Table of Contents

Table of Contents ........................................................................................................... i

Preface ............................................................................................................................ ii

Luca Aceto, Adrian Francalanza and Anna Ingolfsdottir

Program Repair by Stepwise Correctness Enhancement ........................................... 1

Nafi Diallo, Wided Ghardallou and Ali Mili


Vignir Gudmundsson, Mikael Lindvall, Luca Aceto, Johann Bergthorsson and

Dharmalingam Ganesan

Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates ............. 31

Sönke Holthusen, Sophie Quinton, Ina Schaefer, Johannes Schlatow and Martin Wegner

Monitoring Assumptions in Assume-Guarantee Contracts ........................................... 46

Oleg Sokolsky, Teng Zhang, Insup Lee and Michael McDougall

Preliminary Results Towards Contract Monitorability ................................................... 54

Annalizz Vella and Adrian Francalanza
Preface

Luca Aceto
ICE-TCS, School of Computer Science, Reykjavik University
Reykjavik, Iceland
luca@ru.is

Adrian Francalanza
Computer Science, Faculty of Information and Communications Technology
University of Malta, Malta
adrian.francalanza@um.edu.mt

Anna Ingolfsdottir
ICE-TCS, School of Computer Science, Reykjavik University
Reykjavik, Iceland
annai@ru.is

This volume contains the proceedings of PrePost 2016 (the First International Workshop on Pre- and Post-Deployment Verification Techniques), taking place on 4th June 2016 in Reykjavik, Iceland, as a satellite event of the 12th International Conference on integrated Formal Methods (iFM 2016).

The PrePost (Pre- and Post-Deployment Verification Techniques) workshop aimed at bringing together researchers working in the field of computer-aided validation and verification to discuss the connections and interplay between pre- and post-deployment verification techniques. Examples of the topics covered by the workshop are the relationships between classic model checking and testing on the one hand and runtime verification and statistical model checking on the other, and between type systems that may be checked either statically or dynamically through techniques such as runtime monitoring.

We are honoured to have the following invited speakers at the workshop:

- Dino Distefano (Facebook and Queen Mary, University of London, UK) and
- Kim G. Larsen (Aalborg University).

We are most grateful to our programme committee:

- Wolfgang Ahrendt (Chalmers University of Technology, Sweden)
- Maria Christakis (Microsoft Research Redmond, USA)
- Matteo Cimini (Indiana University Bloomington, USA)
- Rance Cleaveland (University of Maryland, USA)
- Christoph Csallner (University of Texas at Arlington, USA)
- Ylies Falcone (Univ. Grenoble Alpes, Inria, Laboratoire d’Informatique de Grenoble, France)
- Klaus Havelund (Jet Propulsion Laboratory, California Institute of Technology, USA)
- Ivan Lanese (University of Bologna, Italy)
- Julien Lange (Imperial College, UK)

*The work of the authors was partly supported by the project “TheoFoMon: Theoretical Foundations for Monitorability” (grant number: 163406-051) of the Icelandic Research Fund.
• Martin Leucker (University of Lübeck, Germany)
• Mohammadreza Mousavi (Halmstad University, Sweden)
• Jorge A. Pérez (University of Groningen, The Netherlands).
We also thank the iFM 2016 organizers and the EPTCS staff.
Program Repair by Stepwise Correctness Enhancement

Nafi Diallo  
CCS, NJIT, Newark NJ USA  
ncd8@njit.edu

Wided Ghardallou  
FST, UTM, Tunis, Tunisia  
widied.ghardallou@gmail.com

Ali Mili  
CCS, NJIT, Newark, NJ USA  
mili@njit.edu

Relative correctness is the property of a program to be more-correct than another with respect to a given specification. Whereas the traditional definition of (absolute) correctness divides candidate program into two classes (correct, and incorrect), relative correctness arranges candidate programs on the richer structure of a partial ordering. In other venues we discuss the impact of relative correctness on program derivation, and on program verification. In this paper, we discuss the impact of relative correctness on program testing; specifically, we argue that when we remove a fault from a program, we ought to test the new program for relative correctness over the old program, rather than for absolute correctness. We present analytical arguments to support our position, as well as an empirical argument in the form of a small program whose faults are removed in a stepwise manner as its relative correctness rises with each fault removal until we obtain a correct program.

Keywords
Program correctness, Relative correctness, Absolute correctness, Program repair.

1 Relative Correctness and Quality Assurance Methods

Relative correctness is the property of a program to be more-correct than another with respect to a given specification. Intuitively, \( P' \) is more-correct than \( P \) with respect to specification \( R \) if and only if \( P' \) obeys \( R \) more often (for a larger set of inputs) than \( P \), and violates \( R \) less egregiously (in fewer ways) than \( P \).

Traditionally, we distinguish between two categories of candidate programs for a given specification: correct programs, and incorrect programs; but the introduction of relative correctness enables us to generalize this binary classification into a richer structure that ranks all candidate programs by means of a partial ordering whose maximal elements are (absolutely) correct.

Also, in our quest for enhancing program quality, we have traditionally used static analysis methods and dynamic testing methods for distinct purposes:

- Program verification methods are applied to correct programs to ascertain their correctness; they are of little use when applied to incorrect programs, because even when a proof fails, we cannot conclude that the program is incorrect (the proof may have failed because the documentation of the program in terms of intermediate assertions and invariant assertions is inadequate).

- Program testing methods are applied to incorrect programs to expose their faults and remove them; but they are of little use when applied to correct programs, since they cannot be used to prove the absence of faults.

Here again, we argue that relative correctness can act as a disruptive concept, since it blurs this neat separation of duties. In [7], we present a relative correctness-based static analysis method that enables us to locate and remove a fault from a program, and prove that the fault has been removed — all without testing. This technique, which we call debugging without testing, shows that we can apply static analysis...
to an incorrect program to prove that, although it may be incorrect, it is still more correct than another. Given that there are orders of magnitude more incorrect programs than there are correct programs, the pursuit of this idea may expand the scope of static analysis methods.

In [5], we discuss how relative correctness can be used in the derivation of a correct program from a specification. Whereas traditional programming calculi derive programs from specifications by successive refinement-based correctness-preserving transformations starting from the specification, we show that we can derive a program by successive correctness-enhancing transformations (using relative correctness) starting from the trivial program abort. We refer to this technique as *programming without refining* [5].

In this paper, we explore the use of relative correctness in program repair. Specifically, we discuss how to perform program repair when we test candidate mutants for relative correctness rather than absolute correctness. We are not offering a viable, validated, empirically supported solution; rather, we are merely analyzing current practice, discussing why we believe a relative-correctness-based approach may offer better outcomes, and supporting our case with analytical arguments as well as a simple illustrative example.

In section 3 we define relative correctness and explore its main properties; since our definitions and discussions rely on relational calculi, we devote section 2 to a brief discussion of relational concepts. In section 4 we critique the current practice of program repair, which is based on a test of absolute correctness, and argue, on the basis of analytical arguments, that using a test of relative (rather than absolute) correctness leads to better outcomes. We complement the analytical argument of section 4 by an empirical illustration in section 5 in the form of a faulty program, which we repair in a stepwise manner by removing its faults one by one, making it increasingly more-correct until it becomes absolutely correct. We summarize and assess our findings in section 6, and we briefly sketch directions for future research.

## 2 Relational Mathematics

We assume the reader familiar with relational algebra, and we generally adhere to the definitions of [18][2]. Dealing with programs, we represent sets using a programming-like notation, by introducing variable names and associated data types. If, e.g. we define set $S$ by the variable declarations

$$ x : X; y : Y; z : Z, $$

then $S$ is the Cartesian product $X \times Y \times Z$. Elements of $S$ are denoted by $s$, and are triplets of elements of $X$, $Y$, and $Z$. Given $s$ in $S$, we represent its $X$-component (resp. $Y$-component, $Z$-component) by (resp.) $x(s)$, $y(s)$, $z(s)$. When no risk of ambiguity exists, we may write $x$ to represent $x(s)$, and $x'$ to represent $x(s')$. A relation on $S$ is a subset of the Cartesian product $S \times S$. Special relations on $S$ include the *universal relation* $L = S \times S$, the *identity relation* $\mathit{I} = \{(s,s') | s' = s\}$, and the *empty relation* $\phi = \{\}$. Operations on relations (say, $R$ and $R'$) include the set theoretic operations of union ($R \cup R'$), intersection ($R \cap R'$), difference ($R \setminus R'$) and complement ($\overline{R}$). They also include the *relational product*, denoted by ($R \circ R'$), or ($RR'$, for short) and defined by:

$$ RR' = \{(s,s'') | \exists s'' : (s,s'') \in R \wedge (s'', s') \in R'\}. $$

The *power* of relation $R$ is denoted by $R^n$, for a natural number $n$, and defined by $R^0 = I$, and for $n > 0$, $R^n = R \circ R^{n-1}$. The *reflexive transitive closure* of relation $R$ is denoted by $R^*$ and defined by $R^* = \{(s,s') | \exists n \geq 0 : (s,s') \in R^n\}$. The *converse* of relation $R$ is the relation denoted by $\hat{R}$ and defined by $\hat{R} = \{(s',s) | (s,s') \in R\}$. The *domain* of a relation $R$ is defined as the set $\text{dom}(R) = \{s | \exists s' : (s,s') \in R\}$, and the *range* of relation $R$ is defined as the domain of $\hat{R}$. Note that given a relation $R$, the product of
Given two relations $R$ and $R'$, we say that $R'$ refines $R$ (abbrev: $R' \sqsupseteq R$) if and only if: $RL \cap R'L \cap (R \cup R') = R$.

Intuitively, a relation $R'$ refines a relation $R$ if and only if it has a larger domain and assigns fewer images than $R$ to elements of the domain of $R$. See Figure 1, where $R''$ refines $R'$, which in turns refines $R$.

### 3 Absolute Correctness and Relative Correctness

#### 3.1 Program Functions

If a program $p$ manipulates variables, say $x : X$ and $y : Y$, we say that set $S = X \times Y$ is the *space* of $p$ and we refer to elements of $S$ as *states* of $p$. Given a program $p$ on space $S$, we denote by $[p]$ the function that $p$ defines on its space, i.e.

$$[p] = \{(s, s') | \text{if program } p \text{ executes on state } s \text{ then it terminates in state } s'\}.$$  

We represent programs by means of C-like programming constructs, which we present below along with their semantic definitions:

- **Abort**: $[\text{abort}] \equiv \phi$.
- **Skip**: $[\text{skip}] \equiv I$. 
- **Assignment**: $[s = E(s)] \equiv \{(s, s') | s \in \delta(E) \wedge s' = E(s)\}$, where $\delta(E)$ is the set of states for which expression $E$ can be evaluated.
- **Sequence**: $[p_1; p_2] \equiv [p_1] \circ [p_2]$. 
- **Conditional**: $[\text{if } (t) \{p\}] \equiv T \cap [p] \cup \overline{T} \cap I$, where $T$ is the relation defined as: $T = \{(s, s') | t(s)\}$. 

![Figure 1: Refinement: $R' \sqsupseteq R$, $R'' \sqsupseteq R'$](image)
4 Program Repair by Correctness Enhancement

- **Alternation:** $[\text{if } (t) \{ p \} \text{ else } \{ q \}] \equiv T \cap [p] \cup \overline{T} \cap [q]$, where $T$ is defined as above.
- **Iteration:** $[\text{while } (t) \{ b \}] \equiv (T \cap [b])^* \cap \overline{T}$, where $T$ is defined as above.
- **Block:** $[\{ x : X; p \}] \equiv \{(s,s') \mid \exists x \in X : (\langle s, x \rangle, \langle s', x' \rangle) \in [p]\}$.

We will usually use upper case $P$ as a shorthand for $[p]$. By abuse of notation, we may refer to a program and its function by the same name.

### 3.2 Absolute Correctness

**Definition 2** Let $p$ be a program on space $S$ and let $R$ be a specification on $S$. We say that program $p$ is correct with respect to specification $R$ if and only if $P$ refines $R$. We say that program $p$ is partially correct with respect to specification $R$ if and only if $P$ refines $R \cap PL$.

