

Formal Testing from Natural Language in an Industrial Context

Augusto Sampaio^(✉) and Filipe Arruda

Centro de Informática, Universidade Federal de Pernambuco, Recife, Brazil
acas@cin.ufpe.br

1 Overview

We present some results on developing formal testing strategies and tools for mobile applications, in the context of a partnership with Motorola, a Lenovo company. Actually, the overall scope is much larger, encompassing image processing, optimisation algorithms, sentiment analysis, energy-aware software design, and other research areas.

Our focus here is on testing. The input to the process are text documents written in natural language. There are two major scenarios. In the first one (see Fig. 1), the text documents specify requirements written in a (controlled) natural language, with well-defined syntax and semantics. A formal model is automatically derived from these requirements, from which test cases are automatically generated. These test cases can be expressed in natural language (for manual execution) or as scripts of an automation framework, like UIAutomator [7].

The second scenario is more challenging: the text documents are test cases written in natural language following no standard whatsoever (Fig. 2). There is no independent requirements specification; the test cases are the requirements. In this case, for the purpose of automation, we use natural language processing techniques to match test steps in natural language with test actions already automated in a database. When there is no match, we adopt capture & replay techniques to carry out automation and execute the test cases in the mobile phone; these actions with their respective scripts are then included in the database for further reuse.

Our major, medium-term, objective is to build a single and integrated framework to support the generation, selection automation and execution of test cases from natural language requirements, as displayed in Fig. 3. In this framework, the input to the automation step might be automatically generated as in the first scenario, Test Cases (CNL) in the figure, or input by a tester: Textual Test Cases, in the figure. In the first case, the test case descriptions follow a standard and the automation can be fully mechanised. Nevertheless, the option for the tester to use the framework to automate textual test cases coming from other sources must be available, as test cases for manual execution are also produced without requirements, based on the test designer expertise.

In the next section, we present the current status of the tools (Sect. 2) that independently mechanise these two scenarios. The underlying formalisms are

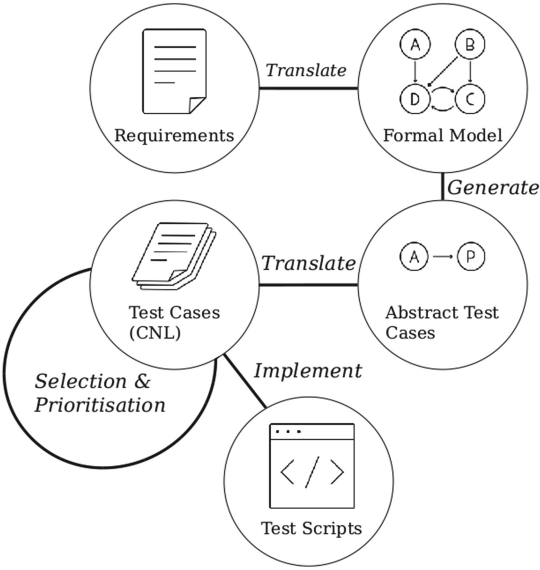


Fig. 1. First scenario

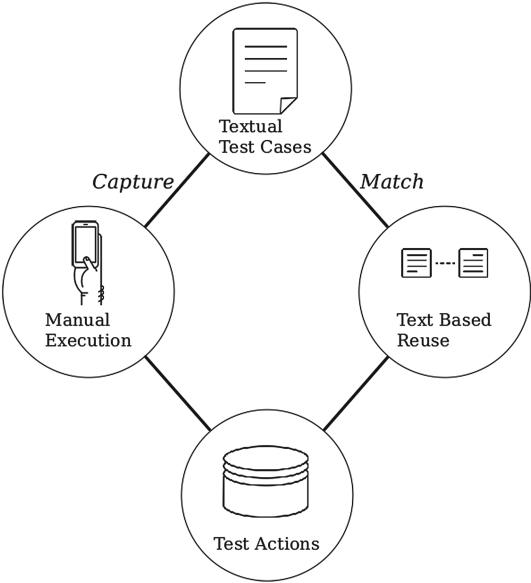


Fig. 2. Second scenario

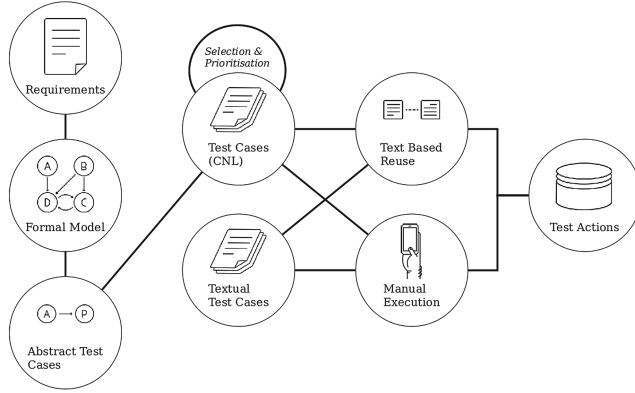


Fig. 3. Proposed framework

considered in Sect. 3. In Sect. 3.1 we show how the CSP process algebra is used as a basis to the test generation strategy; we emphasise the modularity of the approach. In Sect. 3.2 we propose a notion of test step consistency and show how it can be automated in Alloy. In the final section we discuss ongoing work and the remaining steps to the full development of the proposed framework.

