# Using Refinement Checking as System Testing

Cristiano Bertolini and Alexandre Mota

Federal University of Pernambuco, Centre of Informatics, P.O.Box 7851, Cidade Universitária, CEP 50732-970, Recife, PE, Brazil
cbertolini,acm@cin.ufpe.br

**Abstract.** Software testing is an expensive and time-consuming activity; it is also error-prone due to human factors. But, it still is the most common effort used in the software industry to achieve an acceptable level of quality for its products. An alternative is to use formal verification approaches, although they are not widespread in industry yet. This paper proposes an automatic verification approach to aid system testing based on refinement checking, where the underlying formalisms are hidden from the developers. Our approach consists in using a controlled natural language (a subset of English) to describe requirements (where it is automatically translated into the formal specification language CSP) and extracting a model directly from a mobile phone using a developed tool support; these artifacts are normalized in the same abstraction level and compared using the refinement checker FDR. This approach is being used at Motorola; the source of our case study.

**Key words:** System Testing, Refinement Checking, CSP

## 1 Introduction

Although formal methods are considered difficult to be used in practice, formal models are a current trend in the software industry. These models are used, for instance, in analysis, design and testing. In software testing, many approaches are using formal models to automate test case generation [1] as well as to improve the quality of the process of testing itself [2].

This work is the result of a research partnership between Motorola and the Centre of Informatics. We are using the formal language CSP (Section 2.1); a language primarily designed to model concurrent and distributed systems. Also, we use automatic verification based on refinement checking [3] (Section 2.2) using hidden formal models. That is, developers keep using their traditional artifacts and get the expected results without knowing that internally formal models are being handled. The main motivation of our approach is to avoid human errors (increase quality level) as well as to increase productivity (minimize effort).

A general overview of our approach (Section 3) is shown in Figure 1. We consider two inputs: a requirements document written in a Controlled Natural Language—CNL[1] (a subset English), where we reuse the work reported in [4]

[1] The reason to use CNL instead of a graphical language, say UML, is related to the artifacts used at Motorola; they are written in English.

to generate a CSP specification from that document (Section 3.1), and a mobile phone, where we developed a tool called BxT (Section 3.2) to extract a CSP specification from the phone automatically using the source code of a Motorola proprietary test framework called PTF [5]. With both models available, we adjust their abstraction levels (Section 3.3), because the implementation model has more details than the requirement model, and apply a proposed algorithm based on refinement checking, performing a kind of conformance testing, to analyze and report the problems found (Section 3.4). Note that, the whole approach hides the formalisms from the developers.

Although we have focused on mobile phones, any application could be used.

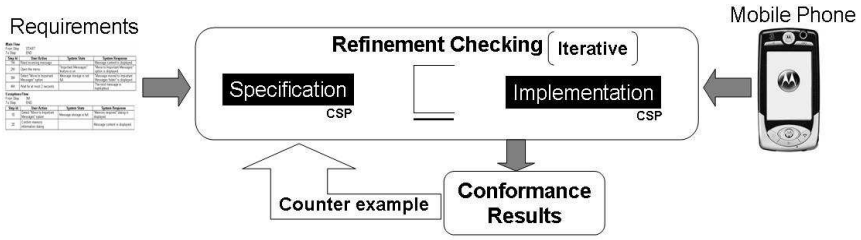


**Fig. 1.** Approach Overview.

Finally, to show the advantage of the proposed approach, we present a case study (Section 4) based on a real mobile application. Also, we consider some related work (Section 5), conclusions and future works (Section 6).

Therefore, the main contributions of this paper are:

- Propose an automatic approach to improve the quality of system testing in the context of mobile applications;
- Develop a tool (BxT) to extract the behavior of the mobile phones as a CSP formal specification;
- Show how to adjust the different abstraction levels of the requirement and implementation specifications to allow a correct refinement checking;
- Present an algorithm based on refinement checking to capture all faults found between requirements and implementation in the style of a testing approach.

## 2 Background

In this section we give an overview of model checking and the language CSP.

### 2.1 CSP Overview

The basic idea of CSP is to represent a system as a set of processes interacting with each other using actions (events). The alphabet of a process $P$, represented

as $\alpha P$, is the set of events this process uses. To use CSP in practice, we have to use its machine-readable version named $\text{CSP}_M$; all CSP elements used in this paper use the $\text{CSP}_M$ version. Once we have a $\text{CSP}_M$ specification, we can apply the model checker FDR [6] (Failures-Divergences Refinement).