This definition is consistent with traditional definitions of partial and total correctness [10][11][6]. Whenever we want to contrast correctness with partial correctness, we may refer to it as total correctness. The following proposition, due to [17], gives a simple characterization of correctness for deterministic programs.

**Proposition 1** Program $p$ is correct with respect to specification $R$ if and only if $(R \cap P)L = RL$.

By construction, $(R \cap P)L$ is a subset of $RL$, and correct programs are those that reach the maximum of $(R \cap P)L$, which is $RL$.

### 3.3 Relative Correctness: Deterministic Programs

**Definition 3** Let $R$ be a specification on space $S$ and let $p$ and $p'$ be two deterministic programs on space $S$ whose functions are respectively $P$ and $P'$. We say that program $p'$ is more-correct than program $p$ with respect to specification $R$ (abbrev: $P' \succeq_R P$) if and only if: $(R \cap P')L \supseteq (R \cap P)L$. Also, we say that program $p'$ is strictly more-correct than program $p$ with respect to specification $R$ (abbrev: $P' \supseteq_R P$) if and only if $(R \cap P')L \supset (R \cap P)L$.

Interpretation: $(R \cap P)L$ represents (in relational form) the set of initial states on which the behavior of $P$ satisfies specification $R$. We refer to this set as the competence domain of program $P$ with respect to specification $R$. For deterministic programs $p$ and $p'$, relative correctness of $p'$ over $p$ with respect to specification $R$ simply means that $p'$ has a larger competence domain than $p$. Whenever we want to contrast correctness with relative correctness, we refer to it as absolute correctness. Note that when we say more-correct we really mean more-correct or as-correct-as. Note also that program $p'$ may be more-correct than program $p$ without duplicating the behavior of $p$ over the competence domain of $p$; see Figure 2. In the example shown in this figure, we have:

- $(R \cap P)L = \{1,2,3,4\} \times S$,
- $(R \cap P')L = \{1,2,3,4,5\} \times S$,

where $S = \{0,1,2,3,4,5,6\}$. Hence $p'$ is more-correct than $p$ with respect to $R$.

In order to highlight the contrast between relative correctness (as a partial ordering) and absolute correctness (as a binary attribute), we consider the specification $R$ on space $S = \{a,b,c,d,e\}$

$R = \{(a,a),(a,b),(a,c),(b,b),(b,c),(b,d),(c,c),(c,d),(c,e)\}$,

and we consider the following programs, along with their competence domains:

- $P_0 = \{(a,d),(b,a)\}$. $CD_0 = \{\}$.
- $P_1 = \{(a,b),(b,e)\}$. $CD_1 = \{a\}$.
Let $R$ be the following specification on $R$: $R = \{(s, s')|x' \geq 3x^2\}$. Let $p$ and $p'$ be the following programs:

Figure 3 shows how these programs are ordered by relative correctness with respect to specification on $R$.

3.4 Relative Correctness: Non-Deterministic Programs

We let $S$ be the space defined by non-zero natural variables $x$ and $y$ and we let $R$ be the following specification on $R$: $R = \{(s, s')|x' \geq 3x^2\}$. Let $p$ and $p'$ be the following programs:
p: \{x=13\times x; \ y=y+2\times x;\},  
p': \{x=19\times x; \ y=y+7\times x;\}.
The functions of these programs are:
\[ P = \{(s,s')| x' = 13 \times x \land y' = y + 26 \times x\} \]
\[ P' = \{(s,s')| x' = 19 \times x \land y' = y + 133 \times x\}. \]

We are interested to analyze the relative correctness of \( p \) and \( p' \) with respect to \( R \); to this effect, according to definition 3, we must analyze the functions of \( p \) and \( p' \). But the effect of these programs on variable \( y \) could not possibly be relevant to this analysis since \( R \) does not refer to \( y \). Hence it ought to be possible for us to analyze the relative correctness of \( p \) and \( p' \) with respect to \( R \) by considering the effect of \( p \) and \( p' \) on \( x \) alone; this yields the following relations:
\[ \pi = \{(s,s')| x' = 13 \times x\}, \]
\[ \pi' = \{(s,s')| x' = 19 \times x\}. \]

Yet, we cannot apply Definition 3 to \( \pi \) and \( \pi' \) because they are not deterministic (since they fail to specify a final value for variable \( y \)). In [4] we present a definition of relative correctness that generalizes Definition 3 and applies to (possibly) non-deterministic programs; referring to the definition given in [4], we find (by inspecting \( \pi \) and \( \pi' \) rather than \( P \) and \( P' \)) that \( p' \) is more-correct than \( p \) with respect to \( R \). Indeed, the competence domain of program \( p \) with respect to \( R \) (i.e. the set of initial states for which \( p \) behaves according to \( R \)) is characterized by the equation \( 13 \times x \geq 3 \times x^2 \), which is equivalent (since \( x \) is a natural variable) to \( x \leq 4 \). Likewise, we find that the competence domain of \( p' \) is characterized by the equation \( x \leq 6 \).

4 Program Repair by Relative Correctness

4.1 Faults and Fault Removal

Now that we know what it means for a program to be more correct than another, we are in a position to define what is a fault, and under what condition we can say that we have removed a fault. Any definition of a fault assumes, implicitly, some level of granularity at which we want to define faults; a typical level of granularity for C-like languages is the single assignment statement, while finer grained features include expressions or operators within expressions. We use the term feature to refer to any program part, or set of program parts, at the selected level of granularity and we present the following definition, due to [16].

**Definition 4** Given a program \( p \) on space \( S \) and a specification \( R \) on \( S \), we say that a feature \( f \) of \( p \) is a fault in \( p \) with respect to \( R \) if and only if there exists a feature \( f' \) such that program \( p' \) obtained from \( p \) when we replace \( f \) by \( f' \) is strictly more-correct than \( p \) with respect to \( R \).

*When such an \( f' \) is found, the pair \((f, f')\) is called a fault removal in \( p \) with respect to \( R \).*

As an illustrative example, we consider the following program \( p \) on space \( S \) defined by variables \( a, x \) and \( i \) declared therein:

\[
p: \text{int main () \ {int a[N+1]; int x=0;  
 \indent int i=0; while (i<N) \{x=x+a[i]; \ i=i+1;\}}}
\]

and we consider the specification \( R \) defined by:
\[ R = \{(s,s')| x' = \sum_{i=1}^{N} a[i]\}. \]
The function of program \( p \) is:
\[ P = \{(s,s')| a \land i' = N \land x' = \sum_{i=0}^{N-1} a[i]\}. \]
The competence domain of $p$ with respect to $R$ is:

$$(R \cap P)L = \{(s, s')|a[0] = a[N]\},$$

which makes sense, since this is the condition under which what the program does (the sum of $a$ from 0 to $N - 1$) coincides with what the specification mandates (the sum from 1 to $N$). Because the competence domain of $p$ is distinct from $\text{dom}(R) = S$, this program is incorrect. At the level of granularity of assignment statements and logical expressions, we see two faults in program $p$ with respect to specification $R$:

- The fault made up of the aggregate of statements $f1 = (i=0, (i<N))$; the substitution $f1' = (i=1, (i<N+1))$ constitutes a fault removal for $f1$.
- The fault $f2 = (x=x+a[i])$; substitution of $f2$ by $f2' = (x=x+a[i+1])$ constitutes a fault removal for $f2$.

Note that $(i=0)$ alone is not a fault in $p$, nor is $(i<N)$ as they admit no substitution that would make the program more-correct. If we let $p1'$ be the program obtained from $p$ by substituting $f1$ by $f1'$, then we find that $f2$ is not a fault in $p'$, even though it is a fault in $p$; the same goes for program $p2'$ obtained by substituting $f2'$ for $f2$. If we substitute $f1$ by $f1'$ and $f2$ by $f2'$ we find two faults again, namely $f1'$ and $f2'$. See Figure 4, where we label each transformation by the corresponding substitution, and we let $s1$ be the substitution $(f1, f1')$ and $s2$ be the substitution $(f2, f2')$. Even though $p$ has two faults, it is one fault removal away from being correct; we say that it has a fault density of 2 and a fault depth of 1.

### 4.2 Testing for Relative Correctness

Now that we have defined what is a fault removal, we address the question: how do we ascertain that we have removed a fault? As is customary for matters pertaining to software products, we can do so in one of two ways:

- Either through a static analysis of the products’ ($p$ and $p'$) source code; this is discussed in [7].
- Or through execution and monitoring of the products in question; this is the subject of this paper.

This raises the question: how do we test a program $p'$ for relative correctness over some program $p$ with respect to specification $R$, and how is that different from testing program $p'$ for absolute correctness with respect to $R$? For the sake of simplicity, we address this question in the context of deterministic programs, and we argue that testing a program $p'$ for relative correctness over some program $p$ with respect to some specification $R$ differs from testing program $p'$ for absolute correctness with respect to $R$ in three important ways:

- **Test Data Selection.** The problem of test data selection can be formulated in the following generic terms: Given a large or infinite input space, say $D$, find a representative subset $T$ of $D$ such that
Program Repair by Correctness Enhancement

analysis of the behavior of candidate programs on \( T \) enables us to infer claims about their behavior on \( D \). Regardless of what selection criterion is adopted to derive \( T \) from \( D \), testing for relative correctness differs from testing for absolute correctness in a fundamental way: for absolute correctness, the input space \( D \) we are trying to approximate is \( D = \text{dom}(R) \), whereas for relative correctness the input space is \( D = \text{dom}(R \cap P) \), i.e. the competence domain of \( p \). Indeed, to prove that \( p' \) is more-correct than \( p \) with respect to \( R \), we must prove that \( p' \) runs successfully for all elements of the competence domain \( D \) of \( p \), which we do by checking that \( p' \) runs successfully for all elements of \( T \) (an approximation of \( D \)).

**Oracle Design.** Let \( \Omega(s,s') \) be the oracle for absolute correctness derived from specification \( R \). Then the oracle for relative correctness of program \( p' \) over program \( p \), which we denote by \( \omega(s,s') \), must ensure that program \( p' \) satisfies \( \Omega(s,s') \) for all \( s \) in the competence domain of \( p \) with respect to \( R \). We write it as:

\[
\omega(s,s') \equiv (\Omega(s,P(s)) \Rightarrow \Omega(s,s')).
\]

As for oracle \( \Omega(s,s') \) it must be derived from specification \( R \) according to the following formula:

\[
\Omega(s,s') \equiv (s \in \text{dom}(R) \Rightarrow (s,s') \in R).
\]

Indeed, we do not want a test to fail on some input \( s \) outside the domain of \( R \), as candidate programs are only responsible for their behavior on \( \text{dom}(R) \). Hence the condition \((s,s') \in R\) is checked only for \( s \) in \( \text{dom}(R) \); for \( s \) outside the domain of \( R \), the test is (vacuously) considered successful.

**Test Coverage Assessment.** When we test a program \( p' \) for absolute correctness with respect to some specification \( R \) using a test data of size \( N \), we gain a level of confidence in the correctness of \( p' \), to an extent that is commensurate with \( N \). On the other hand, when we test a program \( p' \) for relative correctness over a program \( p \) with respect to some specification \( R \) on a test data of size \( N \), \( N \) does not tell the whole story: We also need to know whether \( p' \) behaves better than \( p \) because \( p \) fails often or because \( p' \) succeeds often. Hence we may need to quantify the outcome of the experiment by means of three variables: \( N_0 \), the number of test cases when both \( p \) and \( p' \) succeed; \( N_1 \), the number of test cases when \( p \) fails and \( p' \) succeeds; and \( N_2 \), the number of test cases when both fail. While \( N = N_0 + N_1 + N_2 \) tells us to what extent \( p' \) is better than \( p \) (i.e. to what extent we can be confident that \( p' \) is more-correct than \( p \)), the partition of \( N \) into \( N_0 \), \( N_1 \) and \( N_2 \) tells us whether \( p' \) is better than \( p \) because \( p' \) succeeds often or because \( p \) fails often. See Figure 5.