2 Tools

The two scenarios introduced in the previous section are supported by practical tools, although at present they are not integrated. These tools are briefly described below.

2.1 Test Generation with TaRGeT

This section is based on material presented in [12, 14]; particularly, we use a simplified version of the illustrative example given in [14]. Our focus is on the generation of black-box (functional) test cases. The input to the process are use case templates, which describe interaction of a user with the system through natural language sentences of three kinds: user actions, system states and system responses. All these sentences must follow a writing standard defined in terms of a controlled natural language (CNL). An example is presented in Fig. 4, which describes a use case to move a message to an important folder.

A template, as the one in Fig. 4, describes a use case that is part of the specification of a feature (mobile device functionality). A use case defines several execution flows (main, alternative or exception flows) each one representing a relevant scenario. The main one describes the *happy path*. In our example, the main flow successfully captures moving a message to the important folder.

An alternative flow involves a choice; during the execution of a flow it might be relevant to engage in an alternative behaviour. If an event from an alternative flow happens, the execution proceeds behaving according to the specified

Description

This use case moves messages from Inbox folder to Important Messages folder.

Main Flow

Description: Moving Messages with success

From Step: START

To Step: END

Step Id	User Action	System State	System Response
1M	Scroll to a message		Message is highlighted.
2M	Select “Move to Important Messages” option.	Message storage has enough space.	“Message moved to Important Messages folder” is displayed.

Alternative Flow

Description: Cannot move messages

From Step: 2M

To Step: END

Step Id	User Action	System State	System Response
1A	Clean up messages	Message storage has not enough space.	Clean up is performed.

Fig. 4. Use case template

alternative behaviour. In our example, the alternative flow captures the situation when a message cannot be moved because there is no more storage space.

Templates like this one are the input to the TaRGeT tool [8, 12], whose purpose is to mechanise a test case generation strategy that supports the steps presented in the first scenario discussed in the previous section. The tool generates test cases, also written in CNL, which include the test procedure, a description and related requirements. Moreover, the tool can exhibit traceability information relating test cases, use cases and requirements.

TaRGeT inputs and processes information in use case templates, first checking adherence to the CNL and, if the sentences obey the CNL, it generates test suites. An overview of the input and output artifacts is presented in Fig. 5, which shows a use case template as the input to the tool, and the output is a test suite with CNL test cases for manual execution, in the form of an excel file. The tool also implements several selection mechanisms, based on similarity algorithms and test purposes; this latter selection criteria is discussed in Sect. 3.1.

2.2 Test Automation with Zygon

As mentioned before, there is a challenging testing scenario in which the text documents provided as input are test cases written in natural language. Because

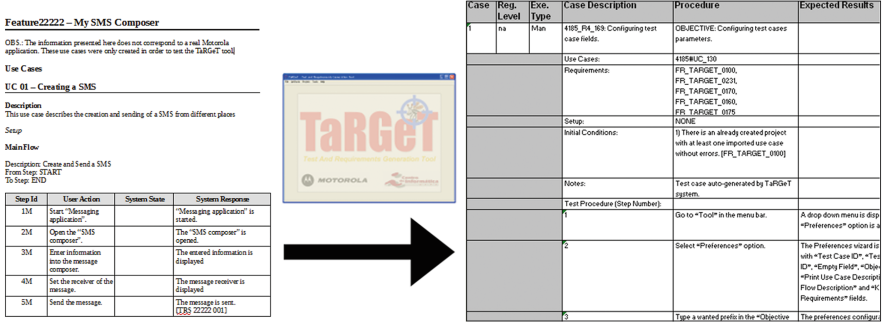


Fig. 5. The TaRGeT tool

there is no formal standard to write these test cases, it becomes difficult to define a meaningful mapping between the text descriptions and an automation code script. Thus, code scripts become scattered because there is no straightforward means to match and reuse test cases already automated.

Therefore, we proposed an intermediate-layer notation called *test action* to fill this gap and implemented it in a tool called Zygon; the description presented in this section (including the figures) is closely based on [2]. A test action, based upon the composite pattern, is a recursive structure that supports several abstraction layers, composition and code-level interpretation for the atomic actions (Fig. 6).

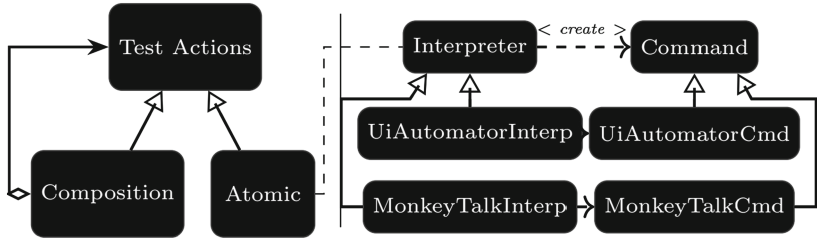


Fig. 6. Overall architecture

The idea behind this proposition is to represent every piece of information (that ranges from a simple test step to a complete TC or even a test suite) uniformly, allowing their retrieval or execution regardless the artifact category. In short, we have shifted from a monolithic (Fig. 7) to a hierarchical (Fig. 8) mapping between natural language descriptions and GUI operations.