We use the following CSP specification from our case study to introduce CSP.

```
SysImpl = P1_Hiding ; SysImpl
P1_Hiding = P1 \ {|goto.DTGOT_SCREEN.(IDLE_SCREEN,{}),
goto.DTGOT_MENU.(MAIN_MENU,{})|}
P1 = steps -> goto.DTGOT_SCREEN.(IDLE_SCREEN,{}) ->goto.DTGOT_MENU.
(MAIN_MENU,{}) -> start.DTSTA_APPLICATION.(IM_APPLICATION,{})
-> (P2 [] PA1)
P2 = steps -> login.DTLOG_APPLICATION.(IM_SERVER_APPLICATION,{})
-> (P3 [] SKIP)
P3 = steps -> select.DTSEL_LISTITEM.(CONTACT_ITEM,{}) -> P4
P4 = steps -> start.DTSTA_LISTITEM.(CONVERSATION_ITEM,{}) -> SKIP
PA1 = steps -> goto.DTGOT_LIST_ITEM.(SAVED_CONVERSATION_FOLDER,{})
-> SKIP
```

This CSP specification captures the behavior of the implementation of a real mobile phone. It starts with the description of the main process `SysImpl`. This process represents a sequential composition (`;`) between the process `P1_Hiding` and itself. Thus `SysImpl` behaves like `P1_Hiding` until a successful termination occurs (we explain this below). In this case, it becomes `SysImpl` again.

The hiding operator (\) is used to abstract some details of a process. For instance, the process `P1_Hiding` hides the events `goto.DTGOT_SCREEN.(IDLE_SCREEN,{}),goto.DTGOT_MENU.(MAIN_MENU,{})` from the process `P1`.

A prefix `(a -> P)` combines an event and a process to produce a new process; a linear behavior. In `steps -> goto.DTGOT_SCREEN.(IDLE_SCREEN,{}) ->...`, after `steps` occurs, the `goto` event is the first event of the resulting behavior.

When we have an external choice (`[]`), it determines a choice between two processes. The environment decides on which process to engage. For instance, in `( P2 [] PA1 )` the environment can choose process `P2` or `PA1`.

The process `Skip` means a successful termination. And the process STOP means a problematic behavior; a deadlock. It can also serve to other purposes (see Section 3.4).

Three complementary semantic models describe the semantics of CSP: traces, failures and failures-divergences [3]. This work uses the traces model to describe process behavior. It is the simplest of all models and captures what a process can do in the form of a set of sequences of events. Thus, for any CSP process `P`, the function $\mathcal{T}$`(P)` gives its traces.

A non-deterministic choice between two CSP processes is represented as `P |~| Q`. In the traces model, we cannot capture non-determinism because $\mathcal{T}$`(P [] Q)` $= \mathcal{T}$`(P |~| Q)` $= \mathcal{T}$`(P)` $\cup \mathcal{T}$`(Q)`. This is useful for this work because test cases are very similar to traces and then the traces model is sufficient.

Having the traces of a processes, we can check their relation via refinement. Let P and Q be two CSP processes, then $\texttt{P} \sqsubseteq_T \texttt{Q} =_{def} \mathcal{T}\ (\texttt{Q}) \subseteq \mathcal{T}\,(\texttt{P})$ that is, Q is better than P if Q has less than or the same possible behaviors of P.

Furthermore, CSP processes can also be represented semantically (using its operational semantics) in terms of LTS (Labelled Transition Systems); a kind of labelled graph (see Figure 5(a) for an example of LTS).

### 2.2 Model and Refinement Checking

Model checking is an automatic technique to check whether a given model satisfies a given desired property. Usually, model checking is used on hardware design, but currently it is being used on software as well [7]. Properties are written as temporal logic formulas (for example, CTL or LTL [8]) and models in some concurrent specification language (for example, CSP and PROMELA [9]). Then, given a temporal logic formula $f$ and a model $M$, model checking, which is represented as $M \models f$, means checking whether $M$ satisfies $f$. In practice, this checking involves representing $M$ and $f$ as LTS (Labelled Transition Systems), a kind of graph, and performing a search using these structures. The essence about this is that model checking is exhaustive (it uses the entire structure if it is necessary [10]) and it obviously needs that such structures be finite.