### 4.3 Program Repair with Absolute Correctness

Most techniques for program repair [1][19][8][3][9][20] proceed by applying transformations on an original faulty program. These transformations may be macro-transformations (including multi-site program modifications), or micro-modifications (intra-statement) using mutation operators such as those provided by the muJava [15] program mutation tool. Two main approaches exist towards assessing the suitability of the generated transformations: test-based techniques [1][19][3][9] (which use the successful execution of the candidate program on a test suite as the acceptance criterion) or specification-based techniques [8][20] (which use a specification and some sort of constraint-solving to determine if the new code complies with the specification). In both cases, mutants are selected on the basis of an analysis of their absolute correctness with respect to the specification at hand (embodied in the oracle in the case of testing).
We argue that selecting mutants on the basis of their absolute correctness is flawed, because when we remove a fault from the original program, we have no reason to believe that the new program is correct, unless we assume that the fault that we have just removed is the program’s last fault. Instead, the best we can hope for when we generate a mutant from a base program is that the mutant is more-correct than the base program with respect to the specification at hand; consequently, we should be testing mutants for relative correctness rather than absolute correctness. Specifically, we argue that when mutants are evaluated on the basis of their absolute correctness on a sample test data $T$, both the decision to retain successful mutants and the decision to reject unsuccessful mutants, are wrong:

- As Figure 6(a) shows (if $CD$ is the competence domain of the original program and $CD'$ is the competence domain of the mutant), a mutant may pass the test $T$ (since $T \subseteq CD'$) yet not be more-correct than the original (since $CD$ is not a subset of $CD'$).

- As Figure 6 (b) shows, a mutant may fail the test $T$ (since $T$ is not a subset of $CD'$) and yet still be more-correct than the original (since $CD \subseteq CD'$).

As a result, neither the precision nor the recall of the selection algorithm is assured.
4.4 Program Repair with Relative Correctness

In light of the foregoing discussion, we argue that if a fault removal is expected to make a program more-correct than the original (vs absolutely correct) then it is only fair that it be tested for relative correctness rather than absolute correctness. In this way, if a program has several faults, we can remove them one at a time in a stepwise manner. To do so, we adopt the test data selection criterion and the oracle design discussed in section 4.2. As we recall, in order to test a program for relative correctness over a program \( P \) with respect to a specification \( R \), we have to select test data in the competence domain of \( P \) with respect to \( R \) (\( CD \) in Figure 7) rather than to select it in \( \text{dom}(R) \). As we can see from Figure 7(b), this ensures perfect recall since all programs that are relatively correct with respect to \( P \) are selected. As for retrieval precision, it depends on the quality of the test data, but as Figure 7(a) shows, we can still select programs that are not more-correct than \( P \), if \( T \) is not adequately distributed over \( CD \); hence precision remains an issue.

5 Illustration: Fermat Decomposition

5.1 Experimental Setup

To illustrate the distinction between program repair by absolute correctness and by relative correctness, we consider a program that performs the Fermat decomposition of a natural number, in which we introduce three changes. The space of a Fermat decomposition is defined by three natural variables, \( n, x \) and \( y \) and the specification is defined as follows:

\[
R = \{(s,s')|(n \mod 2 = 1) \lor (n \mod 4 = 0) \land n = x'^2 - y'^2\}.
\]

A correct Fermat program (which we call \( p' \)) is:

```c
void fermatFactorization() {
    int n, x, y; // input/output variables
    int r; // work variable
    x = 0; r = 0;
    while (r < n) {r = r + 2 * x + 1; x = x + 1; }
    while (r > n) {int rsave; y = 0; rsave = r;
```
while (r > n) {r = r - 2 * y - 1; y = y + 1; }
if (r < n) {r = rsave + 2 * x + 1; x = x + 1; }}

The three changes we introduce in this program are shown below; we do not call them faults yet because
we do not know whether they meet our definition of a fault (Definition 4). A given number of changes
(re: three in this case) can lead to fewer faults (if some changes cancel each other, or if one or more
changes have no effect on the function of the program); also, a given number of changes (three in this
case) can also lead to a larger number of faults (the same change can be remedied either by reversing the
change or by altering the program elsewhere to cancel the change). We revisit this discussion in the next
section.

We let $p$ be the program obtained after introducing the changes to $p'$:

```c
void basep(int& n, int& x, int& y) {
  int r; x = 0; r = 0;
  while (r < n) {r = r + 2 * x - 1; /*change in r*/ x =x+1;}
  while (r > n) {int rsave; rsave = r; y = 0;
    while (r > n) {r =r-2*y+1; /*change in r*/ y =y+1;}
    if (r < n) {r =rsave+2*x-1; /*change in r*/ x =x+1;}}}
```

Most program repair methods proceed by generating mutants of the base program and testing them for
absolute correctness; all we are advocating in this paper is that instead of testing mutants for absolute
correctness, we ought to test them for relative correctness. To illustrate our approach, we generate
mutants of program $p$, test them for absolute correctness, and show that none of them are (absolutely)
correct. If absolute correctness were our only criterion, then this would be the (unsuccessful) end of the
experiment. But we find that while none of the mutants are absolutely correct, some are strictly more-
correct than $p$; hence the transition from $p$ to these mutants represents a fault removal (by Definition 4).
If we take these mutants as our base programs and apply the mutation generator to them, then test them
for strict relative correctness, we can iteratively remove the faults of the program in a stepwise manner,
climbing the relative correctness ordering until we reach a (absolutely) correct program.

Specifically, we start from program $p$ and apply muJava to generate mutants using the single mutation
option with the AORB operator (Arithmetic Operator Replacement, Binary). Whenever a set of mutants
are generated, we subject them to three tests:

- A test for absolute correctness, using the oracle $\Omega(s,s')$.
- A test for relative correctness, using the oracle $\omega(s,s')$.
- A test for strict relative correctness, which in addition to relative correctness checks the presence
  of at least one state in the competence domain of the mutant that is not in the competence domain
  of the base program.

The mutants that are found to be strictly more-correct than the base program are used as new base
programs, and the process is iterated again until at least one mutant is found to be absolutely correct;
we select this mutant as the repaired version of the original program $p$. The main iteration of the test
driver is given below. All the details of our experiment are posted online at https://selab.njit.
edu/programrepair/.

```c
int main ()
{for (int mutant =1; mutant<= nbmutants; mutant++)
  // test mutant vs spec. R for abs and rel correctness
  bool cumulabs=true; bool cumulrel=true; bool cumulstrict=false;
```
while (moretestdata)
{int n,x,y; int initn,initx,inity; // initial, final states
    bool abstcor, relcor, strict;
    initn=td[tdi]; tdi++; // getting test data
    n=initn; x=initx; y=inity; // saving initial state
    callmutant(mutant, n, x, y);
    abstcor = absoracle(initn, initx, inity, n, x, y);
    cumulabs = cumulabs && abstcor;
    n=initn; x=initx; y=inity; // re-initializing
    basep(n, x, y);
    relcor = ! absoracle(initn, initx, inity, n, x, y) || abstcor;
    strict = ! absoracle(initn, initx, inity, n, x, y) && abstcor;
    cumulrel = cumulrel && relcor;
    cumulstrict = cumulstrict || strict;
}

bool R (int initn, int initx, int inity, int n, int x, int y)
{ return ((initn%2==1) || (initn%4==0)) && (initn==x*x-y*y); }

bool domR (int initn, int initx, int inity)
{ return ((initn%2==1) || (initn%4==0)); }

bool absoracle (int initn,int initx,int inity,int n,int x,int y)
{ return (! (domR(initn, initx, inity)) || R(initn, initx, inity, n, x, y)); }

The main program includes two nested loops; the outer loop iterates over mutants and the inner loop iterates over test data. For each mutant and test datum, we execute the mutant and the base program on the test datum and test the mutant for absolute correctness (abscor), relative correctness (relcor) and strict relative correctness (strict); these boolean results are cumulated for each mutant in variables cumulabs, cumulrel and cumulstrict, and are used to diagnose the mutant. As for the Boolean functions R, domR and absoracle, they stem readily from the definition of R and from the oracle definitions given in section 4.2.

5.2 Experimental Results

Starting with program \( p \), we apply muJava repeatedly to generate mutants, taking mutants which are found to be strictly more-correct as base programs and repeating until we generate a correct program. This proceeds as follows:

- When muJava is executed on program \( p \), it produces 48 mutants, of which two \( (m12 \) and \( m44 \)) are found to be strictly more-correct than \( p \), and none are found to be absolutely correct with respect to \( R \); we pursue the analysis of \( m12 \) and \( m44 \).

- Analysis of \( m44 \). When we apply muJava to \( m44 \), we find 48 mutants, none of them prove to be absolutely correct, nor relatively correct, nor strictly relatively correct.

- Analysis of \( m12 \). We find by inspection that \( m12 \) reverses one of the modifications we had applied to \( p' \) to find \( p \); since \( m12 \) is strictly more-correct than \( p \) with respect to \( R \), we conclude that the feature in question was in fact a fault in \( p \) with respect to \( R \). When we apply muJava to \( m12 \), it generates 48 mutants, three of which prove to be strictly more-correct than \( m12 \): we name them...
$m_{12.19, 28.44}$ and $m_{12.20, 28.44}$. All the other mutants are found to be neither absolutely correct with respect to $R$, nor more correct than $m_{12}$.

- **Analysis of $m_{12.19}$**. When we apply muJava to $m_{12.19}$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m_{12.19}$, but one ($m_{12.19.24}$) proves to be identical to $m_{12.20}$ and is more-correct than (but not strictly more-correct than, hence as correct as) $m_{12.19}$.

- **Analysis of $m_{12.20}$**. When we apply muJava to $m_{12.20}$, it generates 48 mutants, none of which is found to be absolutely correct nor strictly more-correct than $m_{12.20}$, but one ($m_{12.20.24}$) proves to be identical to $m_{12.19}$ and is more-correct than (but not strictly more-correct than, hence as correct as) $m_{12.20}$.

- **Analysis of $m_{12.28}$**. We find by inspection that $m_{12.28}$ reverses a second modification we had applied to $p'$ to obtain $p$; since $m_{12.28}$ is strictly more-correct than $m_{12}$, this feature is a fault in $m_{12}$; whether it is a fault in $p$ we have not checked, as we have not compared $m_{12.28}$ and $p$ for relative correctness. When we apply muJava to $m_{12.28}$, we find a single mutant, namely $m_{12.28.44}$ that is absolutely correct with respect to $R$, more-correct than $m_{12.28}$ with respect to $R$, and strictly more-correct than $m_{12.28}$ with respect to $R$.

  » **Analysis of $m_{12.28.44}$**. We find by inspection that $m_{12.28.44}$ is nothing but the original Fermat decomposition program we have started out with: $p'$. The results of this analysis are represented in Figure 8. Note that $m_{12}$ and $m_{44}$ are strictly more-correct than $p$ with respect to $R$; hence (according to Definition 4) the mutations that produced these programs from $p$ constitute fault removals; whence we can say that $p$ has at least two faults, which we write as $\text{faultDensity}(p) \geq 2$. On the other hand, this experiment shows that we can generate a correct program ($p'$) from $p$ by means of three fault removals; if we let the Fault Depth of a program be the minimal number of fault removals that separate it from a correct program, then we can write: $\text{faultDepth}(p) \leq 3$. 

![Figure 8: Relative Correctness-based Repair: Stepwise Fault Removal](image-url)
6 Conclusion

In this paper we discuss how we can use the concept of relative correctness to refine the technique of program repair by mutation testing. We argue that when we remove a fault from a program, in the context of program repair, we have no reason to expect the resulting program to be correct unless we know (how do we ever?) that the fault we have just removed is the last fault of the program. Therefore we should, instead, be testing the program for relative correctness rather than absolute correctness. We have found that testing a program for relative correctness rather than absolute correctness has an impact on test data selection as well as oracle design, and have discussed practical measures to this effect. As an illustration of our thesis, we take a simple example of a faulty program, which we can repair in a stepwise manner by seeking to derive successively more-correct mutants; by contrast, the test for absolute correctness keeps excluding all the mutants except the last, and fails to recognize that some mutants, while being incorrect, are still increasingly more correct than the original. We are not offering a seamless validated solution as much as we are seeking to draw attention to some opportunities for enhancing the practice of software testing.

