model by a standardization process was and still is very strong. The reasons for standardization efforts are as follows:

- The use-case model is used for the specification of functional requirements.
- Use-cases are the source information not only for the implementation of a software product but also for the verification of the product by functional tests; the functional tests can be implemented directly from the use-cases in parallel with the implementation.
- The use-case model is used to estimate the development efforts (use-case points method [5]).
- Use-cases play a crucial role in iterative software development projects as the iteration plans are organized for a set of use-cases or similar constructs.

The contribution of this paper is a notation specification of a use-case model flow language (Use-Case Flows Language, UCFL) used for scenario definition as a supplementary part of a use-case diagram. The specification includes the language metamodel (abstract syntax) – see section III, and concrete textual syntax (T-UCFL) – see section IV. The metamodel takes the form of a UML class diagram, while the concrete syntax is given in the form of context-free grammar. The notation is presented with several examples. It is characterized by a minimal set of keywords used in use-case flow steps. Examples of the T-UCFL usage are contained in section V. And, finally, the content of the research presented in the paper is summarized in section VI.

II. Related Work

A metamodel is a typical form of abstract syntax representation, also used for use-case models ([6], [7]). Such a metamodel can have many different representations, both graphical and textual. The authors decided to propose their own metamodel for the use-case specification formalism in order to overcome the limitations of existing, e.g., the lack of iterations or interruptions.

The concrete syntax of UCFL called T-UCFL takes the form of a free context grammar – such a solution was used in [8] for a similar purpose. The grammar has been developed with best practices in mind. As this is a textual specification, the authors draw inspiration from many existing books [2]-[4] and papers [9]-[10]. These references suggest, among others, different templates of use case scenarios, keyword sets, and ways of identifying steps. To the best of the authors' knowledge, none of them pay attention to the reusability of the step, step ranges, or global flows. The existing reusability mechanisms are defined as reusable templates [11] or patterns [6], [7].

Use-case specifications with globally visible flows are collected in a use-case model. A similar idea is given in [12], where the authors suggest using "several mechanisms to factor out common usage like error handling from sets of use-cases", but the idea is not formalized.

Common elements proposed in this paper include:

- use case name,
- documentation – can be a substitute for a goal, brief description, primary and secondary actors,
- pre- and post-conditions – similar to [13], post-conditions can be divided into subgroups depending on the scenario and return a specific state [9]; such a construct can be used to model minimal guarantees and success guarantees,
- subflows – as potential elements of reuse,
- the main flow of events – sometimes called scenario ([4]) or basic/normal course of events ([8], [12]),
- flows – called alternative flows ([4]), alternative courses (e.g. [2]), or extensions (e.g. [3]).

Use-case flow is typically expressed in terms of actions. Sometimes these actions have no implicit structure, e.g. [12], [4], when the scenario is simply a sequence of sentences. In the proposed approach, the actions are classified and uniquely identified by step identifiers, which enables their reusability. The step identification resembles the one proposed in [2], [11] and implemented in some tools, e.g., CaseCompleted [14] or Enterprise Architect [15].

The use-case semantics, especially the control flow, must be clearly defined. This can be achieved by using specific keywords. The keywords used in the literature to represent the control flow are as follows:

- GOTO step ([11]) or USE-CASE CONTINUES AT step ([14]).
- IF-THEN-ELSE-ELSEIF-ENDIF, MEANWHILE, VALIDATES THAT, DO-UNTIL, ABORT, RESUME step, INCLUDE use-case, EXTENDED BY use-case ([13]).
- IF, VALIDATES, RESUME FLOW, goto, and resume statements are also defined indirectly in separate columns with appropriate names (alternative FlowId, resume FlowId) as identifiers to steps ([16]).
- USE-CASE CONTINUES AT, RETIRED n TIMES, ENDS IN state ([9]).
- COND, INVOKE, REJOIN, FINAL: state ([7]).

Most of them were adapted in the proposed language, e.g., goto, validates that, includes, extends, and final.

III. UCFL Metamodel

This section presents the abstract syntax and semantics of the Use-Case Flow Language specification. The UCFL abstract syntax, in the form of the UML class diagram, is shown in Fig. 1. As the notation focuses on the specification of use-case behavior, the UCFL abstract syntax does not contain either actors or the relationships between them.

A. UCFL Containers

Container is a named element containing flows or their refinements – subflows. We have two types of containers: use-case model and use-case.

1) Use-Case Model: Use-case model is a container of use-cases. It can define publicly visible flows and subflows. Optionally, the use-case model can specify so-called use-case model interruptible regions or flow-interruptible regions (see section III-G) and be documented by a string.