A special kind of model checking, called refinement checking, uses two models written in the same concurrent specification language instead of temporal logics. Its goal is to check whether a model $Q$ is better than another model $P$ in the sense that all properties of $P$ are preserved by $Q$. Formally, given models $P$ and $Q$, one says that $Q$ refines $P$ whether $P \sqsubseteq Q$ holds. To connect this new view with the previous one, we can think that $P$ is a model that is already known to have a certain property given by formula $f$. Thus, $Q \models f \Leftrightarrow P \sqsubseteq Q$.

In our approach, we have the requirements of an application as well as its implementation. We could think of using traditional model checking by means of PROMELA instead of CSP and SPIN [9] instead of FDR [6], but refinement checking is the most natural choice to check whether an implementation conforms to its specification.

Currently, many approaches are using model checking to aid testing with respect at generating test cases. Although refinement checking can also be used in this sense [11], we use it here in its essence. That is, instead of extracting parts os a specification (tests) to execute in an implementation, we simply check whether the implementation model conforms to the specification model.

## 3 The Proposed Approach

The proposed approach (see Figure 1) consists in checking the conformance (refinement) between requirements, say `SysSpec`, and its implementation, say `SysImpl`. As it is usual in the literature [8], we also assume that `SysSpec` is correct but `SysImpl` can have problems (that is, it might not conform to `SysSpec`). To apply refinement checking we need two formal models capturing

the requirements and implementation behaviors. And as formal languages are not widespread in industry, we hide the formalisms to the developers to obtain a better acceptance of our approach.

### 3.1 From Requirements to its Formal Model

To deal with requirements formally but in a hidden way we use a CNL [4]. It is a subset of English that does not introduce ambiguities and syntactic problems. In our Research Project at Motorola, we already have a CNL. It was developed to generate test cases automatically [11]. For example, a simple CNL sentence is "*Select phonebook application*".
Were a requirements document is seen as a set of CNL sentences. A simplified view of the syntax of the CNL we use here is:

```
sentence ::= verb nounPhrase
nounPhrase ::= preposition article modifier noun modifier
                | article modifier noun modifier
                | modifier noun modifier
                | modifier noun
                | noun modifier
                | noun
verb ::= send | call | select ... article ::= the | a | an
preposition ::= from | to ... modifier ::= at least | protected ...
noun ::= phone | folder ...
```

For example, Figure 2 describes part of the feature used in our case study (Section 4). As observed in Figure 2, there are two flows: the main flow and the exception flow; for each flow we have a step description, a system state that complements the steps and the system response. The exception flow is related with some step of the main flow (*From Step:*). The `Step ID` is used to make each step unique in the use case. For instance, `UC_01_1M` is the first step in the `Main Flow` use case. From `UC_01_1M`, we can follow the current main flow. But note that in the `Alternative Flows` use case, the `From Step:` indicates the step `UC_01_1M`. Thus, the flow in the step `UC_01_1A` is an alternative for `UC_01_1M`.

From a CNL use case, we can use the CNL engine to obtain CSP processes. In Figure 3(a), we have the CNL Engine that consists of:

- CNL Sentences: requirements are written in a Microsoft Word document based on a specific template (Figure 2);
- CNL processing: translates each sentence into a CSP event (see Figure 3(b)) and the use case as a whole in a CSP specification.
- CSP alphabet: is formed by all CSP events extracted from the use case;
- CSP process: is the output of the CNL processing and represents the CNL use case described as a CSP specification.

**Main Flow**
From Step: START
To Step: END

| Step Id | User Action | System State | System Response |
|---|---|---|---|
| UC_01_1M | Start IM application. | | The IM application is displayed. |
| UC_01_2M | Log in the IM Server. | | The user is logged in. |
| UC_01_3M | Select a contact. Start a conversation. | | The conversation screen is displayed. There is no IM message in the Conversation Screen. The im editor is empty. |

**Alternative Flows**
From Step: UC_01_1M
To Step: END

| Step Id | User Action | System State | System Response |
|---|---|---|---|
| UC_01_1A | Go to Saved Conversations folder. | | Saved Conversations folder is displayed. |