Our research agenda includes further exploration of the technique proposed in this paper to assess its feasibility and effectiveness on software benchmarks, as well as techniques to streamline test data selection to enhance the precision of relative-correctness-based program repair (re: Figure 7).

Other researchers [14][12][13] have introduced a concept of relative correctness and have explored this concept in the context of program repair. Our work differs significantly from theirs in many ways: we represent specifications by relations whereas they specify them with assertions; we capture program semantics with input/output functions whereas they capture them by means of execution traces; we define relative correctness by means of competence domains and specification violations whereas they define it by means of correct traces and incorrect traces; we introduce relative correctness as a way to define faults whereas they introduce it as a way to compare program versions; we have explored the implications of relative correctness on several aspects of software engineering, whereas they focus primarily on software testing.

This paper complements our earlier work in the following manner: In [16] we introduce relative correctness for deterministic programs, and explore the mathematical properties of this concept; in [4] we generalize the concept of relative correctness to non-deterministic programs and study its mathematical properties. In [5] (Programming without Refinement) we argue that while we generally think of program derivation as the process correctness preserving transformations using refinement, it is possible to derive programs by correctness-enhancing transformations using relative correctness; one of the interesting advantages of relative correctness-based correctness enhancing transformations is that they capture, not only the derivation of programs from scratch, but also virtually all software maintenance activities. We can argue in fact that software evolution and maintenance is nothing but an attempt to enhance the correctness of a software product with respect to a specification. In [7] (Debugging without Testing) we show how relative correctness can be used to define faults and fault removals, and that we can use these definitions to remove a fault from a program and prove that the fault has been removed, all by static analysis, without testing.

Acknowledgements

The authors are very grateful to the anonymous reviewers for their insightful feedback, which has greatly contributed to the clarity and content of this paper.
References

Model-based Testing of Mobile Systems – An Empirical Study on QuizUp Android App

Vignir Gudmundsson  
Reykjavik University  
Reykjavik, Iceland  
vignirgudmunds@gmail.com

Mikael Lindvall  
Fraunhofer CESE  
Maryland, USA  
mlindvall@fc-md.umd.edu

Luca Aceto  
Reykjavik University  
Reykjavik, Iceland  
luca@ru.is

Johann Bergthorsson  
Plain Vanilla Games  
Reykjavik, Iceland  
johann@plainvanillagames.com

Dharmalingam Ganesan  
Fraunhofer CESE  
Maryland, USA  
dganesan@fc-md.umd.edu

We present an empirical study in which model-based testing (MBT) was applied to a mobile system: the Android client of QuizUp, the largest mobile trivia game in the world. The study shows that traditional MBT approaches based on extended finite-state machines can be used to test a mobile app in an effective and efficient way. Non-trivial defects were detected on a deployed system that has millions of users and was already well tested. The duration of the overall testing effort was of three months, including the construction of the models. Maintaining a single behavioral model for the app was key in order to test it in an efficient way.

1 Introduction

Testing of mobile systems (apps) is a new challenge for software companies. Mobile apps are often available on multiple platforms, operating systems, programming languages, etc. This makes it necessary to ensure that the app behaves in the same way independently of the platform on which it runs and of what language is used. In addition, mobile systems often have many configuration options that influence the way they behave. Thus app developers must ensure that the app works as expected for different configurations. Apps are often tested manually. Such manual testing typically relies on the manual creation and execution of test cases that mimic realistic usage of the app. Manual testing is tedious, and is likely to be exhausting rather than exhaustive, especially when a large number of combinations of usage scenarios for various configurations must be covered. For each configuration, the tester has to manually enter data, manually swipe the screen, click on buttons, and manually compare the actual result and behavior with the expected one. Differences between actual and expected result and behavior are then manually documented and reported as issues. It is also very tedious and challenging for humans to validate the expected results of a graphical user interface (GUI) properly, because the state tends to be very verbose. For example, for every screen or state in an app, there are usually many elements to validate. This state of affairs should be compared with API testing, where one has simple error codes as responses.

Whenever a new version of the system under test (SUT) has been developed, the manual tester must go through the same process again. Since new versions are typically released several times per year, and sometimes even twice or thrice per month, the manual testing effort can be significant and is likely to miss key errors because of the large amount of elements to validate.

The problems related to manual testing have been observed by many software organizations who strive to automate the testing process. This has led to the successful use of test case execution frameworks...
Such frameworks are very helpful because they can automatically run the tests once they have been encoded as executable test cases (a.k.a. concrete test cases and test scripts/programs). This type of automation frees up the manual tester because the executable test case automatically enters data, issues commands, compares actual and expected values, and reports detected issues. JUnit is now available also for mobile apps.

However, such test programs must still be created and maintained manually, which can be difficult. One reason is that mobile apps typically have a significant number of screens, input fields and buttons and can be highly configurable. Another reason is that it can be hard to imagine the many different ways in which the app can be used or misused. Thus, the manual creation of executable test cases can be a very complex activity.

In previous work [9, 10, 13], we have used model-based testing (MBT) to test non-mobile systems because MBT addresses the problem of manually creating test cases. Instead of creating one test case at a time, MBT builds a model of the system under test, which is then used to automatically generate executable test cases. While MBT is a promising technology that has been shown to work well for non-mobile systems, it is still unclear if it works well also for mobile apps. This leads to the research question we address in this paper:

**Can MBT be used, in an effective and efficient way, to test mobile systems using the same approach that has been used to test non-mobile systems?**

The reason this is a valid question is that, even though there are many commonalities between a mobile app and a non-mobile app such as the types of controls (buttons, input fields etc.), there are also some differences, such as gestures in the form of swipes, and it was unclear if the modeling notation would allow us to describe these new input methods. In addition, MBT only works well when the app is running in a robust testing environment and it was not clear to us if the available testing environments would be suitable for the type of testing ‘our’ version of MBT needed (e.g. request and response). It was also unclear if the app simulators would be powerful enough, and if not, if testing on a mobile device itself would be an option. To answer our research question, we examined the mobile system QuizUp, which is a trivia game that supports both the Android and iOS mobile operating systems. At the time of the study, QuizUp had over 400 topics including over 220,000 questions, which have grown to over 1,200 topics and more than 600,000 questions since then. We consider QuizUp a strong representative of a standard mobile system, i.e. a system that uses networking, stores data in databases, requires user authentication, contains most basic types of UI elements, etc.

We also address whether MBT can be applied efficiently to mobile apps. In the context of this research, efficient means ‘with reasonable effort’ such as an internship where the duration is limited to six months, during which the person has to understand the system under test, build all testing infrastructure, build the model, generate test cases, test the system, and analyze results. Our tenet is that if all of these activities could be achieved in such limited time, then it would be reasonable to conclude that the technology is applicable to a wide range of systems since we applied the same type of MBT to other types of systems in the past [9, 19, 20]. Thus, another goal of this work was to study the overall costs and benefits of using MBT in general.

The results of our study show that MBT is feasible also for mobile apps because it was applied with reasonable effort and it detected several software bugs that, despite extensive effort, were not found by the traditional testing approach used at QuizUp. Summarizing, this paper contributes an industrial case study, with an analysis of the costs and benefits resulting from using extended finite state machine (EFSM) MBT on a mobile app.

1https://quizup.com/
1.0.1 Structure of the paper
In Section 2, we introduce the MBT approach, which is based on state-machines, that was used. In Section 3, we present the case study where MBT was applied to QuizUp’s Android client. In Section 4, we compare and discuss the similarities and differences between applying MBT on QuizUp and on other systems, especially GMSEC. In that section, we also discuss some related work as well as some ideas for future work. In Section 5, we present and discuss our conclusions.

2 Model-based testing with state machines

2.1 Model representations
We chose to use the EFSM style of modeling representations because state machines are easily understood and well-studied theoretically [22]. Moreover, easy-to-use open source tools that generate test cases from them are readily available [16].

Informally, an EFSM consists of states and transitions [22]. It contains a specific state, called the start state, where the machine starts computing. In the context of MBT, a transition represents a system stimulus, or action, that moves the user from one state of the system to another. Such actions could, for instance, be calling a method or clicking a button. The states are used to embed assertions to check that the user is in the expected system state. These assertions are boolean expressions based on the return code that is expected from a particular method call when a system is in a certain state.

The EFSM model is a generalization of the traditional Finite-State Machine model with guards or helper functions associated with transitions [12]. Guards are boolean expressions or functions that are evaluated during model-traversal time. State variables allow one to store the history of traversal and can be used to embed test oracles in the model. Based on the evaluation of a guard, a model-traversing tool or algorithm chooses among the transitions whose guards are satisfied. Using EFSMs we are able to encode data parameters as variables as opposed to embedding the data at the model level as transitions or states. Such encoding can, for example, be done to parameterize, or initialize, the model at the start of each traversal, storing data as variables or other data structures. This allows one to modify the data to be tested without changing the model and thus it does not affect the number of states and transitions in the model using EFSMs. Hence, the model is easy to review, comprehend, and evolve. In addition, guards and helper functions can be used to return data from actions based on the current traversal history.

2.2 Overview of MBT using state machines
MBT uses a model to represent the expected behavior of the SUT, which is used to generate test cases. Each test case is a path through the model. Such a path can, for instance, be randomly generated—thus building a certain amount of randomness automatically in the testing process. MBT addresses the problem with test cases being static and not covering enough corner cases, which are often exercised by taking an unusual path through the system or using some functionality repeatedly and in unexpected ways.

The model is created from the perspective of a user focusing on the functionality of the SUT’s interface, where a user can be either a human being or another program/system. Thus, testing using this approach is driven by the specifications of the interface of the SUT and does not use the internal structure of the software under test as the driver [24]. The assumed benefit is that, even though the

http://www.graphwalker.com/
complexity of the SUT is large, the model and sub-models typically remain manageable in size. Thus, instead of depending on the complexity of the actual system, the models are as complex as the model of the interface under test.

The process is as follows: 1) A model is built by the tester based on the requirements, existing test cases, and by exploring the SUT. 2) A tool traverses the model and generates abstract test cases, which are sequences of state and transition names. 3) All state and transition names in the model are extracted programatically and turned into a table that lists each state and transition name. This table constitutes the mapping table when executable code fragments are manually entered by the tester for each state and transition. 4) An instantiator program automatically creates executable test cases by replacing each name in the abstract test case with the corresponding code from the mapping table. 5) The test cases are executed automatically. 6) The failed test cases are analyzed by the tester.

3 A Mobile Case Study: QuizUp

QuizUp is a mobile trivia game that allows users to challenge each other on several hundred topics (e.g. arts, science, sports) using almost a quarter million questions at the time of this study. Users participate in a social experience by communicating and competing against friends or strangers in a real-time trivia quiz. The application, initially released for the iOS platform, turned out to be an overnight success\(^3\). Within a year since its initial release, QuizUp released an Android client of the application \(^4\), resulting in over 20 million Android and iOS users.

Testing the application is a complex task due to its data-driven design, its complexity and the configuration options it provides. The data-driven design is embodied by the real-time dependence on large amounts of data to display. The application communicates with the QuizUp servers, through web services, to fetch data (e.g. HTTP GET query) from the QuizUp databases, as well as posting new or updated data to the databases (e.g. HTTP POST query). For most of the scenes in the application, a significant portion of the data being displayed depends on data from the QuizUp databases. Thus, as testers, we can only control data, or the order of data, beforehand in limited scenarios, as we do not have any authority over the databases.

The complexity of the app is largely due to the game-play scenes of the game. A user can compete against other users (a.k.a. opponents) in any of QuizUp’s topics. After the user has requested to play a game on a particular topic, the system searches, in real-time, for an opponent who has also requested to play that particular topic at that particular moment.