The test action representing the TC illustrated in Fig. 7, that checks whether an email can be sent, could be composed by several actions, which in turn could also be a composition of other atomic actions (screen interactions), as shown in Fig. 8. This potentialises the reuse and reduces effort by only mapping code scripts to atomic actions.

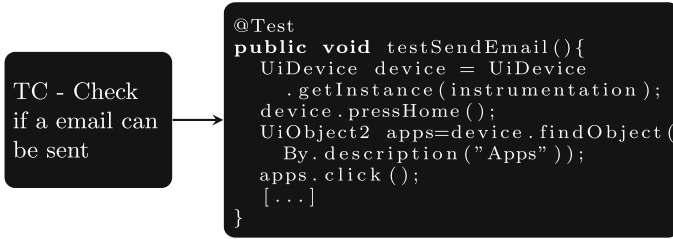


Fig. 7. Typical TC automation based on capture & replay

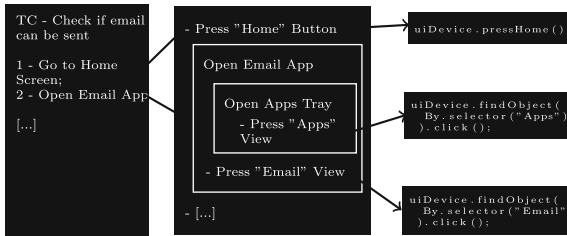


Fig. 8. Test case automation using hierarchical test actions

Because test actions may be organized and composed in any order to create others yet more complex, we employ an algorithm to match each test step with a similar action description stored in the database. In this way, as more actions are automated and stored, the greater is the chance to find a similar one instead of spending time to generate the automation code script. In previous work, for instance, we reported a reuse ratio up to 71% in an industrial context [2].

However, in order to assist testers when there is no previously saved test action that is similar enough, the Zygon tool is able to capture user interactions on the phone and store them as test actions, also giving them a natural language description. It is worth noting that the application UI is web-based and the process is transparent to the user (Fig. 9). In summary, we developed a full-stack solution by which a tester is able to automate an entire test suite without any programming skills during her common activities, reducing both time and effort to automate tests.

Similar to traditional Capture & Replay tools, Zygon also captures user inputs to reproduce them later. However, instead of capturing low-level events in order to strictly reproduce them, the tool listens to the Android accessibility events¹, yielding high-level descriptions of what was performed on the device, mitigating issues with different screen sizes, besides being more meaningful to the

¹ <http://developer.android.com/intl/en-us/reference/android/view/accessibility/AccessibilityEvent.html>.

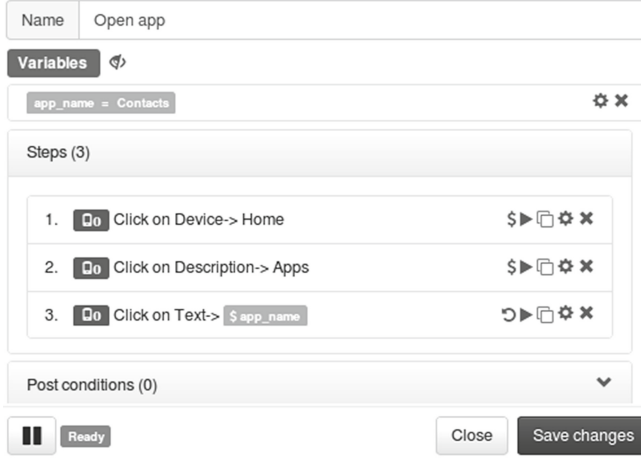


Fig. 9. Capture screen (Zygon)

end user. For instance, instead of “Press the screen at the coordinate (100,250)”, Zygon captures “Click the button with description ‘Apps’”.

A custom keyboard is also installed during the capturing in order to get what the user is typing. However, the user can transform the text typed in a variable to reuse the same action in other situations, as shown in Fig. 9.

3 Underlying Formalisms

In this section we discuss how the underlying formalism (particularly, the process algebra CSP) supports an incremental strategy for test generation (first scenario), and how contracts and Alloy models support a consistent automation strategy for the second scenario; we propose a new notion of test step conciseness, and its mechanisation, in the context of test automation.

3.1 Process Algebraic Approach to Test Generation

In this section we briefly introduce an approach to test case generation based on the CSP process algebra. The details can be found in [14]; the purpose here is to emphasise the modularity supported by this approach, and particularly the fact that it has been conservatively extended to capture several facets of test generation, like control behavior, data, quiescence and time. We explain the CSP notation and the test case generation strategy on demand.

As previously explained, the first step is to translate the (input) use case template into CSP. For the example presented in Fig. 4, the corresponding CSP model is given below.

$$\begin{aligned}
I1 &= START; I1 \\
START1 &= scrollToAMsg \rightarrow msgHighlighted \rightarrow selMoveToIMOpt \rightarrow \\
&\quad reqStoInfo \rightarrow (ALT1 \sqcap ALT2) \\
ALT1 &= msgStoIsNotFull \rightarrow msgMovedToIMDisp \rightarrow Skip \\
ALT2 &= msgStoIsFull \rightarrow performCleanUp \rightarrow reqCleanUp \rightarrow cleanUpOk \rightarrow \\
&\quad msgMovedToIMDisp \rightarrow Skip
\end{aligned}$$

The process $I1$ recursively behaves as the $START$ process (semicolon is sequential composition). The process $START$ engages in a sequence of events captured using the prefix operator; the process $a \rightarrow P$ communicates the event a and then behaves like P . After communicating the last event, $reqStoInfo$, $START$ offers a choice (\sqcap) between the processes $ALT1$ and $ALT2$; the decision is taken by the environment; in CSP terms, this is called external choice. The process $ALT1$ captures the happy path, when the storage is not full and the message can be moved to the Important Messages folder. The process $ALT2$ captures a full storage state; some cleaning up is performed so that the message can be moved to the same folder. These two processes terminate successfully (behaving like $Skip$); when any of them terminates, $I1$ recurses.