**Fig. 2.** Use Case written in CNL.

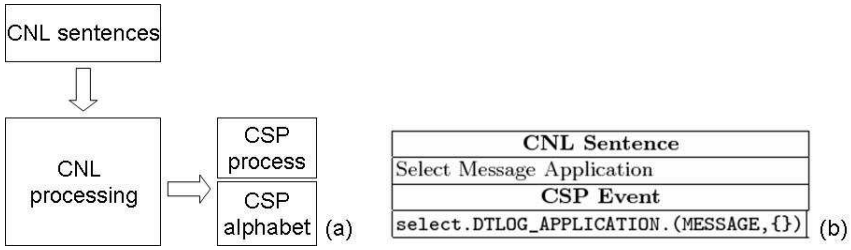


**Fig. 3.** CNL engine.

The CNL engine checks for some inconsistencies. When a grammar error is reported, the user must adjust the corresponding CNL sentence in the use case. The other case occurs when a word is not found in the CNL database. Here the user can change the word for a synonymous already available or store the new word in the database.

### 3.2 From an Implementation to its Model

To obtain a CSP specification of a mobile phone application, we implemented a tool called BxT–Behavior Extractor Tool. It obtains all relevant paths of a mobile phone application by analyzing the source code of a Motorola proprietary test framework called PTF (Phone Test Framework) [5]. This framework gives us access to all mobile phone functions. Furthermore, to produce a model towards a specific objective, BxT was designed to allow the user to guide the model generation. It suffices to enter an objective (for example, an application or a set of menu names) and the model is generated toward such an objective. It is worth

noting that BxT extracts a high-level behavior from the source-code, that is, a functional behavior related mainly to navigational aspects.

Another important aspect of this generation is reusing the module of CNL processing to obtain a compatible alphabet between the requirement and implementation models. This is accomplished by writing sequences of PTF method calls using CNL sentences. For example, the navigation corresponding to Figure 4(b) is expressed in CNL as "Select Message".

These are the main reasons our implementation models are closer to our requirement models as will be discussed in Section 3.3.

Given a mobile phone application, BxT assumes that the initial state is idle (the initial screen of the phone) and using the PTF structure of method calls it gets all paths of a specific application. BxT yields a CSP specification as output but can also output a log of PTF method calls.

Figure 4(a) shows the BxT engine: for each screen, the phone captures all its components using the corresponding PTF of the source code. The components are the screens, buttons, labels, pictures, panels, etc. For each component, a set of properties can be captured like text, state, color, position, etc. BxT captures the components for the navigation.

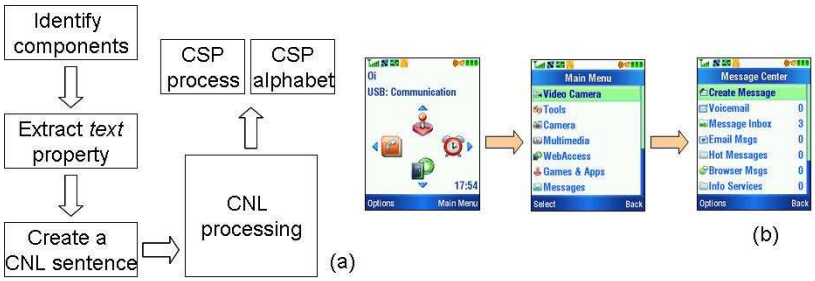


**Fig. 4.** BxT engine and mobile phone screens.

Figure 5(a) corresponds to the use case of Figure 2 as an LTS. And Figure 5(b) to the CSP implementation model of Section 2.1. By a careful observation of these models, we can note that Figure 5(b) almost contains Figure 5(a). The difference between these two models comes from two sources: low-level navigation events (for example, Go to main menu) and extra transitions not allowed by the requirements. The former are natural and related to the artifact used to generate the model: the implementation. The later, however, are problematic because they express behaviors not allowed and thus are considered errors in our approach.

### 3.3 Dealing with different Abstraction levels

When we first met these two models, requirements and implementation, we thought that the behavioral difference between them would be enormous because of the usual semantic gap between them. However, once the BxT tool

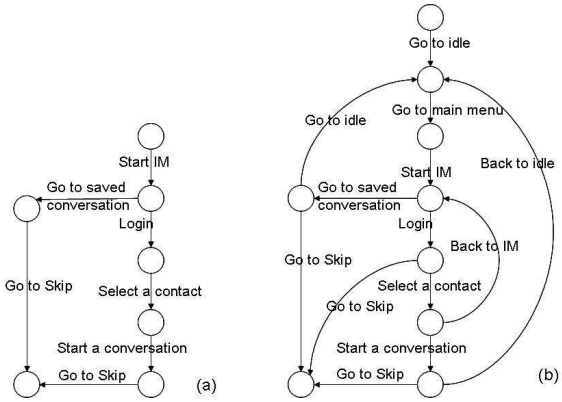


**Fig. 5.** Implementation (a) and specification (b) in LTS representation.

captures the navigational (or system) behavior based on the PTF framework, it only adds details necessary to achieve a certain objective. Thus, while requirements capture more objective oriented behaviors, the implementation behavior complements such a behavior with detailed navigational transitions.