It is critical that the app runs on different multiple mobile platforms. The Android and iOS clients are native clients implemented by separate teams. The Android client is implemented in Java; the iOS client is implemented in Objective-C. Nevertheless, the clients should follow the same set of business rules and requirements, and therefore behave conceptually in the same way.

Due to the above-mentioned testing challenges, the QuizUp team conducts large amounts of testing. The team has addressed ambitious testing goals, resulting in the rapid and astounding success of the application, but with a high cost and a substantial effort in thorough Quality Assurance (QA). The testing effort can be divided into several categories. The QuizUp team has developed a set of executable test

\(^3\)https://itunes.apple.com/us/app/quizup/id718421443
\(^4\)https://play.google.com/store/apps/details?id=com.quizup.core
cases, written in Calabash\(^5\), that address some of the testing goals. However, the test suite is manually written and limited to short and common scenarios for the iOS client, and not for the Android client. The team had no automated tests for the Android client; instead it managed a beta group of Android users that provided rapid feedback prior to new releases. The team also included five QA members who constantly verified that new versions and updates of the application met its business rules and requirements through manual regression testing. The QuizUp team had developed an extensive Excel sheet that outlined hundreds of singular tests (rows of action/expected outcome) that QA testers and third party testers used as reference. The team found it tedious and difficult to maintain this spreadsheet. The team also outsourced a large set of end-user acceptance tests to a contractor company that assisted them with QA. Thus, the overall testing effort was significant.

Since the QuizUp team is interested in improving their testing processes through automation, our primary goal was to study the feasibility of using MBT on the QuizUp application. This interest was sparked by the possibility that the QuizUp team would maintain the MBT models and infrastructure after the study. However, the learning curve for applying MBT on a mobile application, such as QuizUp, was unclear. Thus, another goal of the study was to clarify the needed effort.

Following the initial research question, we derived the following sub-questions: 1) Can the QuizUp application be modeled in such way that test cases can be automatically generated to test its core features? 2) Can we design the model in such way that it can test both the Android and iOS client without modifying the model? Since the two clients should represent the ‘same’ QuizUp application, it was desirable to maintain a single QuizUp model instead of maintaining two separate models. Since the QuizUp team has thorough testing processes, we would consider the MBT approach to be successful if it were able to detect non-trivial issues. We decided to test the Android client of QuizUp through its GUI. Although we did not test the iOS client, the implemented testing approach was designed in such way that the iOS client could be easily integrated in future work. We chose the Android client over the iOS client because it was more accessible for test automation due to Apple’s hardware restrictions\(^6\), and due to the fact that no automated tests for the Android client existed. The derived tests were run on mobile emulators using the Genymotion Android emulator\(^7\). Although current emulator technology supports the emulation of the physical state of mobile devices, such as network reception, sensors, battery status, etc., we did not test for those activities and events. The study was mainly performed at QuizUp’s facilities. Questions were asked to the QuizUp team during the process when the documentation was ambiguous or not specific enough. Apart from that, the effort was carried out independently of the QuizUp team. The findings were reported to the QuizUp team during and after the study. In particular, frequent interaction with the QuizUp team was crucial during the construction of the models because we did not have a concrete documentation or specification of the QuizUp system.

### 3.1 Core features of the QuizUp application

The application is divided into scenes, which are either access scenes or in-game scenes. Access scenes identify the user, through log-in or sign-up, prior to using the in-game scenes of the application. In-game scenes enable the logged-in user to use various features of the game, such as playing trivia question rounds against other users, communicating with opponents and viewing leader boards. Most scenes in the QuizUp application contain sub-scenes as well as sub-sub-scenes. Below, we will describe a few selected sub-scenes.

---

\(^{5}\)https://github.com/calabash/calabash-ios  
\(^{6}\)Apple requires automated tests for iOS applications to run on Mac OS X systems to which we had no access.  
\(^{7}\)https://www.genymotion.com
3.1.1 Rules of QuizUp

There are additional behaviors related to the sequence of actions and the navigation between scenes that we describe as rules because they influence the testing that needs to be performed. 1) First-time log-in: New users should see introductory information displayed in the Messages and History scenes. 2) Clean up upon logging out: When a user logs out through the Settings scene, the session data should be cleaned up. 3) Profile changes should be recognized in all relevant scenes: Changing profile information, such as the user’s name or title, should enforce an update in all other scenes that display that particular data. 4) Games should be stored in the History scene: After playing a game against an opponent, a record of that game should be stored in the History scene, where the user can view information and statistics about that game. 5) Sent messages in Chat conversation should be stored in the Messages scene: A message history with a particular opponent can be accessed through multiple different scenes in the application. The conversation should always be up to date independently of where it was entered. 6) The scene prior to entering the Settings scene matters. The Settings scene has two tabs, a settings tab and a profile tab, and which tab should be set depends on the scene prior to entering the Settings scene. The profile tab should be set when entering the Settings scene by using the shortcut button in the Profile scene. The settings tab should be set upon entering the Settings scene from any scene other than the Profile scene and the Settings scene itself. Attempts to enter the Settings scene while the Settings scene is open should be ignored.

3.1.2 Testing questions

From the scenes and rules we derive the testing questions listed below. Some are specific to QuizUp while others also apply to other mobile applications. Q1: Are there issues related to the access scenes displaying error messages when invalid information is input? Q2: Are there issues related to the log-out clean-up functionality? Q3: Are there issues related to displaying the correct scene headers for any given scene? Q4: Are there issues related to the question phase of the Game-play scene? Q5: Are there issues
with the information (scores and statistics) in the Game-play scene after playing a game? Q6: Does the History scene ever show outdated information? Q7: Are there issues related to stored messages in the Messages scene? Q8: Are there issues related to updating user information? Q9: Are there issues with navigating to the correct tab in the Settings scene?

3.2 Applying MBT on QuizUp

Modeling was driven by the documentation of the core features of the application, the associated rules, and the derived testing questions. The model was then used to generate abstract test cases, which were then automatically translated into a set of concrete test cases that were executed on QuizUp’s Android client. For this study, we used the Appium UI test automation tool\(^8\) because 1) it is cross-platform and therefore can be used for both Android and iOS test automation, 2) it is open-source, 3) it is well documented and 4) has a very active community. Appium extends the well-known Webdriver JSON wire protocol specified by Selenium\(^9\). Tests can be written in any language that has a WebDriver library. Languages such as Ruby, Python, Java, JavaScript, PHP, and C# all include a WebDriver library. For this study we chose to use Appium’s Python implementation of the Webdriver library. Appium is a web server that exposes a REST API. It receives connections from a client, listens for commands, executes those commands on a mobile device, and responds with an HTTP response representing the result of the command execution. Appium uses the Android UIAutomator\(^10\) and the iOS UIAutomation tools\(^11\) from the Android and iOS SDK’s to inject events and perform UI element inspection. Using the Appium library, UI element objects can be retrieved by UI inspection. Gestures, clicks and keyboard inputs are examples of methods that can then be applied to the retrieved UI element objects.

3.2.1 The modeling goal

The primary modeling goal was to design the model so that the derived test cases would be realistic albeit unusual while answering the testing questions and determining whether QuizUp is behaviorally consistent with the requirements. A second modeling goal was to design the model in such way that the derived tests from the model could run on QuizUp’s production server. The production server hosts the live version of the application that global users can access. That means that we, as testers, do not have full control of the data space. New users can emerge, the QuizUp team can update the list of available topics, and messages can arise randomly from real users. Thus, we would have to design the model in such way that the implemented test code in the mapping table would not be dependent on specific data in the application, but rather implemented to identify types of data and select elements dynamically.

The third modeling goal was to handle users with different levels of maturity. That is, the derived tests should not be dependent on a particular test user or his status in the game. An arbitrary test user could therefore vary from being a new user who just finished signing up to being an advanced user who has played hundreds of games.

3.2.2 The QuizUp model as hierarchical EFSMs

We modeled the QuizUp application as a collection of EFSMs that were structured in a hierarchical fashion using five layers to manage complexity. It is worth mentioning that some scenes in the application

\(^8\)http://appium.io
\(^9\)http://docs.seleniumhq.org/projects/webdriver
\(^10\)http://developer.android.com/tools/help/uiautomator
can be accessed as sub-scenes from many different scenes. For example, the Game-play scene can be accessed from scenes such as the Home scene, Topics scene and more. Thus, the layer depth of the Game-play scene itself can vary depending on which scene provides access to it.

The highest layer (see Figure 2) of the model is concentrated on using the Email Log-in or Email Sign-up scenes before entering the application’s In-game scenes, where each state for these scenes serves as an entry state to the second layer of the model. The design, thus, explicitly ensures that the ‘dummy’ user (the test case) has logged in prior to using any of the core in-game features. Unusual but valid sequences, such as repeatedly entering and exiting these scenes, are possible outcomes from a random traversal of this model.

As a result of how we designed the highest layer of the model, the second layer of the model is both concentrated on the step-by-step actions of logging in or signing up, as well as accessing the core features of the in-game scenes. The model is designed in such way that it explores realistic, as well as unusual, sequences for inputting the users credentials (email and password) in order to access the application. Prior to inputting the credentials, a "dummy" test user is retrieved from a service called QTDS (QuizUp Test Data Service), which was implemented specifically for this study. The service is currently very minimal and can be viewed as a partial mock-up of QuizUp’s databases. It stores the credentials and key information (e.g. name, title, country) of different test users in a JSON file. Additionally, the user maturity level is logged by using a helper function, since some in-game scenes (e.g. Messages scene) display different information depending on the maturity of the user.

The design of the model permits robustness testing (testing of invalid sequences). Based on the documentation, we categorized the types of emails and passwords that a user can input. There are three types of emails to input; a valid email, an email for a non-existing user (invalid), and a badly formed email (invalid). There are two types of passwords to input: a valid password and an invalid password. The model permits any combinations for inputting these different types of emails and passwords. Each transition for these email and password types has an associated helper function that helps us identify which combination was traversed. We are then able to embed the appropriate test oracle for validating the traversed combination. The ‘validEmailLogin’ guard only returns true if both a valid email and a valid password are provided as input. Otherwise, the guard returns false. The ‘invalidEmailLogin’ guard then allows us to understand which type of error message we should expect to be displayed depending on the type of invalidity of the input combination.

The second-layer model for the Email Sign-up scene has a similar structure and emphasis as the model for the Email Log-in scene. The second-layer model for the in-game scenes of the application is shown in Figure 3. The model includes entry states for the Profile, Home, Topics, History, Messages and Settings scenes, as well as a state for the application’s sidebar. Each of the entry states has a sub-model
(in the third layer of the model). Therefore, the model is designed as an intermediate layer for navigating between different in-game scenes using the sidebar. When the ‘dummy’ user is led to a new scene during the traversal, the helper function ‘Scene’ is used to add that particular scene to an array which contains the history of traversed scenes for a particular traversal. The function allows us to later embed assertions that rely on the history into our model. As described above, a user can log out from the Settings scene. Thus, a transition for logging out is available from the Settings scene and will bring the user back to the Welcome scene.

### 3.2.3 Generation of abstract test cases

A model traversal starts at the start state in the highest layer of the model and stops when a given stopping criterion is met. We used yEd\(^{12}\), to create the models in this study and used Graphwalker to traverse the models\(^ {13}\). Graphwalker is an open-source Java library that generates test sequences from FSMs and EFSMs that are, for instance, created using yEd models and stored in Graphml format. Graphwalker offers stopping criteria such as state coverage and transition coverage. We used Graphwalker to generate 100 abstract test cases with built-in assertions for the core features using the random-path algorithm covering 100% of states and transitions. The number 100 is a balance between the costs and benefits of generating and running many test cases. Generating a large number of test cases does not cost anything, since there is no effort or significant time associated with test case generation. Running a test case does not cost anything either in terms of manual effort. However, it takes a relatively long time to execute a test case. For practical purposes it turns out that 100 test cases (in this case) were a good number because the tester could start the test execution, leave for a couple of hours or the night, and then come back and

\(^{12}\text{http://www.yworks.com}\)

\(^{13}\text{http://www.graphwalker.com}\)
analyze the results.