2) Use-Case: Use-case is a basic modeling element that represents interactions between the system and its actors via flows and subflows. It may have optional documentation describing the use-case goal. The use-case may also specify use-case interruptible regions (see section III-G).

Pre and postconditions: A use-case may require some preconditions to be met in order to enable the use-case behavior. These preconditions (if any) are sentences in natural language. The use-case behavior can change the state of the system. The state changes are represented by postconditions. Each postcondition is a sentence in natural language with a state name, e.g., success, partial success, failure, or other, defined by a modeler.

Generalization: Use-cases can be related to each other with generalization relationships. A use-case can be the parent of many use-cases (children). Only leaves of the inheritance tree can have flows defined. A justification for this decision is given later in this section.

B. UCFL Container Elements

1) Flow: Flow is a key element used to structure the use-case behavior. It is a sequence of steps referring to actions performed either by an actor or by the system. A step has a sequence number and a step identifier constructed from the flow identifier.

From the perspective of a graphical language representation, a flow is a path (possibly looped) in the graph without any branches. Flows can be assembled into a graph using specific actions, e.g., conditional. The first action in the flow can

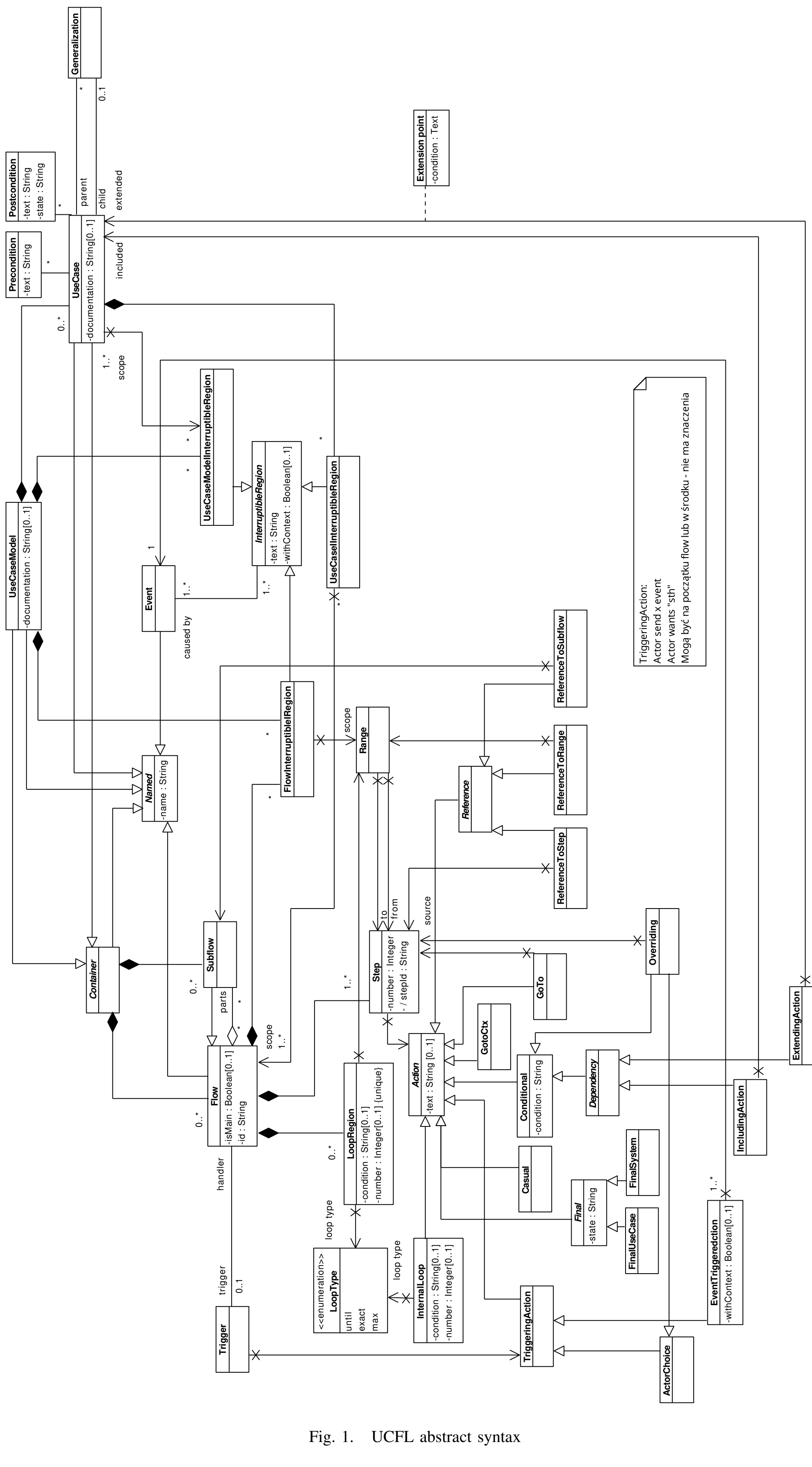

Fig. 1. UCFL abstract syntax

connect it to another flow (as a branch of another flow) - see Fig. 2.