Thus, from the previous observation about Figures 5(a) and 5(b), to do a correct comparison between requirements and implementation, we need to hide all events of `SysImpl` that do not occur in `SysSpec` (the additional navigation details). This corresponds to the following refinement relation:

$$\texttt{SysSpec} \sqsubseteq_T \texttt{SysImpl} \setminus \texttt{Impl\_Details}$$

where

$$\texttt{Impl\_Details} = \alpha\texttt{SysImpl} \setminus \alpha\texttt{SysSpec}$$

### 3.4 Conformance Results

With the previous refinement relation available, we just have to perform a verification using the model checker FDR to obtain two possible situations: everything is all right and thus the implementation satisfies the requirements, or there is a conformance problem and FDR generates a counter-example. This counter-example is very similar to a failing test case. Thus we need to collect all possible counter-examples to adjust the implementation accordingly, following a traditional testing process. As the model checker provides only one counter-example for each verification (the shortest possible path), to discover of them we must propose an iterative procedure to gradually find the next counter-example.

The key idea to collect all counter-examples of the previous refinement relation is based on the properties of two basic elements of CSP: the traces of an external choice and the definition of traces refinement. Suppose that we have the refinement relation $\texttt{SysSpec} \sqsubseteq_T \texttt{SysImpl}$ and such a refinement produces a

trace s as a counter-example. Recall from Section 2.1 that SysSpec $\sqsubseteq_T$ SysImpl means $\mathcal{T}$(SysImpl) $\subseteq$ $\mathcal{T}$(SysSpec) and that $\mathcal{T}$(P [] Q) = $\mathcal{T}$(P) $\cup\mathcal{T}$(Q). Thus, to find the next possible counter-example we update the specification SysSpec with the choice of performing the counter-example s as a CSP process.

To build a process from a given trace we introduce the MakeCSPProcess function which takes as input a trace $< e_1, \ldots, e_n >$ and produces as output the process $e_1$->...->$e_n$-> STOP as follows:

$$\texttt{MakeProcess}(< e_1, \ldots, e_n >) = e_1\texttt{->}\ldots\texttt{->}e_n\texttt{-> STOP}$$

The function MakeProcess produces a process ended in STOP because T(STOP)={<>}, which does not interfere in the refinement relation. Therefore, the refinement SysSpec $\sqsubseteq_T$ SysImpl becomes SysSpec [] F $\sqsubseteq_T$ SysImpl, after the update of SysSpec, where F=MakeProcess(s), for a counter-example s.

**Algorithm 1** Conformance procedure.

```
1.  input: CSP specifications SysSpec, SysImpl, N is the maximum number of
2. counter-examples of interest
3.  output: The set Res of counter-examples found
4.  s ← SysSpec ⊑T SysImpl;
5.  If (s ≠<>)
6.  then {
7.    while ((s ≠<>)and(#Res < N)) do {
8.      Res← Res ∪ <s>;
9.      F ← MakeProcess(s);
10.     SysSpec ← SysSpec [] F;
11.     s ← SysSpec ⊑T SysImpl;
12.   }
13. } else {
14.     println("No counter-example was found.");
15. }
```

From the previous discussion, we propose the Algorithm 1 which yields the set Res of all counter-examples found. That is, if the refinement relation between SysSpec and SysImpl does not hold (SysSpec $\not\sqsubseteq_T$ SysImpl), then a counter-example s is generated and included in the set Res. The counter-example s becomes the process F using the function MakeProcess, and thus it is combined with the process SysSpec using an external choice. The refinement (SysSpec $\sqsubseteq_T$ SysImpl) is repeated until it holds (in this case s = $\langle\,\rangle$).

An observation about Algorithm 1 is that it can run indefinitely. To deal with this problem the parameter N is used to collect only N (#Res<N) counter-examples.