### 3.2.4 Mapping abstract labels to concrete code fragments

We used Appium’s Python library to communicate with the QuizUp application. The code fragments for each label in the QuizUp model were manually inserted into the mapping table. We wrote a script that automatically tracks whether new states or transitions were added to the model, and if so, it generates a template code fragment for new labels.

We learned from the GMSEC case study that non-executable mapping tables can be messy and that maintaining a single mapping table becomes challenging when the model size increases [10]. Thus, we decided to implement a class structure which would allow us to have a separate mapping table for each scene. Each scene class inherits a scene base class. The base class includes generic code fragments for actions accessible from all scenes (e.g. pressing the Android back button). Therefore, it was unnecessary to implement code fragments for such actions in each scene class.

### 3.2.5 Executing the concrete test cases

For this study, we implemented a small command-line tool called Kelevra intended for MBT projects. The goal for the tool is that both inexperienced and experienced individuals in MBT can use it as a unified interface to trigger the various phases of MBT. Using Kelevra’s `instantiate` command we were able to automatically create concrete test cases from the set of 100 abstract test cases. We then used Kelevra’s `run` command with Appium configured as an argument to execute the test suite on two different Genymotion emulators. The Google Nexus 5 and Samsung Galaxy S4 devices were the chosen devices to emulate.

#### Table 1: Questions that detected issues.

<table>
<thead>
<tr>
<th>ID</th>
<th>Testing Question</th>
<th>Ans.</th>
<th>UI</th>
<th>DATA</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q3</td>
<td>Are there issues related to displaying the correct scene headers for any scene?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
<tr>
<td>Q8</td>
<td>Are there issues related to updating user information?</td>
<td>Yes</td>
<td>No</td>
<td>Yes</td>
</tr>
</tbody>
</table>

### 3.2.6 Analyzing the results and effort

See Table 1 for a summary of the results from executing the concrete test cases. In Column 3, an answer to each testing question is provided. Column 4 is devoted to issues that can be traced back to badly formed or missing UI elements. Column 5 deals with issues that can be traced back to incorrect data. The value in a particular cell indicates whether we detected issues related to a particular testing question.

**Sample issue related to updating user information** We detected an issue in the key scene list in QuizUp’s sidebar in relation to Testing Question 8. The issue, which was not known to the QuizUp team despite their testing effort, originated in the Settings scene (Profile tab) when the ‘dummy’ user updated his name. The name should thereafter be updated in all scenes. However, this was not the case and therefore the test case failed.

The EFSM model for QuizUp consists of 125 states, 192 transitions, 87 requests (method calls), and 74 assertions. The reason for the significant difference between the number of transitions and the number
of states is that actions such as opening the sidebar and navigating back using the Android back button were accessible from a majority of the states. The high number of assertions is due to the fact that we were able to embed appropriate test oracles using EFSMs.

See Table 2 for effort spent before modeling and testing the QuizUp application. We had learned MBT and the Graphwalker tool through the GMSEC study [10], therefore, no such effort was required for this study. However, applying MBT on a mobile application was new to us and we had to familiarize ourselves with the Appium test driver. We also evaluated other options, such as using the Android UIAutomator directly. We also spent time setting up the Genymotion emulators. Finally, we had to implement a test driver for Appium that was plugged into Kelevra.

<table>
<thead>
<tr>
<th>Task</th>
<th>Effort</th>
</tr>
</thead>
<tbody>
<tr>
<td>Understanding test automation tools, setting up Genymotion emulators</td>
<td>2 weeks</td>
</tr>
<tr>
<td>Implementing the Appium test driver customized for QuizUp</td>
<td>1 week</td>
</tr>
<tr>
<td>Designing the EFSM model and mapping tables</td>
<td>5 weeks</td>
</tr>
<tr>
<td>Executing test cases and analyzing the results</td>
<td>1 week</td>
</tr>
</tbody>
</table>

We then spent time understanding QuizUp, incrementally designing the QuizUp EFSM model, and maintained the mapping table as states and transitions were added. During those five weeks we also implemented the test oracles for the EFSM model. There were 15 guards and helper functions (112 source lines of Java-like code that is executed during model traversal) implemented for the QuizUp model. After we finished the model and mapping table, a week was spent on executing the generated test cases and analyzing the results. Thus, it took 9 weeks for a programmer who was familiar with MBT (but not with mobile applications) to create a significant EFSM model, generate test cases, instantiate them, execute them on two different emulators, and analyze detected defects.

4 Discussion and future work

Despite the fact that QuizUp is a mobile system, it has certain similarities with non-mobile system we previously tested allowing us to use the same MBT approach again. The key similarity is that QuizUp is a state-based system, where the user interaction can be described as sequences of events in the form of stimuli and responses. This stimuli-response pattern allowed us to model the app as a state machine, which is the key component of the MBT approach used in this study. Most reactive systems can be described in the same way. However, some reactive systems do not provide a detailed enough response to all stimuli, making it difficult to determine their state.

We will now compare the QuizUp results to our most recent study on GMSEC [10]. At the time of the QuizUp study, we had gained experience in MBT which meant that we were much more efficient in setting up the QuizUp study (3 weeks compared to 7 weeks in the previous study). Based on the results from the previous study, we were also able to immediately determine that EFSMs would be the most appropriate model representation for QuizUp without first experimenting with FSMs. When comparing the model size (i.e. numbers of states and transitions) for GMSEC and QuizUp, it is clear that the QuizUp model is much larger. The GMSEC model consisted of 49 states and 62 transitions while the QuizUp model consisted of 125 states and 192 transitions. The reason for the difference is twofold. Firstly, we
tested more features for QuizUp than for GMSEC and, secondly, in QuizUp, we were able to retrieve more detailed information about the state of the system at any given time. In QuizUp, for example, error messages (text strings displayed in the GUI) were separate for different scenarios or sequences. On the other hand, for GMSEC, we only used simple API return codes to assert whether an action should be successful or not.

There are 8 requests and 15 assertions for the GMSEC model, but 87 requests and 74 assertions for the QuizUp model. The reason for the large difference is that a state in GMSEC reflects the expected behavior of the system after a particular API method call, whereas a state in QuizUp is comprised of dozens of UI elements that formed a view or a scene in the application. Therefore, for some states in QuizUp, we had to validate multiple UI elements, and data, in order to be confident that we were in fact in a particular state.

An idea based on the study is to create a protocol for modeling mobile applications. Since mobile applications are built using platforms such as Android and iOS, we could provide a standard protocol for modeling common patterns and UI behaviors. The protocol would probably only cover a small subset of possible patterns, but it would be convenient to use the suggested models provided by the protocol as templates when modeling. However, we did not construct a protocol for modeling mobile applications because the projected effort was out of scope for this study.

In addition to a mobile model protocol we discussed adding a simple textual language to describe the present UI elements and patterns in a particular view or scene in a mobile application. For example, ‘Click Button Ok’ could be a statement that would be compiled, or interpreted, to a transition in a model. This could especially help inexperienced individuals in MBT with little modeling experience. However, since we did not construct a mobile model protocol, we decided not to design and implement a textual language.

A couple of notable studies of mobile testing can be found in the literature. In [1] Amalfitano et al. focus on testing the GUI of a mobile app to find bugs. Since mobile apps are extremely state sensitive, they use test-case generation based on state machines (as opposed to the ‘stateless’ event-flow graphs they employed in their earlier work). They also develop techniques to handle security in mobile apps. The MobiGUITAR framework is implemented as a tool chain that executes on Android. MobiGUITAR uses ripping (an automatic method for creating a state machine model of the GUI of the app), test generation (from the model and test adequacy criteria) and test execution (of tests in the JUnit format in the current implementation). They applied the tool to test four apps from Google Play: Aard Dictionary, Tomdroid, Book Catalogue and WordPress Revision 394. Their testing revealed ten bugs from 7,711 test cases in total. Their conclusion is that the combination of model learning (their ripping phase) and model-based testing is promising for achieving better fault detection in Android app testing. The main difference is that Amalfitano et al. rip the GUI from the app automatically and create a model from it, while our model is based on the requirements and other artifacts that describe the app. As far as we understand, in the work by Amalfitano et al. the test cases do not have oracles and determine failure based on crashes, while our model has oracles for every request-response pair.

In [7] de Cleva Farto and Endo address similar research questions to ours and also focus on apps for Google Android. They applied MBT to test the AddressBook app modelling the behaviour of the SUT as an Event Sequence Graph (ESG). The model was created manually as in our study. They develop abstract test cases from the ESG models and then make them concrete and execute them using the Robotium platform. The research questions addressed in that paper are very similar to ours and so are their conclusions. The main difference is that de Cleva Farto and Endo let three groups conduct MBT as an experimental study for a short time (less than an hour), whereas our study spanned three months of work in order to conduct in-depth testing of a commercial app. It is also unclear whether and how
oracles are modeled in their approach, while oracles are modeled as an integral part of our methodology.

5 Conclusions

We presented an empirical study where MBT was applied to the Android client of QuizUp through its GUI. The main goal of the study was to examine whether MBT could be used, in an effective and efficient way, to test mobile systems using the same approach that has been applied to other types of systems, and whether this would be feasible with reasonable effort.

The study shows that we were able to use an EFSM-based MBT approach to test a mobile system. Although QuizUp is indeed different from other systems we have tested (e.g. QuizUp has a graphical user interface, GMSEC does not), there are certain similarities that allowed us to use the same MBT approach. The most notable similarity is that both systems are reactive, state-based systems where the user interaction can be described as sequences of events. We found that maintaining a single behavioral model was key in order to test the QuizUp app in an efficient way. This was demonstrated, for example, by the fact that the test cases were able to detect non-trivial issues in a system that was already well-tested. Regarding the effort, as a comparison, much of the effort in the GMSEC study was devoted to learning and applying MBT for the first time, whereas our gained MBT experience allowed us to set up faster and perform a more extensive modeling effort for QuizUp. The effort data and the detected defects show that MBT started paying off as soon as we had applied the process for the first time. We also found that MBT provides a systematic way to test mobile systems and, even though there are still manual steps involved, it is possible to achieve a high degree of automation with reasonable effort for someone who has no or little previous experience with MBT. A possible extension to our work would be to minimize the manual steps even more. Constructing a protocol for modeling mobile applications, for example, would be beneficial to standardize the modeling effort. Another option would be to implement a language to describe a SUT and its possible actions. For mobile applications, we would describe the UI elements and patterns in a particular view or scene under test. The constructed textual description could then be translated into a model representation such as EFSM.

Acknowledgements: This research was partly supported by the project “TheoFoMon: Theoretical Foundations for Monitorability” (grant number: 163406-051) of the Icelandic Research Fund.

References


Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates

Sönke Holthusen¹, Sophie Quinton², Ina Schaefer¹, Johannes Schlatow¹, Martin Wegner¹

¹TU Braunschweig
{s.holthusen, i.schaefer}@tu-bs.de, schlatow@ida.ing.tu-bs.de, wegner@ibr.cs.tu-bs.de

²Inria Grenoble - Rhône-Alpes
sophie.quinton@inria.fr

As the issue of change after deployment in safety-critical embedded systems, our goal is to substitute lab-based verification with in-field formal analysis to determine whether an update may be safely applied. This is challenging because it requires an automated process able to handle multiple viewpoints such as functional correctness, timing, etc. For this purpose, we propose an original methodology for contract-based negotiation of software updates. The use of contracts allows us to cleanly split the verification effort between the lab and the field. In addition, we show how to rely on existing viewpoint-specific methods for update negotiation. We illustrate our approach on a concrete example inspired by the automotive domain.