As a rich process algebra, CSP offers several additional operators for combining processes; here we use only parallelism and hiding. The process $P \parallel [X] Q$ is the generalised parallel composition of the processes P and Q with synchronisation set X . This means that, regarding events in X , P and Q can only communicate when both are ready to engage in the same events; for the events not in X , each one behaves independently. The interleaved composition $P \parallel\parallel Q$ is a particular case of parallel composition when the synchronisation set is empty. In this case, both processes can evolve totally independently. The process $P \setminus X$ behaves as P , but hides all events in X , making them internal.

The test case generation strategy under consideration is based on the CSP traces model and traces refinement. A process Q refines another process, say P , in the traces model, denoted $P \sqsubseteq_T Q$, if, and only if, $traces\llbracket Q \rrbracket \subseteq traces\llbracket P \rrbracket$. Traces refinement can be automatically verified using, for instance, the FDR tool [9]. If the refinement does not hold, the tool produces a trace (the shortest counter-example), say ce , such that $ce \in traces\llbracket Q \rrbracket$ but $ce \notin traces\llbracket P \rrbracket$. Some facilities are available to make FDR generate subsequent counterexamples, if there are any can be obtained. Two other classical and more elaborate semantic models of CSP are the failures and the failures-divergences models. The first one captures deadlock situations, whereas the latter captures livelock traces as well. Further details can be found, for example, in [16].

Selection criteria can be used to guide the test case generation. The main selection mechanism is via the definition of a test purpose, which allows marking certain traces of the specification; this is also specified as a process in CSP. The effect of marking the relevant traces is achieved by parallel composition of the specification with the test purpose. Consider that a given test purpose, say TP , is defined to select some test scenarios from a specification S . The parallel

composition of S and TP (denoted parallel product), with synchronisation set α_S , is $STP = S \parallel_{\alpha_S} TP$.

Note that, with this synchronisation set (the entire alphabet of S), the test purpose TP synchronises in all events of S until there are no further events to synchronise, when TP communicates an event $mark \in MARKS$, and then both S and TP deadlock. As a result, the parallel product will have the traces of the form $ts = t \frown \langle mark \rangle$, for $t \in tracesS$, where each ts is a test scenario of interest. Because $ts \notin traces[S]$ (due to the $mark$ event), then the shortest counterexample of the refinement $S \sqsubseteq_T STP$, say ts_1 , is generated. If TP does not select any scenario from S , no $mark$ is included in the parallel product STP , and so this will be the same as S ; in this case no counterexample is generated.

In the context of our example, the shortest scenario for moving a message is:

$\langle scrollToAMsg, msgHighlighted, selMoveToIMOpt, msgStoreHasSpace, msgMovedToIMDisp, accept \rangle$

The conformance notion adopted in this approach is the relation **cspio**, intended to capture the **ioco** [21] relation in the CSP setting. As an informal intuition, consider an arbitrary trace σ of the specification. Then $I \text{ cspio } S$ holds provided, for all such traces, the set of output events of the implementation, after performing σ , is a subset of the outputs performed by S after σ . The standard semantic models of CSP do not distinguish between inputs and outputs, but this is essential in testing. Here we assume that the alphabet of a process is split into disjoint input and output sets of events. In the formulation, it is enough to reference the set of output events, which we denote \mathcal{O} . The relation **cspio** is formalised by the following definition.

Definition 1 (CSP input-output conformance).

$$I \text{ cspio } S \triangleq \forall \sigma : traces[S] \bullet out(I, \sigma) \subseteq out(S, \sigma)$$

where $out(R, \sigma) = \{a : \mathcal{O} \mid \sigma \frown \langle a \rangle \in traces[R]\}$

cspio fully captures **ioco** if the CSP processes (S and I) are annotated with quiescence (δ).

Theorem 1 [14] below captures **cspio** using process refinement.

Theorem 1 (Verification of cspio).

$$I \text{ cspio } S \Leftrightarrow S \sqsubseteq_T (S \triangle ANY(\mathcal{O}, STOP)) \parallel [\Sigma] I$$

where $ANY(X, R) = \Box a : X \bullet a \rightarrow R$.

With the result established by Theorem 1, it is possible to mechanically verify $I \text{ cspio } S$ using a tool like FDR, provided, of course, there is an implementation model I .

The relation **ioco** is defined in a model called Straces [21]. This model explicitly includes a special event to represent quiescence (δ). Although there is no implementation of suspension traces in any refinement checker for CSP, it is possible to automate verification via an encoding as standard traces refinement. As shown in [17], if all quiescences are identified in the traces as the special output δ , then the relations **cspio** and **ioco** coincide.