Another important contribution is using our navigation oriented analysis for investigating parts of the mobile device considered critical by experienced test

designers. This is a step forward in the analysis because it can take less time to be accomplished as well as analyze exactly those parts historically known as problematic. In particular, the models presented in this paper followed such a directive using as objective the parts of the mobile phone related to the feature of messaging.

## 4 Case Study: Mobile Application

In this section, we apply our proposed approach in a real case study: a mobile phone application of Motorola.

This case study is simple to save space. However, it is a real scenario in the development of a mobile application. Its simplicity comes from the fact that we have used our navigation-oriented analysis to analyze just those parts of the application related to the feature of messaging.

The first effort in the direction of analyzing the conformance between requirements and implementation of our mobile application is to obtain the formal specifications of both specification and implementation. Thus, applying the following CNL processing engine in the use case description of Figure 2 we obtain the CSP specification.

```
SysSpec = UC_01_1M ; SysSpec
UC_01_1M = steps -> start.DTSTA_APPLICATION.(IM_APPLICATION,{})
-> isstate.DTISS_APPLICATION_STATEVALUE.(IM_APPLICATION,{}).
(DISPLAYED_VALUE,{}) -> (UC_01_2M [] UC_01_1A)
UC_01_2M = steps -> login.DTLOG_APPLICATION.
(IM_SERVER_APPLICATION,{}) -> UC_01_3M
UC_01_3M = steps -> select.DTSEL_LISTITEM.(CONTACT_ITEM,{})-> SKIP
UC_01_1A = steps -> goto.DTGOT_LIST_ITEM.
(SAVED_CONVERSATION_FOLDER,{}) -> SKIP
```

Furthermore, using our BxT engine (Figure 4(a)) in the real mobile device under analysis we obtain the CSP specification shown on Section 2.1.

To generate the requirements model we assume that the model is written correctly and it take few seconds to extract the formal model CSP. In other hand, the model from implementation took about 3 minutes to BxT extract the formal model CSP.

With these two formal models we can make a series of classical (basic) analysis such as deadlock, livelock and non-determinism. These classical concurrent properties are interesting to be analyzed because, in general, the specification as well as the implementation are intended to be free of these problems. If one finds such problems these artifacts can be immediately adjusted (if necessary[2]).

Figure 6(a) shows the results of using the FDR model checker: both requirements and implementation are deadlock and livelock free as well as deterministic.

[2] Sometimes a deadlock situation (a device has broken) is intentional and thus the specification can be maintained as it is.

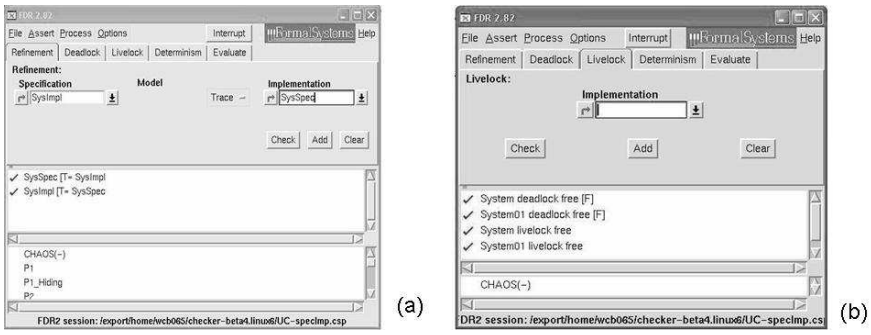


**Fig. 6.** Classical properties and refinement checked.

After the previous basic analysis, we can start our conformance checking by executing Algorithm 1 until the refinement relation between requirements and implementation be satisfied.

When an error is found, this means an implementation bug. By asking FDR for the counter-example, we get the sequence of events presented in Figure 7. This trace shows a sequence of events that is possible to be performed by the implementation model but not by the requirements model. Therefore, this is the implementation bug.