1 Introduction

Critical embedded systems currently offer limited support for software updates. In the automotive domain, this leads to outdated software being used on mechanically durable vehicles. Adaptable vehicle electronics could be used, for example, for increasing safety via software patches, or to limit costs by a better usage of computational resources. In a different context, aerospace computing platforms are currently migrating from fixed, ground-verified configurations to re-configurable platforms, and from functionally separated solutions to integrated systems with the capability to change during operations. These two examples illustrate the fact that we need to support flexible adaptation of safety-critical embedded systems via continuous change after deployment.

At the moment software updates in critical embedded systems are prepared via intensive verification and tests performed in the lab. Unfortunately, this lab-based verification process is becoming increasingly difficult, in particular because of complex platform dependencies between changing applications—e.g., for example through timing, fault handling, security mechanisms, etc. Every particular system configuration is different and there are just too many of them to perform exhaustive a priori verification, even using abstraction techniques. For this reason, our goal is to use in-field formal analysis to determine whether a specific update in a given system configuration may be safely applied.

Although promising, in-field verification of software updates for safety-critical embedded systems is also challenging because it requires an automated process able to handle multiple viewpoints such as functional correctness, timing or security, with access to limited resources. For this purpose, we propose a contract-based methodology. Such an approach is particularly well-suited to cleanly split the verification effort between the lab (pre deployment) and the field (post deployment). In addition, it lends itself to our original approach whose specificity is to address multi-viewpoint verification based on a combination of existing viewpoint-specific methods. Our work is motivated by the project Controlling...
Using Multi-Viewpoint Contracts for Negotiation of Embedded Software Updates

Concurrent Change (CCC)\(^1\), which addresses new methods to develop and control embedded system platforms integrating changing applications under high requirements to real-time, safety, availability, and security. The methodology is currently being implemented as a complete tool chain. We strive to present our current results both at a high level of abstraction, so that our results can be reused or adapted to other contexts, and at a lower level of abstraction so that our theory matches the practical needs of the CCC project. This dual approach is reflected in this paper.

This paper is organized as follows. Section 2 introduces the general methodology that we propose. Section 3 then presents the actual context in which we develop this approach. In Section 4 we show the effectiveness of our methodology on a concrete example in the automotive domain. Finally, Section 5 discusses the state of the art and Section 6 concludes.

2 Contracting for software update negotiation

We present here our general approach to in-field negotiation of software updates.

2.1 Problem statement

Let us state first that, independent of verification, software updates require a component-based approach. The overall objective of in-field negotiation is then to guarantee that conformance to system requirements is preserved during the update. It is however likely that establishing conformance monolithically will not be feasible in practice; hence the use of contracting, which clearly splits responsibilities between a component and its environment so as to make design and verification easier. As a result, our methodology assumes that the following primitives are given.

Component framework. We suppose a given notion of component so that one can specify requirements and verify that a given system conforms to them.

- component: In this paper we address software updates so the term component refers to software components. How these components are specified depends on the actual framework, but at this point we do not make any assumption about them. A set of components form what we call a software model.
- system model: A software model alone is not sufficient to describe a system if one wants to verify non-functional properties. Therefore, we define a system model as a triple made of:
  - a software model;
  - a platform model, which is not subject to change;
  - a configuration that describes all runtime parameters relating components and platform which may be changed during negotiation.
- system requirements: The whole verification problem supposes that there exists some formal representation of the expected behavior of a system, which we suppose given as a set of requirements. In addition, we need a formal relation between systems and requirements, that we call conformance, to allow establishing that a given system conforms to its specified requirements.

\(^{1}\)http://ccc-project.org/
**Contract theory.** We further assume primitives relating to contracting.

- **contract**: A contract for a component $K$ is a pair $(A, G)$ consisting of an assumption $A$ on the system in which $K$ will execute and a guarantee $G$ on the way $K$ will behave in such a system.

- **contract satisfaction**: Components and contracts are related via a contract satisfaction relation. Intuitively, a component satisfies a contract $\mathcal{C} = (A, G)$ if, assuming that $A$ holds, then $G$ holds, too.

- **compatibility of contracts**: This concept is derived from the above primitives. A set of contracts is compatible with respect to a platform model if there exists a feasible configuration, i.e., a configuration such that, in any system made of components satisfying the given contracts, if assumptions on the environment of the system hold then all assumptions and therefore all guarantees hold.

Contract compatibility is the cornerstone of the contract-based approach as it does not require the actual components to be available, only their contracts. Because of that, we use from now on the term software model to refer to the set of contracts abstracting the components.

Besides, note that an additional verification step is in principle needed after compatibility has been established in order to derive conformance to system requirements from assumptions and guarantees. In our case, system requirements are always expressed directly as assumptions or guarantees so we will largely ignore this step in the rest of the paper.

In a context where a component framework and a corresponding contract theory have been developed, we can now formulate our problem as follows.

**Definition 1** An update request is a pair $(\text{changeType}, \mathcal{C})$ where

- changeType describes the change to undertake which is one of the following \{add, remove, update\};
- $\mathcal{C}$ is the contract attached to the component undergoing change.

The objective of the negotiation process is to determine whether the update request can be granted and if that is the case then for which configuration.

**Definition 2** A negotiation process is a function taking a system model $\text{sys} = (\text{software}, \text{platform}, \text{config})$ and an update request $(\text{changeType}, \mathcal{C})$ as parameters and returning an answer: either no, or yes along with a feasible configuration $c$.

This means that negotiation is a synthesis problem: we need to synthesize a configuration that makes all contracts compatible.

### 2.2 Design and verification flow

The use of contracts effectively splits the verification process between the lab and the field, as we show in the following.

**In the lab.** At design-time (see Figure 1), during software development and test in the lab, additional data for helping the in-field negotiation process are prepared and formalized as contracts. They will typically include properties of the software components which can only be obtained using involved verification methods, e.g., requiring manual intervention such as theorem proving, or intensive computation, for example in the case of model checking. Assumptions on the environment in which the system will function may also be specified. Contract satisfaction is then established in the lab, as both the component and its contract are available. This step is performed for every component independently and saves the effort to verify every possible combination of components and possibly versions of components.
In the field. After deployment, whenever an update is requested, formal verification is performed to determine whether the update may take place without risking a violation of system requirements. This step precedes the actual update and the system architecture is designed to ensure that it does not disrupt the running system. A negotiation process takes place, aiming at finding a system configuration which will guarantee that all system requirements will be satisfied after the update. This is the challenge of the approach as this synthesis and verification phase must be performed automatically, with limited memory and in reasonable time. Updates will typically be performed while the system is suspended (e.g., overnight while the car is parked) so that the time constraint exists but is less tight than the memory constraint.

To succeed, the component in charge of the negotiation, which we call the Negotiation Controller (NC), can rely on the contracts that have been prepared at design time. A first specificity of our approach is that we do not try to tackle all viewpoints together as in e.g., [3]. Instead, we rely on viewpoint-specific analysis engines, each of which uses a model that is best adapted to the particular viewpoint it is dealing with. This also means, we do not encode the search for a configuration into one big constraint-solving problem including all viewpoints. We reduce the possible variables for the main problem to a minimum, while the viewpoints can solve viewpoint-specific problems on their current partial configuration. Besides, the brute-force approach that involves checking all possible configurations until a feasible one is found is not viable due to the size of the configuration space. To tackle this, a second specificity of our approach is that we rely on added capabilities of the viewpoint-specific analysis engines. Indeed, our experience is that these engines, although used in general for checking one given configuration, can be extended to provide information about sets of configurations. Our strategy is therefore to start from an over-approximation of the set of feasible configurations and iteratively update this set based on the results provided by the analysis engines. More precisely we proceed as shown in Figure 2.

1. The NC computes a set of configuration candidates which is represented as a set of constraints.
2. The NC then picks one candidate and triggers the viewpoint-specific engines so that they check if that candidate configuration is feasible w.r.t. the considered viewpoint — and as such a witness to the software model being compatible with respect to the platform model.
3. If the configuration is not feasible, one engine returns a set of additional constraints to the NC. If the configuration is feasible for all viewpoints then the update can be accepted.
4. The last two steps repeat until the NC can no longer find a candidate or if a feasible configuration has been found.

In the rest of this paper, we show in more detail how we follow this approach in a specific, applied context.
3 Software update negotiation in the CCC project

In this section, we present the architecture that we are currently developing to support negotiation in the context of the CCC project. Our goal in this section is to define all the concepts used in Definition 2, namely: a software model represented as a set of contracts, a platform model and a notion of configuration — and to show how we use them to perform contract negotiation.

Platform architecture. Similar to integrated architectures like AUTOSAR [2] or ARINC 653, our approach offers a clear separation between software/application and hardware/platform components through a Runtime Environment (RTE). We use the Genode OS Framework [6] as RTE, which allows for the design of component-based systems where access control is applied via so-called capabilities. Our architecture complements the RTE with a so-called Multi-Change Controller (MCC) which acts as negotiation controller. Similar to the EPOC [9] project, the MCC is divided into a model domain, in which analyses are performed without actually modifying the system, and an execution domain, in which the update is implemented. In this paper, we focus on the model domain.

Component framework. Our software is built as a set of functions made of components. For convenience, we only consider here functions with a single component so most of the time, we will abstract functions away. Components are interconnected via services: components can offer services to each other, or require services from one another. Matching between offered and required services is made via comparison of the corresponding service interfaces, which describe the services’ expected type of communication.

Communication between components can be performed synchronously via Remote Procedure Calls (RPCs) or asynchronously using signals. More precisely, RPCs allow passing arguments to the component being called, but the caller cannot continue its execution until the callee has finished handling the request. In contrast, signals can be used asynchronously, only notifying the callee and ending the communication afterwards.

Components are implemented as one or more threads such that each thread can be activated by: (1) another component requiring a service via an RPC or a signal; (2) another thread of the same component; or (3) a timer.
Requirements. Although the scope of the CCC project is larger, we consider in this paper the following viewpoints.

- **functional dependencies**: This viewpoint addresses issues related to the structural information presented above, e.g., to guarantee that all required services are indeed provided.
- **timing**: The timing viewpoint is typically concerned with establishing that a function reacts to an input within a specified delay.
- **functional correctness**: Functional correctness focuses on control flow properties, e.g., expressing that a given service cannot be called before some initialization step has been performed.

All the above system requirements can be expressed at the component or function level so we specify them directly in our contracts. Conformance to these requirements is thus established if contract compatibility is guaranteed.

**Contract framework.** To express assumptions and guarantees related to our three viewpoints we have developed a contract language capturing all the relevant information. Rather than using a formal definition, in the following we choose to describe our language through an example.

**Functional dependencies.** Listing 1 shows the contract for a component T, which requires a service **object_recognition** (line 3) and provides a service **trajectory_calculation** (line 4). The names of the required and provided services are IDs, linking to a more detailed service description in a global repository. Knowing the required and provided services of all components, functional dependencies can be resolved [14].

**Timing.** The timing requirement is specified at lines 20–22 and states that the RPC call to **object_recognition** must complete within 100 time units. In addition, to enable timing analysis [7], contracts must contain enough information to extract a task graph [8]. A task graph is a directed graph in which nodes represent tasks and edges represent the asynchronous (or synchronous) activations/calls and returns. For that purpose the thread structure as well the nature of communication (RPC or signal) must be known. In our example, the component is made of two threads called **trajectory_calculation_get** and **trajectory_calculation_init**. The former is triggered via RPC (line 7 in the listing). When activated it first proceeds with executing some local processing task which takes between 1 and 5 time units to complete on a computing resource of type **CPU_type_1** (lines 8 to 10). It then performs an RPC call to use a service (line 11) and finally executes another task before completing (lines 12–13).

```plaintext
1 component T
2   services
3     requires object_recognition
4     provides trajectory_calculation
5   threads
6     thread trajectory_calculation_get
7       on RPC trajectory_calculation.get() 
8       task tc1
9       onto CPU_type_1
10       wcet=5 bcet=1
11       RPC object_recognition.get()
12     task tc2
```
onto CPU_type_1
wcet=5 bcet=1
thread trajectory_calculation_init
on RPC trajectory_calculation.init()
task tci
onto CPU_type_1
wcet=10 bcet=5
timings
timing 100
object_recognition.get()
control_flow
not trajectory_calculation.get()
until trajectory_calculation.init()

Listing 1: An example contract.