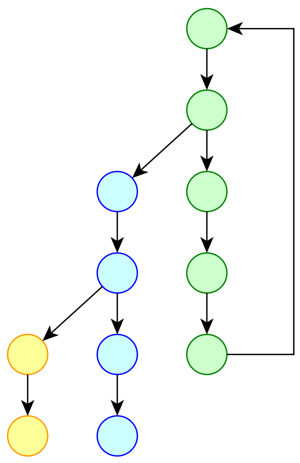

Fig. 2. Flow visualization – different flows are represented by different colors

The flow declaration introduces a flow identifier and a name (both of which must be unique within the context of the flow owner) and, optionally, a trigger. A trigger specifies an action (called a triggering action) that enables the flow. If the flow has a trigger, the flow is called a handler. If the owner of the flow is a use-case then the flow can be marked as a main flow (a use-case must define exactly one main flow; other flows are alternatives).

A flow can additionally define flow interruptible regions (see section III-G) and loop regions (see section III-E).

A flow can be constructed from subflows.

2) Subflow: Subflow is a specialized flow with the restriction that its steps must refer to actions that form a sequence that is casual, finals, and internal loops actions (see section III-F). The subflow is a primary reusable element. It can be shared by several flows; however, a subflow cannot contain interruptible regions or loop regions.

C. Range

Range is a sequence of steps (from–to) included in one flow. Ranges define the scope of flow interruptible regions (see section III-G) or loop regions (see section III-E).

D. Loop Type

Loop type is an enumeration of literals defining different types of loops: `until` (do something until condition), `exact` (do something the exact number of times), `max` (do something the maximum number of times). The type is specified when defining a loop region or an internal loop action.

E. Loop Region

Loop region is the specification of a range that can be repeated in the manner defined by a Loop type. If the Loop type is set to `until`, the condition for the loop region must be defined. Otherwise, the number attribute must be set.

F. Actions

Each step of the flow must refer to one action describing the actor-system interaction in an informal way (*text* attribute in *Action* metaclass).

1) Triggering Actions: A trigger specifies an action (so-called triggering action) that enables the flow. It is the only action that does not need to be referenced by a flow step because it is assumed that it will be performed by an actor to start the flow. There are two types of triggering actions: actor choice action and event-triggered action.

Actor choice action: A flow can be started at the request of the actor, represented by the Actor's choice triggering action.

Event-triggered action: A flow can be started by an actor sending an event to the system, which is modeled by an event-triggered action.

The event-triggered action must refer to an event and optionally can contain a request to store the context before the event is handled (attribute *withCtx*). The event is understood as something that happens at a specific time that requires the system reaction. The event has a name that serves as an event identifier. The context defines the name of the running flow or subflow within the region scope (if any) and its running step, which allows the behavior to be resumed later.

2) Casual action: Casual action is the most general. It is used to model anything the actor or system must do, that cannot be expressed by other actions.

3) Finals: A modeler can define a final action to express that the system has completed its operation (Final system action) or that a use-case has completed its operation in a particular state (Final use-case action). Such an action should be the last one in the flow (or subflow in the case of the final system action).

4) Conditional: Conditional action represents a decision made by the system under specific conditions. Such an action may check whether another use-case has ended in a particular state. It is usually the first action of the alternative flows.

5) Internal Loop: Internal loop represents a case where a particular action is to be repeated in the manner defined by the loop type (see section III-D for details).

6) GoTos:

GoTo: Goto action is used to define unconditional loops. You can jump to a particular step in the same flow or any flow in the same use-case provided that the referenced step exists.

GoTo Ctx: The special version of goto action - `Goto ctx` - allows you to return to the previously saved context (the interrupted action is executed again).

7) Overriding: Overriding is a specific action used as a branching mechanism in the flow definition. This action points to the step in the base flow that is being overridden. The action in the source step must be of the same type as the parent of the overriding action (Actor choice or Conditional).

8) References: References represent the reusable elements of the T-UCFL. Depending on the scope of reusability, three types of reference are distinguished: reference to step, reference to subflow, and reference to range.