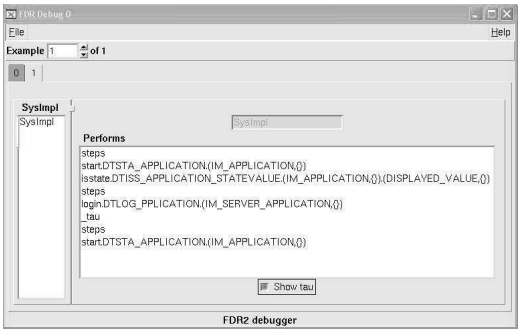


**Fig. 7.** Counter-example.

With this counter-example at hand and following the proposed algorithm, we store this counter-example and use it to obtain a new CSP requirement specification (presented below). A new refinement checking uses this new requirement specification by also following Algorithm 1. As this last refinement checking does not produce any counter-example; the algorithm finishes successfully.

```
SysSpec = UC_01_1M ; SysSpec [] F UC_01_1M = ...
F = steps -> start.DTSTA_APPLICATION. (IM_APPLICATION,{}) ->
isstate. DTISS_APPLICATION_STATEVALUE.(IM_APPLICATION,{}).
(DISPLAYED_VALUE,{}) -> steps -> login.DTLOG_APPLICATION.
(IM_SERVER_APPLICATION,{}) -> steps ->
start.DTSTA_APPLICATION.(IM_APPLICATION,{}) -> STOP
```

Because BxT is based on PTF, the counter-example of Figure 7 could be easily rewritten as a Motorola PTF automatic test case.

An observation, depicted in Figure 6(b), is because the abstraction level between requirements and implementation in our context (black-box view of mobile phone applications) is very close, we indeed have an equivalence relation between requirements and implementation when the implementation conforms to the specification. That is, both refinement relations $\texttt{SysSpec} \sqsubseteq_T \texttt{SysImpl}$ and $\texttt{SysImpl} \sqsubseteq_T \texttt{SysSpec}$ are satisfied ($\texttt{SysSpec} \equiv_T \texttt{SysImpl}$).

To give an idea of the effort spent to perform the approach proposed here, we have used a PC with 1GB and 1, 7MHZ. The requirements in CSP were obtained instantaneously, although we have used about 3 hours to understand the feature and write its CNL. The BxT produced the model in 5 min, and Algorithm 1 took another 50 sec. That is, the effort needed to use our approach is basically associated to understand the feature in question.

## 5 Related Work

The most related works to testing and software model checking are associated to classical model checking [12, 13]. Usually, the properties are described in some temporal logic (like LTL) and the model of the implementation is described in some formal method (like automaton, state machines, *etc*), then each property is checked into the model. Concerning refinement checking, both properties and model are represented in the same language; in our case CSP.

Testing Graphical User Interfaces (GUI) involves dealing with models. There are approaches to generate test cases automatically, select test cases and ripping tools for GUI testing [14]. Thus, our approach is similar to GUI testing but differs in the fact that we extract our models using the proprietary testing framework of Motorola PTF [5]. Thus, we can extract as much information as we need, whereas by extracting from the GUI we are limited by what is offered to interact with.

Doron, Vardi, and Yannakakis [15] proposed an approach combining model checking and testing: black box checking. Their approach tests whether an implementation satisfies some given properties. It assumes that the structure of the implementation is unknown and there is incomplete information to perform model checking directly. Thus, the specification model is adjusted using a learning approach based on Angluin's algorithm. The black box checking approach is characterized as a problem and several algorithms are proposed to solve it. A variation of the black box checking is the Adaptive model checking [12]. This approach considers an incorrect but not obsolete model and explores it by learning the incorrect model to construct the correct model adapting black box testing

and machine learning techniques together. In this context, our approach assumes that the requirements are trustable such that any problem must be related to the implementation. Thus we do not need a learning approach to improve our requirements model.

Considering the specification and implementation models, the approach of Güldali [13] uses both models to do a coverage-driven model checking. In this case, model checking performs automatic testing. In addition, the approach generates test cases through model checking [13]. In both approaches, a motivation to incorporate model checking is to test case generation. Differently from that, we are interested on when we can use model checking instead of testing.

Other approaches combining model checking and software testing have been proposed with different purposes and applicability. Godskesen and Skou propose algorithms to generate test for embedded systems combining the fault model of the system [16]. Vlad, Herv and Thierry presented a combination of verification techniques and conformance testing with a test case generation algorithm [17]. Both works are related to test case generation; they suggest the use of model checking to automate the test case generation.

## 6 Conclusions

Usually, testing just covers a set of critical situations due to time and cost restrictions. Therefore, sometimes the experience of test designers can leave certain situations uncovered and this can result in serious problems.