Although the refinement assertion in Theorem 1 captures **cspio** conformance, it does not show how quiescent states (δ) of S and I are effectively signaled. To take advantage of Theorem 1 in a mechanisation of conformance verification for **ioco**, we use a notion of priority for CSP processes in [15]. We define, for a process P , a corresponding process P_δ that outputs δ in all quiescent states of P [5].

Definition 2.

$$P_\delta \triangleq \text{prioritise}(P \parallel \text{RUN}(\{\delta\}), \langle \mathcal{O}, \{\delta\} \rangle)$$

The behaviour of $\text{prioritise}(P, R)$ is similar to that of P , but it prevents any event in X_i in the relation R (represented as an ordered sequence), for $i > 1$, from taking place when τ (an internal event), \checkmark (termination) or an event in some X_j , with $j < i$, is possible. The events in X_1 have the same priority as that of τ and \checkmark . Events not in R are incomparable to all other members of R .

The fact that the event δ happens only in the absence of output is captured by the order of the sets $\langle \mathcal{O}_{UT}, \{\delta\} \rangle$ in the prioritise operator, which prioritises output events over δ .

The following theorem [5] captures our proposed strategy for **cspio** taking quiescence into account. We use \mathcal{O}_δ as an abbreviation of $\mathcal{O} \cup \{\delta\}$. Similarly, Σ_δ stands for $\Sigma \cup \{\delta\}$.

Theorem 2 (Verification of cspio).

$$I \text{ cspio } S \Leftrightarrow S_\delta \sqsubseteq_T (S_\delta \triangle ANY(\mathcal{O}_\delta, STOP)) \parallel [\Sigma_\delta] I_\delta$$

Regarding soundness, initially, we proved that the encoding in the traces model captures **ioco** [14]. Currently, we are defining a new Straces model for CSP, including the definition of all operators and the relevant healthiness conditions. Some initial results are presented in [5].

The mechanised verification of conformance, captured as a refinement expression, is an important advantage of a formalisation using a process algebra like CSP. Unlike an explicit algorithm for checking conformance, as presented in [22] for **ioco**, we benefit from the expressive power of the refinement notions and the model checker for CSP to verify conformance in a simple way.

An example of the modularity of this approach is that quiescence is handled in an orthogonal way, preserving the structure of the conformance verification theorem. Similarly, as shown in [14], state can also be incorporated as a

conservative extensions of the presented conformance verification strategy. The model that specifies the control behaviour of a use case is composed in parallel with a process that represents an abstract memory to record the state of variables. Despite this model increment, the test generation strategy is entirely reused.

We have also explored another application domain (timed reactive systems). In [4], we discuss how the test generation strategy can be incrementally evolved to address time aspects (in addition to control and data), in an orthogonal way.

We are currently considering the extension of the strategy to handle hybrid systems; this is very challenging, as there is a shift of paradigm, due to the complexity inherent to dynamic systems. This produces a vertical impact on the overall strategy. For the CNL, the main change is to add structure so that the requirements engineering is able to write differential equations, in addition to the textual presentation. However, the other steps of the strategy require more radical adaptations. We are currently working on the definition of a conformance relation that combines the discreet features of **ioco** with tolerance (output and timed values) margins as is common in relations for dynamic systems [1, 6].

3.2 Contract Based Approach to Consistent Automation

Regarding test automation from test cases described in natural language, despite the fact that we were able to uniformly represent every test artifact improving the reuse among them, we still faced some problems concerning consistency and dependency management. There was no way to guarantee, for instance, that a sequence of test actions could actually be correctly executed. It was sorely dependent on the tester or test engineer experience and individual knowledge about the given domain.

Further subtle problems appeared in the execution stage when multiple test cases, although consistently composed, rely upon prior configurations that could also interfere or even cancel other dependencies. For that matter, we had to develop a strategy to automatically check consistency of individual test actions as well as their dependencies, in order to provide a coherent (and possibly optimal) execution order.

The strategy consists on defining: which actions are individually valid; what are their dependencies and behaviors; and how to correctly dispose actions or what actions can be inserted to allow the execution of a set of test cases. The valid actions and their dependencies are represented as a domain model that is automatically translated into Alloy [11] signatures, facts and predicates, see Listing 1.1. Then, for every execution request, a predicate is evaluated to find a valid sequence of test actions including those requested.

Listing 1.1. Alloy model

```

open util/ordering[State]
2 sig State{conditions: set Action, current: one Action} {
    some this.next implies migrate[this, this.next]
4 dependenciesAreSatisfied[current, conditions]
}
6 sig Action {operation: Operation, patient: Patient}{
    operationIsValid[operation, patient]
8 }
abstract sig Operation{ dependsOn: set Operation,
10                      cancels: set Operation }
abstract sig Patient{}
12 [.] //Placeholder  $\forall$  operations and patients
    //Example
14 one sig Logout extends Operation{}{
    cancels = Login and dependsOn = Login
16 }
[.]
18 pred directDependenciesAreSatisfied( action: Action,
    conditions: set Action) {
20     let directDependencies = action.operation.dependsOn •
        directDependencies = none or (
22          $\forall$  directDependency: directDependencies •
            some condition: conditions •
24             condition.operation in directDependency
            and condition.patient in action.patient
26         )
    }
28 pred extrasDependenciesAreSatisfied(a: Action,
    conditions: set Action){
30     let extraRel = extraDependencies[a.operation][a.patient] •
        no extraRel or some cond: conditions •
32         (cond.operation -> cond.patient) in extraRel
    }
34 pred new(a: Action, o: Operation, patient: Patient){
    a.operation in o and a.patient in patient
36 }
pred migrate(s: State, s': State) {
38     s.current in s'.conditions
    s'.conditions - s.current in s.conditions
40     removeCanceledActions(s, s')
}
42 pred dependenciesAreSatisfied( action: Action,
    conds: set Action){
44     extrasDependenciesAreSatisfied[action, conds]
    directDependenciesAreSatisfied[action, conds]
46 }
pred addStep(s: State, action: Action){
48     s.current = action
}