Reference to step: Reference to step is the simplest reference action, where the scope of reusability is limited to a single action defined in the step to which the reference action refers. You can imagine that the reused action is copied in place of the reference to the step action.

Reference to subflow: Reference to subflow is the reference action in which the scope of reusability is a particular subflow. When the subflow activity is finished, the control flow is passed back to the original flow.

Reference to range: Reference to the range is the reference action in which the scope of reusability is limited to a specific range.

9) Dependencies: Including: A use-case can include the behavior of another use-case. The semantics of this action is similar to the «includes» relationship in the UML [17] where the including use-case is the owner of the flow with the including action, and the included use-case is the one indicated by the including action.

Extending: The flow of a use-case can contain an extending action. The semantics of this action is like the «extends» relationship in the UML [17] where the extended use-case is the flow owner with the extending action, and the extending use-case is that indicated by the extending action. The extension point describes a condition that must be satisfied for the extension to take place.

G. Interruptible Regions

The UCFL allows the definition of interruption (exception) handling mechanisms using so-called interruptible regions. Such a region points to its scope. The scope of the region can be either a set of use-cases (use-case model interruptible region), a set of flows defined within a use-case model, or a use-case (use-case interruptible region), a range (flow interruptible region). The scope can be interrupted by any event, that caused the interruption.

1) Use-case model Interruptible Region: Use-case model Interruptible Region enables specification of the interruption mechanisms at the use-case model. The interruption scope can refer to any flow or a use-case defined in this container.

2) Use-case Interruptible Region: Use-case Interruptible Region enables specification of the interruption mechanisms at the use-case level. The interruption scope can refer to any flow defined in this container.

3) Flow Interruptible Region: Flow Interruptible Region enables specification of the interruption mechanisms at the flow level. The interruption scope can refer to a range (step from, step to) defined in the flow context.

H. Use-case generalization relationship

Use cases are classifiers and can inherit one from another. An example of such inheritance is shown in Fig. 3. Assuming that the use-case specification is given in natural language, the question arises of how the use-case generalization influences their specification, which can "include possible variations of its basic behavior, including exceptional behavior and error handling" [17].

Generally, a behavior is a specification of events that may occur during the use-case lifetime. The specification must contain at least one event – the event of its invocation [17]. The behavior is invoked when an instance of the owning classifier (i.e., use-case) is created.

In the case of use-case inheritance, a child's specification of events (including the triggering one) is inherited from the parent use-case, which makes the whole specification ambiguous. Therefore, to avoid possible problems and misinterpretations, we assume that any parent use-case must serve only as a root of a use-case hierarchy. Use-case triggers for the hierarchy leaves should determine which child to run.

IV. T-UCFL Informal Description

This section demonstrates the use of the T-UCFL concrete syntax ([18]) with several examples. The language grammar has been designed to keep the language flexible and concise. However, as the specification is intended to be processed by computers, the grammar may impose some constraints on the use of the language, such as the need to enclose elements in quotes or the use of certain keywords.

A. T-UCFL Containers

The container – as an abstract class – has no textual representation.

1) Use-case Model: A use-case model is a container and a namespace for all other elements. Its declaration defines the model's name (e.g., Buying) and optional documentation. Its definition contains shareable elements with global visibility (flows and subflows), an optional declaration of interruptible regions, and a list of use-cases. The concrete syntax assumes that the documentation is textual; however, for readability purposes, the authors decided to use a graphical version in the example presented below (see Fig. 3).

```
Use-Case Model: Buying
Documentation:
```

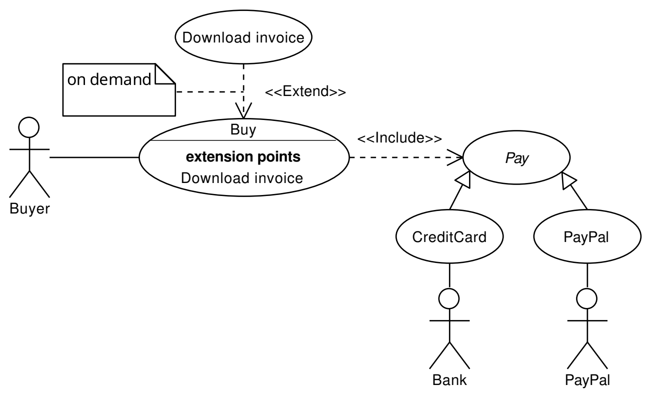

Fig. 3. Buying use-case model documentation in the form of a use-case diagram

2) Use-case: Use-case specification consists of a use-case declaration followed by the use-case definition. The use-case declaration defines a unique use-case name within the use-case model (e.g., `CreditCard`) and, optionally, use-case documentation.

```
Use-Case: CreditCard
Documentation: "Use-case enables payment
  with a credit card."
```

3) Pre and postconditions: A use-case declaration can also contain pre- or post-conditions placed after documentation (if any). The precondition section has one or more statements expressing conditions, for example

```
Preconditions:
- "Actor is logged in the system."
```

Quotation marks are required by formal grammar and can be skipped if the use-case specification is not going to be automatically translated.