The approach presented in this paper proposes an alternative way, than testing, to improve quality by only requiring that the test team writes the requirements in a CNL and plug the phone in a computer; the CNL engine and the BxT tool do the rest. After obtaining both models, we simply have to execute Algorithm 1 to obtain the set of test cases that fails. Although this work have focused on mobile phones, any application seen abstractly (by a navigational point of view) can benefit from it.

Because BxT is based on PTF, it represents a PTF extension to generate CSP models to a specific mobile application. Thus, we can easily rewritten the counter-example of Figure 7 as a Motorola PTF automatic teste case. A set of generated test cases based on counter-examples could be applied as a good regression test suite.

Although the literature reveals some works relating model checking with software testing [12, 13], we argue that refinement checking is more appropriate to our context because we simply need a refinement theory to compare requirements with corresponding implementation. The generation of test cases is a natural consequence of the refinement theory, we did not need to focus on test case generation as long as our purpose is simply to check whether the implementation conforms to its corresponding specification. We have used the traces model to perform our analysis because it is directly related to testing. Using the traces model can be seem fragile because the process STOP refines any process P (for any process `P`, `T(STOP)`$= \{\langle\rangle\} \subseteq \mathcal{T}($`P`$)$). Therefore, if the implementation model

has fewer events than the requirements model we cannot detect this. To overcome this we may use other semantic models, such as the failures model.

Although the current version of the BxT tool is already useful, it has some limitations: behaviors that need specific data (for instance, passwords) occur state related information (for example, empty email) and non functional requirements like performance are not captured. Therefore, as future work we intend to improve BxT to deal with such situations.

## References


1. Rajan, A.: Automated Requirements-Based Test Case Generation. SIGSOFT Softw. Eng. Notes **31**(6) (2006) 1–2
2. Drabick, R.D.: Best Practices for the Formal Software Testing Process: A Menu of Testing Tasks. Dorset House Publishing Company (2003)
3. Roscoe, A.W.: The Theory and Practice of Concurrency. Prentice Hall PTR, Upper Saddle River, NJ, USA (1997)
4. Cabral, G., Sampaio, A.: Formal Specification Generation from Requirement Documents. In: Brazilian Symposium on Formal Methods, Natal, Brazil, ENCS (2006)
5. Esipchuk, I., Validov, D.: PTF-based Test Automation for JAVA Applications on Mobile Phones. In: Proceedings of the IEEE 10th International Symposium on Consumer Electronics (ISCE). (2006) 1–3
6. Systems, F.: (FDR2 Model-Checker.) http://www.fsel.com/.
7. Group, S.: (BANDERA Model-Checker.) http://bandera.projects.cis.ksu.edu/.
8. Clarke, E.M., Grumberg, O., Peled, D.: Model Checking. The MIT Press (2000)
9. Barland, I.: (Promela and SPIN Reference.) http://www.spinroot.com/.
10. Andersen, H.R.: Partial Model Checking. In: LICS '95: Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science, Washington, DC, USA, IEEE Computer Society (1995) 398
11. Nogueira, S.: Automatic CSP Test Case Generation Guided by Purposes. (In Portuguese). Master's thesis, Federal University of Pernambuco, Brazil (2006)
12. Groce, A., Peled, D., Yannakakis, M.: Adaptive Model Checking. In: Tools and Algorithms for the Construction and Analysis of Systems : 8th International Conference, TACAS 2002, Held as Part of the Joint European Conference on Theory and Practice of Software, ETAPS 2002, Grenoble, France, LNCS (2002)
13. Güldali, B.: Model Testing Combining Model Checking and Coverage Testing. Master's thesis, University of Paderborn, Germany (2005)
14. Xie, Q., Memon, A.M.: Designing and Comparing Automated Test Oracles for GUI-Based Software Applications. ACM Transactions on Software Engineering and Methodology **16**(1) (2007)
15. Peled, D., Vardi, M.Y., Yannakakis, M.: Black Box Checking. J. Autom. Lang. Comb. **7**(2) (2001) 225–246
16. C. Godskesen, B.N., Skou, A.: Connectivity Testing through Model-Checking. In: International Conference on Formal Techniques for Networked and Distributed Systems (FORTE), Madrid, Spain (2004)
17. Vlad, R., Herv, M., Thierry, J.: Automatic Verification and Conformance Testing for Validating Safety Properties of Reactive Systems. In Fitzgerald, J., Tarlecki, A., Hayes, I., eds.: Formal Methods 2005 (FM05). LNCS, Springer (2005)