```

By referring to Listing 1.1 from lines 2 to 5, we define the concept of the state of a system for the purpose of test case execution. A state comprises a set of conditions (actions that were executed in past states) and the current action to be executed. An action, in turn, is an operation over a patient (lines 6–11) such as “Send an Email” or “Press a button”. All operations and patients from the domain model would be automatically rendered between lines 12 and 17.

It is worth mentioning that some operations have inherent dependencies: A “logout” operation can only be performed after “login”; similarly, to “delete” a message, one has to have been created before (via `Operation.dependsOn`). On the other hand, if a “delete” operation is performed, we have to remove the action “create” from the conditions of the next state (via `Operation.cancels`).

We check these inherent dependencies evaluating the predicate defined between the lines 18 and 27: the direct dependencies are satisfied if a given action has no dependencies or if its dependencies are present in the set of conditions of the current state. Besides these direct dependencies, we have to also check indirect ones, such as: to send an email, one has to ensure first that there is an active connection to the internet. These “extra” dependencies are checked by evaluating the predicate defined between lines 28 and 33. With a valid action and all dependencies satisfied (a valid state of the system), then the migration to the next state happens. This is covered by the predicate defined between lines 37 and 41, by which all conditions plus the current action become the conditions of the next state, but removing the actions from the conditions of the next state that were canceled by the current action.

Example. A very simple example of finding a correct sequence of execution can be illustrated by trying to execute the action “Send an email”. Testers with no experience could naively try to build a sequence with “Turn the WiFi on and then Send an email”, but considering a default scenario that no action was performed before, and by analysing the dependency graph in Fig. 10, such a test sequence is considered inconsistent. The proposed consistency analysis strategy can easily warn the tester that this sequence cannot be executed, by evaluating the predicate referred in Listing 1.2.

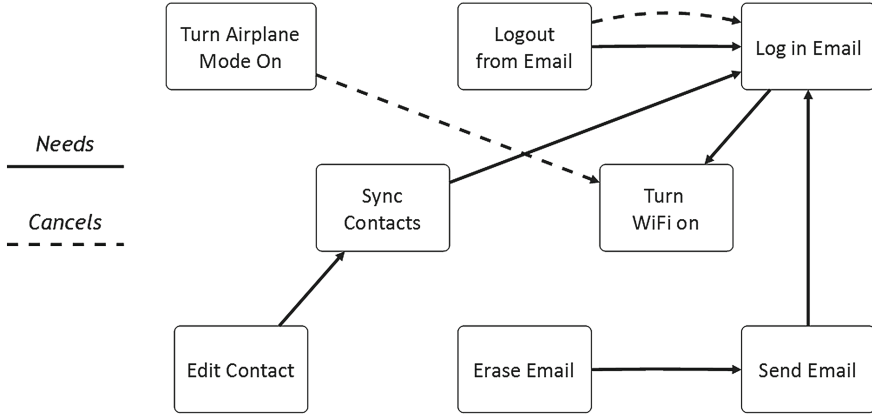
Listing 1.2. Testing consistency

```

1  pred test{
2      let s=first, s'=s.next•
          s.conditions = none and
4          some activateWiFi, sendEmail: Action • {
              new[activateWiFi, Activate, WiFi]
6              new[sendEmail, Send, EmailMessage]
              addStep[s, activateWiFi]
8              addStep[s , sendEmail]
          }
10 }

12 run test for 10

```

**Fig. 10.** Dependency graph

Since the given sequence is detected as inconsistent, one can ask the Alloy Analyzer to find a valid one, which is achieved by evaluating the predicate in Listing 1.3. In this predicate, it is first assured that no actions were performed in the initial state and, for every state in the system, a valid action must be performed. Then, we declare that one of these valid actions should be “Send an Email”. In this case, the Alloy Analyzer finds an instance of the model that satisfies this predicate: Turn WiFi On → Login Email → Send Email, which give us the final state described in Fig. 11.

Listing 1.3. Finding dependencies

```

pred findDependencies{
2   first.conditions = none
   ∀ s: State •
4     some anyAction: Action • {
       new[anyAction, Operation, Subject]
6       addStep[s, anyAction]
     }
8   some s: State •
       some sendEmail: Action • {
10      new[sendEmail, Send, EmailMessage]
       addStep[s, sendEmail]
12    }
}

```

In summary, a simple dependency analysis can both detect inconsistent sequences, as well as automatically insert actions to turn an inconsistent sequence into a consistent one. The main challenge here is scalability; this and other concerns are considered in the next section.