Each post-condition section, if any, should define a name of a final system state name (e.g., success, partial-success, error) followed by one or more conditional statements, e.g.,

```
Postcondition(success):
- "An order is stored by the system."
```

4) Generalization: If a use-case has a parent, its name follows the child use-case name and "`-->`" symbol, e.g., `Pay` is the parent use-case for the `CreditCard` use-case:

```
Use-Case: CreditCard --> Pay
```

B. T-UCFL Container Elements

1) Flow: Flow is a named element with an additional string identifier. A flow can be defined in the context of a use-case model, typically as a handler for some event or in the context of a particular use-case. A use-case should have exactly one flow with the reserved name: `Main flow`, and any number of alternative flows with unique identifiers.

Each flow defines a sequence of numbered steps. The step number is constructed with a sequence number preceded by the flow identifier (skipped for the main flow), e.g.:

```
Use-Case: Buy
Trigger: ...
Main flow:
  1. ...
  2. ...
Flow B: The_order_data_invalid
  B1. ...
Flow C: Unsuccessful_payment
  C1. ...
```

The example shown above presents the `Buy` use-case with the main flow and two alternative flows `B` and `C` (`B` is the identifier, `The_order_data_invalid` – is the flow name). The main flow of the use-case has a triggering action defined.

2) Subflow: A subflow is an element of reuse. It can be visible globally (subflows defined at the use-case model level) or locally (subflows defined at the use-case level). They serve to split long flow definitions into manageable fragments. Only casual, final, and internal loop actions are allowed in the subflow definition.

A subflow example is given below:

```
Subflow P: Car_info
  P1. ...
  P2. ...
```

C. Range

Range defines a subsequence in a flow, identified by two steps identifiers, e.g., `2.-3.` consists of 2 steps (2 and 3 in the `Main flow`), `A.5.-A.7.` consists of 3 steps in the flow `A`.

D. Loop Type

Loop type is a keyword (one of `until`, `exact`, `max`) used together with a loop region or internal loop action to specify the loop type – see section V for examples.

E. Loop Region

The loop region works in a similar way to an interruptible flow region. It specifies a range of steps to be repeated as specified by the associated loop type. The loop region is placed after all the steps of the flow, e.g.,

```
Main flow:
  1. ...
  2. ...
Steps 1-2 can be repeated exactly 3 times
Steps 1-2 can be repeated
  until "condition."
```

F. Actions

1) Triggering Actions: A triggering action is typically used to specify how an actor starts a particular flow. In this case, it is specified before the flow, after the `Trigger` keyword. Examples of triggering actions include Actor choice action or Event-triggered action. However, the triggering actions can also be referenced by flow steps.

Actor choice action: A flow can be started by an actor (their decision). Such action must start with `Actor wants` and be followed with "decision" written in quotation marks, e.g.,

```
Use-Case: CreditCard --> Pay
Trigger: Actor wants "to pay with a credit
card"
Main flow: ...
```

2) Event-triggered action: A flow can also be started by an event sent asynchronously by an actor. Such an action must start with `Actor sends` and be followed with the event name and one of `event` or `event with ctx`, e.g.,

```
Actor sends cancelling_service event
Actor sends cancelling_service event
```

```
with ctx
```

The latter action contains the request to store the context before the event is handled.

3) Casual action: Casual action is the most general. It is a free text without keywords present in other types of actions like `verifies`, `includes`, `ends with` or `goto` (e.g., `"System asks about the order data"`), representing something that the actor or the system must do. The grammar requires this action to be enclosed in quotation marks.

4) Finals: The modeler can define a final action expressing that the system finishes operation (`The system ends`) or a use-case finishes in a specific state (e.g., failure) with the phrase: `The use-case ends with failure`. Such an action should be the last one in the flow. The first one means that the system stops running.

5) Conditional: The conditional action represents a decision made by the system. It must contain the phrase `System verifies` or `System verifies that`, followed by a phrase containing a condition, e.g., `System verifies that "the order data are valid"`. Such an action may check whether another use-case ended in a specific state, e.g. (`System verifies that Pay use-case ended with failure`).