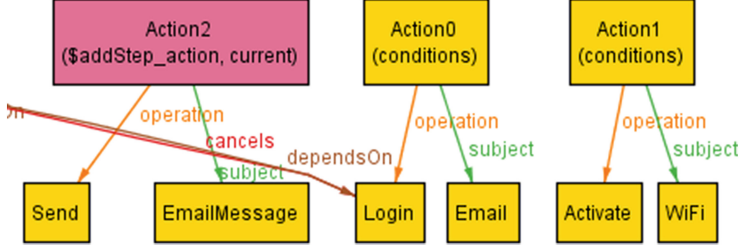


Fig. 11. Instance of our model with a valid sequence to “Send an email”

4 Ongoing Work: Integrated Framework

We are currently working on the integration of the strategies and tools for test case generation and automation, as already summarised in Fig. 3. The main activities involved are discussed below.

Controlled Natural Languages. In order to be able to automate the textual test cases generated by TaRGeT, we need to revise the current use case templates so that the descriptions are presented in sufficient detail to allow Zygon to match all the steps with actions in the database, without requiring capture. Therefore, we need a writing standard common to both TaRGeT and Zygon, which, currently, is not the case. The challenge relies on how to connect both languages since they have different abstraction levels: the former is usually specified in general terms while the latter is tightly coupled to the UI implementation and describe concrete actions in full.

The CNL for representing test actions (as these are stored in a database) is heavily built upon the concept of a frame, which is a structure to store data about a previously known situation [13]. These frames contain prefixed *slots* or *terminals* that, when filled, represent an instance of a specific situation. Therefore, we can automatically build the frames from the use cases, but the frames would still miss some important properties, such as the default values for all possible *slots* (in order to allow generic statements) and rules for valid instances. To fill this gap, we have two choices under analysis: (1) compel users to give sufficient details yet inside use cases or (2) build a detached domain knowledge model to provide a bridge between use case and test action CNLs, with the missing information.

Soundness and Consistency Notions. The approach to test generation is based on a well-defined conformance relation (**cspio**). As usual, this assumes as test hypothesis that both the model and the implementation can be specified in the formal model (in our case, as CSP processes), so that they can be related. Soundness of the generated test cases, in this context, means that if the execution of a test case gives a fail verdict, then the implementation is non-conforming.

On the other hand, concerning the capture & replay approach, there are no requirements or models, so there is no reference for defining conformance. As explained in the previous section, we defined a consistency notion to check whether the sequence of test actions in a test case is coherent in the sense that the set up for executing a given action is ensured by previous actions in the sequence.

An interesting aspect to consider is whether it makes sense to promote this notion to the use case level. This would allow to ensure that the steps of the generated test cases be consistent by construction. This is a complementary notion to that of soundness, already proved. However, as discussed in the previous topic on considerations about the CNL, this consistency notion can be associated with a detailed use case template or with a detached domain knowledge model.

Populating the Database. Assuming that we have a finite number of (non-recursive) frames, it becomes feasible to pre-populate the database with automated actions prior to the test case generation, since the (parametrised) test action will be executed the same way in spite of variations of slot values (similar to the behavior of variables in a program) [13]. In complex systems, however, this approach might not be practical because we would lose too much time trying to automate all the frames at once. As an alternative, we can employ an interactive, on demand approach, as it is currently used in existing projects.

Scalability. The evaluation carried out by the Alloy Analyzer to find a valid sequence of steps may be impractical, specially when the domain model has a huge number of different actions and patients. For a model representing only a specific application, the time to find a valid sequence for an Alloy scope of 10 instances is usually negligible, but that is not the case when testing multiple apps in a mobile platform. For that matter, we are exploring the alternative of partitioning the dependency graph to consider only reachable nodes for particular applications, in order to reduce the Alloy model and consequently the number of combinations to be analysed.

Some Related Approaches. The proposed framework integrates textual test case generation from use case descriptions and an approach to test case automation, based on capture & replay, from textual test case descriptions. A detailed comparison of work related to each strategy can be found in [2,14]. Here we mention just a few examples.

Concerning textual test case generation from use case descriptions, with the aim of GUI testing, some relevant approaches are, for instance, [3,10,19]. The approach described in [19] is closely related to ours, since it uses natural language for the specification of use cases, maps use cases to a formal model (FSM) and generates textual test cases.

The search for an optimal mapping between natural language description and concrete tests has also been an active research area. Some examples

are [18,20,23]. Cucumber [18], for instance, assists the writing of acceptance tests in a behavior driven development environment: parameterised scenarios are written in natural language and semi-automatically mapped to a source code or stub, in order to accelerate the process and provide a better tracking. However, besides the implementation being developer-centric, in-depth reuse and consistency/dependency checking are outside the scope of the tool.

The distinguishing feature of the proposed framework is to integrate two promising strategies (and related tools), which have been used in an industrial context; together, they allow a mechanised generation of automated test cases from natural language use cases, benefiting from the extensibility of both strategies, as well as from soundness and consistency notions, as previously explained. Nevertheless, each strategy and tool can still be used in isolation.

Acknowledgments. The work described here had the contribution of several colleagues: Hugo Araujo, Flavia Barros, Ana Cavalcanti, Gustavo Carvalho, Alexandre Mota and Sidney Nogueira, among others.

References

1. Abbas, H., Hoxha, B., Fainekos, G., Deshmukh, J.V., Kapinski, J., Ueda, K.: Conformance testing as falsification for cyber-physical systems (2014). arXiv preprint: [arXiv:1401.5200](https://arxiv.org/abs/1401.5200)
2. Arruda, F., Sampaio, A., Barros, F.: Capture and replay with text-based reuse and framework agnosticism. In: Proceedings of the 28th International Conference on Software Engineering and Knowledge Engineering. KSI Research Inc. <http://dx.doi.org/10.18293/SEKE2016-228>
3. Bertolino, A., Gnesi, S.: Use case-based testing of product lines. *ACM SIGSOFT Softw. Eng. Notes* **28**(5), 355–358 (2003)
4. Carvalho, G., Sampaio, A., Mota, A.: A CSP timed input-output relation and a strategy for mechanised conformance verification. In: Groves, L., Sun, J. (eds.) *ICFEM 2013*. LNCS, vol. 8144, pp. 148–164. Springer, Heidelberg (2013). doi:[10.1007/978-3-642-41202-8_11](https://doi.org/10.1007/978-3-642-41202-8_11)
5. Cavalcanti, A., Hierons, R.M., Nogueira, S., Sampaio, A.: A suspension-trace semantics for CSP. In: 10th International Symposium on Theoretical Aspects of Software Engineering, TASE 2016, Shanghai, China, 17–19 July 2016, pp. 3–13 (2016). <http://dx.doi.org/10.1109/TASE.2016.9>
6. Dang, T., Nahhal, T.: Coverage-guided test generation for continuous and hybrid systems. *Formal Methods Syst. Des.* **34**(2), 183–213 (2009)
7. Android Developers: UiAutomator (2016)
8. Ferreira, F., Neves, L., Silva, M., Borba, P.: TaRGeT: a model based product line testing tool. In: Tools Session of CBSOFT (2010)
9. Goldsmith, M., Roscoe, B., Armstrong, P.: Failures-divergence refinement-FDR2 user manual (2005)
10. Hartmann, J., Vieira, M., Foster, H., Ruder, A.: A UML-based approach to system testing. *Innov. Syst. Softw. Eng.* **1**(1), 12–24 (2005)
11. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. MIT Press, Cambridge (2012)

12. Machado, P., Sampaio, A.: Automatic test-case generation. In: Borba, P., Cavalcanti, A., Sampaio, A., Woodcock, J. (eds.) PSSE 2007. LNCS, vol. 6153, pp. 59–103. Springer, Heidelberg (2010). doi:[10.1007/978-3-642-14335-9_3](https://doi.org/10.1007/978-3-642-14335-9_3)
13. Minsky, M.: A framework for representing knowledge (1975)
14. Nogueira, S., Sampaio, A., Mota, A.: Test generation from state based use case models. *Form. Asp. Comput.* **26**(3), 441–490 (2014). <http://dx.doi.org/10.1007/s00165-012-0258-z>
15. Roscoe, A.W.: *Understanding Concurrent Systems*. Springer Science & Business Media, London (2010)
16. Roscoe, A.: *The Theory and Practice of Concurrency*. Prentice Hall Series in Computer Science. Prentice-Hall, Englewood Cliffs (1998)
17. Sampaio, A., Nogueira, S., Mota, A., Isobe, Y.: Sound and mechanised compositional verification of input-output conformance. *Softw. Test. Verif. Reliab.* **24**(4), 289–319 (2014). <http://dx.doi.org/10.1002/stvr.1498>
18. Soeken, M., Wille, R., Drechsler, R.: Assisted behavior driven development using natural language processing. In: Furia, C.A., Nanz, S. (eds.) TOOLS 2012. LNCS, vol. 7304, pp. 269–287. Springer, Heidelberg (2012). doi:[10.1007/978-3-642-30561-0_19](https://doi.org/10.1007/978-3-642-30561-0_19)
19. Some, S.S., Cheng, X.: An approach for supporting system-level test scenarios generation from textual use cases. In: *Proceedings of the 2008 ACM Symposium on Applied Computing*, pp. 724–729. ACM (2008)
20. Thummalapenta, S., Sinha, S., Singhania, N., Chandra, S.: Automating test automation. In: *2012 34th International Conference on Software Engineering (ICSE)*, pp. 881–891. IEEE (2012)
21. Tretmans, J.: Test generation with inputs, outputs and repetitive quiescence. *Software—Concepts and Tools (TR-CTIT-96-26)* (1996)
22. Weiglhofer, M., Wotawa, F.: On the fly input output conformance verification. In: *Proceedings of the IASTED International Conference on Software Engineering*, pp. 286–291. ACTA Press (2008)
23. Wong, E., Zhang, L., Wang, S., Liu, T., Tan, L.: Dase: document-assisted symbolic execution for improving automated software testing. In: *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, vol. 1, pp. 620–631. IEEE (2015)

Formal Methods: Foundations and Applications

19th Brazilian Symposium, SBMF 2016, Natal, Brazil,

November 23-25, 2016, Proceedings

Ribeiro, L.; Lecomte, T. (Eds.)

2016, X, 253 p. 62 illus., Softcover

ISBN: 978-3-319-49814-0