6) Internal loop: One can define that a given action should be repeated a specific number of times specifying its loop type, e.g.,

- `"Actor selects products" max 3 times.`
- `"Actor selects products"`
  `until "he is satisfied".`

7) GoTos: GoTo: GoTo action is used to define unconditional loops. We can jump to a particular step in any flow defined in the specific context (a use-case model or a use-case) provided that the referenced step exists, e.g., `Goto 2.` (a jump to the 2nd step in the `Main flow`), `Goto A3.` (a jump to the 3rd step of flow `A`).

GoTo Ctx: GoTo ctx is a special version of the goto action that passes the control flow to the previously saved context (if any). If no context is stored, the semantic is undefined. The interrupted action defined by the context is executed again.

8) Overriding: Overriding actions are used to link flows in a graph. They point to the action in another flow and should be of the same type as the overridden action. Typically, they start alternative flows in a use-case. An example of an overriding action when a decision is made by the system might look like this:

```
B1.3. System verifies that "the order
      data are invalid"
```

The 3rd step in the main flow will be overridden in the `B` flow with the action given above.

9) References: A reference is a basic reusability mechanism. One can reuse another step, step range, or subflow behavior. Examples of such actions are given below:

- `A1.2.` (a step reference; in the 1st step of flow `A`, the 2nd step of the main flow is reused)
- `A2.B3.` (a step reference; in the 2nd step of flow `A`, the 3rd step of flow `B` is reused)
- `B3.A1.-A2.` (a range reference; in the 3rd step of the flow `B`, the range of two steps 1-2 from the flow `A` is reused)

Technically, the reference to a singular step or step range can be thought of as a shortcut for a preprocessing mechanism that copies the referenced elements to the places where they are used and renumbers the steps respectively. Let us assume that flow A contains the steps:

```
A3. Action 1
A4. Action 2
```

and that flow B contains the steps:

```
B1. Action 3
B2. A3.-A4.
B3. Action 4
```

The result of such preprocessing can look like this:

```
B1. Action 3
B2.a. Action 1
B2.b. Action 2
B3. Action 4
```

Subflows must be directly referenced (keyword `subflow` followed by the subflow name) in the appropriate actions, e.g.

```
A2.subflow Car_Info
```

10) Dependencies: Including: One use-case can include or extend another use-case behavior. This is modeled with dependency actions: including or extending. An example of the including action is given below:

```
System includes Pay use-case.
```

When the included use-case reaches the final action, the control returns to the including use-case.

Extending: Two use-cases can also be linked with an extension relationship. The flow of the extended-use case should contain the extension point definition, e.g.,

```
Extension point: "Actor requires the
  invoice downloading."
```

The flow is extended with `Download_invoice` use-case

The extension point specifies a condition under which the flow is extended with another behavior (here: "Actor requires the invoice downloading"). The control returns to the extended use-case when the extending use-case reaches the use-case final action.

G. Interruptible Regions

An interruptible region defines a scope for which the normal operation of the system can be interrupted by a specific event (its name is given) coming from an actor.

1) Use-case model Interruptible Region: Use-case model interruptible region is the one with the widest scope. If it is present, it is placed at the beginning of the use-case model definition, e.g., where any use case can be interrupted by the `close_system` event.

```
Use-Case Model: Document_Editor
Any use-case can be interrupted
  by close_system event
```

2) Use-case Interruptible Region: The scope of a use-case interruptible region is limited to a specific use-case. If it is present, it is placed at the beginning of the use-case definition, e.g.,

```
Use-Case Model: Buy
Any flow can be interrupted by
  close_system event
```

3) Flow Interruptible Region: The scope of a flow interruptible region is limited to a range within a specific flow. If it is present, it is placed after all flow actions, e.g.,

```
1. ...
10. The use-case ends with success
Steps 1.-3. can be interrupted by
  cancelling_service event with ctx
```

The flow interruptible region narrows the scope of the event handling mechanism, e.g., the interruption will be only handled within between steps 1-3 (inclusively).

V. Example Specification

This section contains a small example of a use-case model from Fig. 3, which presents most of the constructs introduced informally in section IV.

The first part of the T-UCFL model specification is related just to the model:

```
Use-Case Model: Buying
Trigger: Actor sends
  cancelling_service event
Flow A: Cancelling_service_event_handler
  A1. "System asks
      for cancellation confirmation"
  A2. Actor wants
      "to cancel the operation"
  A3. The use-case ends with failure

Flow B: Cancellation_denied
  B1.A2. Actor wants "to deny cancellation"
  B2. Goto ctx
```

This part of the model specification is composed of the use-case model called `Buying`; global flow `A` named `Cancelling_service_event_handler`, which is shared among all use-cases and can be triggered by the `cancelinig_service` event generated by an actor; the global flow `B` named `Cancellation_denied` which is a branch of the `A` flow (see a reference step `B1.A2.`). The `Goto ctx` action (if performed) will pass the control flow to the context.

The remaining parts of the T-UCFL specification contain subsequent use-case specifications.

```
Use-Case: Buy
Postcondition (success):
- "An order is stored by the system"
- "An invoice is generated, assigned
  to the order, and stored by the system"

Trigger: Actor wants "to buy an item"
Main flow:
  1. ...
  2. "System asks about order data
     (including payment method)"
  3. "Actor delivers the order data"
  4. System verifies that "the order
     data are valid"
  5. System includes Pay use-case
  6. System verifies that "the Pay
     use-case ended with success"
  7. "System stores an order,
     generates an invoice,
     and sends it by e-mail"
  8. "System informs about
     order completion and enables
     an option to download the invoice"
  9. Extension point: "Actor requires
     invoice downloading"
The flow is extended with
  Download_invoice use-case
  10. The use-case ends with success

Steps 1.-3. can be interrupted
  by cancelling_service event with ctx
```

The main flow contains several conditional actions (e.g., 4, 6). The 5th step contains an including action (`Pay` use-case is included). The 9th step contains an extending action with the condition defined. Finally, there is a flow interruptible region consisting of step range `1.-3.`.

Then two alternative flows (`B` and `C`) are defined:

```
Flow B: The_order_data_invalid
  B1.4. System verifies that
        "the order data are invalid"
  B2. "System informs about invalid data"
  B3. Goto 2.

Flow C: Unsuccessful_payment
  C1.6. System verifies that
        "the payment ended with failure"
  C2. "System informs about the lack
      of payment"
  C3. Goto 2.
```

In both cases, the first step refers to the step with conditional action in the main flow and contains the condition, which complements the condition from the main flow.

The specification continues with the `Download_invoice` use-case specification:

```
Use-Case: Downolad_invoice
Postcondition (success):
- "An invoice is downloaded to
   the Buyer's computer"

Main flow:
  1. "System presents the invoice details
     and asks for confirmation
     of the invoice download"
  2. Actor wants "to download the invoice"
  3. "System sends the last buyer invoice
     to the buyer's computer"
  4. The use-case ends with success
Steps 1.-3. can be interrupted
  by cancelling_service event with ctx

Flow B : Downloading_not_confirmed
  B1.2. Actor wants "to skip downloading"
  B2. The use-case ends
      with partial success
```

The use-case has only one alternative flow `B`, which – in contrast to the `Buy` use-case, is started by the actor's choice action.

There is also a group of three interrelated use-cases in the `Buying` use-case model. The first is the `Pay` use-case, which is abstract and has no flow. It has the following form:

```
Use-Case: Pay
Documentation: "Abstract use-case.
  A root hierarchy for different
  payment methods"
Postcondition (succes):
- "payment succesfull"
Postcondition (failure):
- "payment unsuccesfull"
```

The specification also contains two concrete use-cases (`CreditCard`, `PayPal`) that inherit from the `Pay` use-case. Because of limited space only the first is presented:

```
Use-Case: CreditCard --> Payment
Documentation: "Use-case enables payment
  with a credit card"
Trigger: Actor wants "to pay with a
  credit card"
Main flow:
  1. "System asks for credit card details"
  2. "Actor delivers credit card details"
  3. "System sends a request to a bank
     for payment and waits
     for bank response"
  4. System verifies that "the payment
     was successful"
  5. The use-case ends with success
Steps 1.-3. can be interrupted
  by cancelling_service event

Flow B: Payment_unsuccessfull
  B1.4. System verifies that
        "the payment was unsuccessful"
  B2. The use-case ends with failure
```

Other examples, together with the language abstract and concrete syntax, are available at [18].

VI. Summary

The concept of a new use-case model specification language (UCFL) consisting of the metamodel, and a textual concrete syntax (T-UCFL) was introduced in the paper. The main purpose of the language is the specification of use-case behaviors. It has commercial origins, as the need for the reusability-oriented approach to use case modeling was recognized during the authors' commercial activities.

The T-UCFL syntax was stabilized on plenty of advanced experiments focused on modeling non-trivial behaviors of some invented software systems. The UCFL metamodel was inferred from these experiments.

Concepts introduced in the paper are designed to extensively support the reusability and avoid redundancy in use-case flows for the whole use-case model. The reusability is achieved at different granulation levels, from a singular step, steps' range to a subflow. Flow initial fragments are reused by definition as they are shared with alternative flows. Inclusion, extension, and generalization between use-cases are also supported.

The language helps to introduce changes into the use-case model. A change made in one place "is visible" in many places referring to the changed element.

The UCFL introduced in the paper seems to be very promising and could be further developed. It is internally consistent, concise, and semi-formal - the specification mimics those written in natural language.

It is worth noting that the paper only introduces a textual concrete syntax. However, other syntaxes may be introduced, especially graphical ones.

In the future, the authors intend to extend the proposed notation with tool support. They also work on graphical concrete syntax and bidirectional transformations between concrete syntaxes. Of course, the usability of the language needs to be validated by external users, first in academia and then in industry.

References

[1] S. Liu, J. Sun, Y. Liu, Y. Zhang, B. Wadhwa, J. Dong, and X. Wang, "Automatic early defects detection use case documents," in *Proc. 29th ACM/IEEE international conference on Automated software engineering*, 2014, pp. 785–790.

[2] S. Adolph, P. Bramble, and A. Pols, *Patterns for Effective UseCases*. Addison-Wesley Professional, 2003.

[3] A. Cockburn, *Writing Effective Use-Cases*. Addison-Wesley, 2000.

[4] G. Overgaard and G. Palmkvist, *Use-cases: Patterns and Blueprints*. Addison-Wesley, 2005.

[5] S. Diev, “Use cases modelling and software estimation: applying use case points,” *ACM SIGSOFT Software Engineering Notes*, vol. 31, no. 6, pp. 1–4, 2006.

[6] M. Śmiałek, J. Bojarski, W. Nowakowski, A. Ambroziewicz, and T. Straszak, “Complementary use case scenario representations based on domain vocabularies,” in *Proc. MODELS’07*. Berlin, Heidelberg: Springer-Verlag, 2007, pp. 544–558.

[7] M. Śmiałek, A. Ambroziewicz, and P. R, “Pattern library for use-case-based application logic reuse,” in *Proc. Databases and Information Systems. Communications in Computer and Information Science*, vol. 838. Cham: Springer, 2018, pp. 90–105.

[8] S. Iqbal, I. Al-Azzoni, A. G, and K. HU, “Extending uml use case diagrams to represent non-interactive functional requirements,” *e-Informatica Software Engineering Journal*, vol. 14, no. 1, pp. 97–115, 2020.

[9] S. Mustafiz, J. Kienzle, and H. Vangheluwe, “Model transformation of dependability-focused requirements models,” in *Proc. ICSE Workshop on Modeling in Software Engineering*, 2009, pp. 50–55.

[10] I. Santos, R. Andrade, and P. Santos Neto, “Templates for textual use cases of software product lines: results from a systematic mapping study and a controlled experiment,” *Journal of Software Engineering Research and Development*, vol. 3:5, 2015.

[11] M. Ochodek, K. Koronowski, A. Matysiak, P. Miklosik, and S. Kopczynska, “Sketching use-case scenarios based on use-case goals and patterns,” *Software Engineering: Challenges and Solutions. Advances in Intelligent Systems and Computing*, vol. 504, pp. 17–30, 2017.

[12] D. Rosenberg and S. Kendall, *Applying Use Case Driven Object Modeling with UML: an Annotated e-Commerce Example*, 1st ed. Boston: Addison-Wesley, 2001.

[13] T. Yue, L. Briand, and Y. Labiche, “A systematic review of transformation approaches between user requirements and analysis models,” *Requirements Eng*, vol. 16, pp. 75–99, 2011.

[14] “CaseCompete,” Tech. Rep. [Online]. Available: https://casecomplete.com

[15] “Enterprise architect,” Tech. Rep. [Online]. Available: https://www.sparxsystems.com

[16] J. Thakur and A. Gupta, “Automatic generation of sequence diagram from use case specification,” in *Proc. 7th India Software Engineering Conference. Association for Computing Machinery*, New York, NY, USA, 2014, pp. 1–6.

[17] S. Cook, C. Bock, P. Rivett, T. Rutt, E. Seidewitz, B. Selic, and D. Tolbert, “Unified modeling language (UML) version 2.5.1,” Object Management Group (OMG), Standard, Dec. 2017. [Online]. Available: https://www.omg.org/spec/UML/2.5.1

[18] B. Hnatkowska and P. Zabawa, “Use-case flow (UCF) case-studies,” Repository, 2023. [Online]. Available: https://github.com/bhnatkowska/UCF