**Functional correctness.** Finally, the functional requirement is specified at lines 23–25. It requests that the trajectory calculation must always be initialized before it is used. Such control flow properties can be easily checked on a control flow graph extracted from the contracts.

Note that our contracts do not explicitly distinguish assumptions from guarantees. In addition, a significant part of the guarantees hold independent of the assumptions and can thus be regarded as part of a specification rather than a contract. We have chosen this presentation to improve readability. In the contract for component T, the assumptions correspond to the three viewpoint-specific requirements. The most notable guarantees regard worst-case execution time bounds, which can only be formally established using involved verification techniques including abstract interpretation [18].

**System model.** Let us synthesize the information presented above to define our system model. Remember that a system model is made of a software model, a platform model and a configuration.

- **software model**: this is the set of contracts used as an abstraction layer on top of components. Note that only the components that are actually part of the chosen configuration will be loaded when a new configuration is set up.

- **platform model**: for contract negotiation we need some abstraction of the platform as well. The only non-functional viewpoint that we address here is timing, so information about the available processing resources and their scheduling policy is sufficient, as will be detailed later. As already mentioned, we assume that the platform model is fixed.

- **configuration**: this describes how the selected software components are connected to each other and to the hardware. It consists of a set of connections (functional dependencies) between offered and required services, a mapping of tasks to the platform resources, and priorities assigned to tasks.

Denote software the set of software contracts that make up the software model. The set of services (and whether they are required or offered by a given component) as well as the set of tasks that correspond to a given component can easily be extracted from the component’s contract.

Denote platform the set of processing resources that constitute the platform model. In our case all resources have the same scheduling policy, namely static-priority preemptive, so the only additional information we need about them is their type in order to guarantee an appropriate mapping of tasks.

**Definition 3** A configuration config is a tuple \((C, SC, M, Π)\) where:
• $C \subseteq \text{software}$ represents the set of selected components, i.e., components which must be loaded and run on the platform.

• $SC$ is a set of service connections, i.e., of triples $(c_1, s, c_2)$ with $c_1, c_2 \in C$ representing a connection between two components $c_1$ and $c_2$ via a service $s$.

• $M$ is a function associating to each task the resource onto which it is mapped.

• $\Pi$ is a total priority order on tasks.

Note that not all these configurations are well-formed as we discuss below.

**Negotiation process.** In this section we formally define how the configuration space is constrained during the contract negotiation. The objective of the negotiation process is to find a feasible configuration. To achieve this the MCC picks a configuration from a set of candidates called the configuration space and runs it through the viewpoint-specific analysis engines to determine whether it is feasible. Initially the configuration space contains all well-formed configurations (see below). It is then iteratively restricted based on the feedback provided by the viewpoint-specific analysis engines.

It has been shown in [14] that the functional dependency resolution problem can be formulated entirely as a set of constraints, so there is no need for a specific, dedicated analysis engine. Such constraints include in particular:

1. For all $(c_1, s, c_2) \in SC$, $c_1 \neq c_2$ and the service $s$ is indeed required by component $c_1$ and offered by component $c_2$.

2. For every selected component $c_1 \in C$ requiring a service $s$, there is a unique selected component $c_2 \in C$ such that $(c_1, s, c_2) \in SC$.

Regarding mapping and priority assignment, not all configurations are well-formed either as the following constraints must be met:

3. A task $\tau$ can only be mapped onto a resource $r$ if $r$ is of the type for which execution time bounds are provided in the contract corresponding to $\tau$.

4. Tasks from the same thread must have the same priority.

**Definition 4** A configuration $(C, SC, M, \Pi)$ is well-formed if and only if it satisfies the four above mentioned conditions.

In contrast, functional correctness may require intricate analysis which is best kept separate from the main set of constraints. Whenever a configuration is rejected by the functional correctness analysis engine checking control flow properties, a new constraint is added to restrict $SC$. Such constraint will typically forbid any configuration in which $c_1$ is selected and $(c_1, s, c_2)$ is a service connection if it has been established that $c_1$ is “misusing” the service offered by $c_2$. Note that this removes a fairly large set of configuration candidates, namely all possible mappings and priority assignments for each $SC$ eliminated.

Finally, timing analysis is the most challenging part of our negotiation scheme. Encoding the response-time analysis as an ILP (integer linear programming) problem has been done [17], but at the cost of either a prohibitive complexity or pessimism, meaning that configurations which may be feasible are discarded. Our strategy is to perform timing analysis using state of the art timing analysis tools such as e.g., pyCPA [12] but to better exploit the provided result. Suppose for example that timing analysis cannot guarantee that a given task $\tau$ will meet its deadline in the worst case. This means that for the current mapping any configuration that keeps the same priority for $\tau$ and higher-priority tasks will violate $\tau$’s timing requirement. A full theory of how such constraints can be generated is in progress and out
of the scope of this paper but it is particularly interesting to remark that such issues have been relatively
ignored so far in the real-time systems research community.

4 Application to an automotive example

Let us now illustrate our approach to contracting for update negotiation on an example taken from the
automotive domain. We consider a car in which a function for parking assistance is deployed and show
how it can be updated to add a function for lane detection. We first describe the software components we
use and their relevant properties before elaborating on the negotiation process.

4.1 System model and viewpoints before the update

The component P for parking assistance depends on trajectory_calculation, a service for calculating a trajectory for the car (i.e., a list of vectors to steer the vehicle). That service, which is provided
by component T, itself needs an object_recognition service to determine which objects must be avoided. Suppose that two components offering this service are available (O₁ and O₂), with O₂ additionally offering a service object_masking for masking objects.

**Initial configuration.** Assume that during deployment all the above components have been uploaded
onto a unique hardware resource called CPU1. Figure 3 shows how functional dependencies are solved in
the current configuration. Note that only P, T and O₂ are selected — meaning that the code and contract
of O₁ are available but not currently in use in the system. The rest of the configuration, namely task
mapping and priority assignment will be discussed later.

![Figure 3: Resolution of functional dependencies chosen during deployment. Boxes represent compo-
nents, circles and semicircles represent services that are offered and required, respectively. Service
names are abbreviated to their acronym.](image)

**Initial requirements.** Let us describe our system requirements. Listing 2 shows the contract for com-
ponent P. Note that two keywords have not been introduced before, namely *initialization* and
*time*. The former refers to an initial *operational mode* that precedes the normal mode in which all other
calls are performed. The latter term is used to introduce an *activation pattern* which describes how often
a given thread is activated based on a timer. In our case, thread park_assist is activated periodically
every 200 time units with a possible jitter of at most 5 time units.

The contract for T has already been introduced in Listing 1. Due to space constraints the contracts
for components O₁ and O₂ are omitted. In addition to functional dependency constraints, there are only
two timing requirements and one control flow property to verify:

1. Each activation of the park_assist thread must complete within 150 time units — this require-
   ment is specified in Listing 2.
Listing 2: The contract of P

component P
services
  requires trajectory_calculation
threads
  thread init
    on initialization
      RPC trajectory_calculation.init()
  thread park_assist
    on time (period=200 jitter=5)
      task p1
        onto CPU_type_1
        wcet=3 bcet=1
        RPC trajectory_calculation.get()
      task p2
        onto CPU_type_1
        wcet=7 bcet=1

 timings
  timing 150
  park_assist

2. A call to object_recognition_get must return within 100 time units.
3. trajectory_calculation_get must not be called before at least one call to trajectory_calculation_init has been made.

The last two requirements are from the contract for component T.

Timing viewpoint. The task graph presented in Figure 4 shows two task chains corresponding to the normal mode of operation of P and its initialization mode, respectively. This graph has been obtained by unfolding the task graphs inside the thread description with the concrete task graphs of called threads. This means the RPC to trajectory_calculation.init() in Listing 2 is replaced by the corresponding task graph from the contract for component T (Listing 1). Furthermore, the call of object_recognition.get() is replaced by the corresponding task. This continues until all RPCs are replaced by tasks. Note that O2 has a task which is never activated (namely the task corresponding to the object_masking service) and is thus omitted in the task graph.

Figure 4: The task graph before the update. Note that or2 corresponds to the task performing the object_recognition service in component O2.

Because our platform model consists of only one resource, all tasks are mapped to CPU1. In presence
of several operational modes timing analysis is performed separately for each mode [15]. Therefore in the normal mode of operation only the first chain of tasks is considered. Our two timing requirements translate at the task level into latency constraints: one from a call to task p1 to the end of p2; the other from a call to tc1 to the end of tc2. We will detail how such requirements are verified in the description of the negotiation process.

Control flow graph. In our example we have chosen a very simple functional correctness requirement, which can be directly derived from the semantics of the keyword initialization in our contract language. We therefore omit the description of the control flow graph that can be derived from the contracts and focus on the more intricate timing aspects of our example.

4.2 Update scenario

Our objective is to add to the system a component L implementing lane detection, for which it requires services for object_recognition, object_masking as well as a service for steering provided by a component S which is also to be uploaded. We omit the contract for S but show the contract for L in Listing 3: component L is made of a single thread which sequentially calls services object_recognition, object_masking and steering while performing some internal computation in between calls.

4.3 Contract negotiation

Now we can show how our negotiation process is performed to allow the addition of the lane detection function.

Functional Dependency. The first step of contract negotiation aims at finding the set of partial configurations guaranteeing that every service required by a component in the software model is provided by another component in that model. In our case we need to account for the services required by the component to be added to the software model, namely L (note that S does not require any service). Functional dependency is thus solved on the software model consisting of components P, T, O1, O2, S and L. Figure 5 illustrates the result, as we explain now. Full lines between offered and required services represent links for which there is no alternative. For example, T is the only component offering service trajectory_calculation required by P, making the connection between the two mandatory. In contrast, the object_recognition service required by T can be provided by O1 and O2, which we denote using dashed lines. Any configuration which matches one of those represented in Figure 5 will guarantee compatibility at the functional dependency viewpoint. For the sake of simplicity we will assume that the service interfaces for object_recognition prevent O1 and O2 from offering that service to more than one component. In that case there are only two possible solutions left.

Functional correctness. The part of the control flow graph which is relevant for the requirement imposed by T does not change after the update so the functional correctness viewpoint does not need to constrain the configuration space: any configuration which satisfies the functional dependency constraints also passes the functional correctness test.

Timing. The last, more elaborate viewpoint we consider for our example is timing. Figure 6 shows the task graph corresponding to one of the possible functional configurations. The second possible task graph
is obtained by swapping tasks or1 and or2. The values indicated next to the tasks are their worst-case execution time and therefore a property of the task and not of the task graph.

A simple load analysis, performed, e. g., using pyCPA\(^2\), will reject the functional configuration shown in Figure 6. The reason is that the task chain corresponding to the lane assist function is activated every 100 time units. If or1 is part of this chain, then that chain requires 90% of the available CPU time. However, the park assist chain has a 15% load (i. e., 30 time units every 200 time units). This means that no matter what the priority assignment is, the processor will be overloaded. The only option consists in swapping or1 and or2 in the functional configuration.

Our objective is now to find a priority assignment that satisfies the specified latency constraints. Remember that tasks inherit their priority from the threads to which they belong. In addition, we do not assign the same priority to multiple threads as this typically only adds more pessimism to the analysis result.

Let us focus on the end-to-end latency requirements imposed by P and L, which are respectively 150

\(^2\)https://pycpa.readthedocs.org
and 75 time units. The constraint on P is relatively loose and in fact it will be satisfied even if the park assist function is blocked by the complete lane assist task chain. Therefore, giving higher priority to the tasks involved in the lane assist function will always provide a feasible configuration. Interestingly, it is also possible to establish that the lane assist chain cannot afford to be blocked by the or1 task. In that case, a constraint stating that or1 must have lower priority than all the tasks involved in the lane assist function can be added to restrict the configuration space.

4.4 Discussion

The main purpose of our example was to illustrate how the contract negotiation process works, and in particular how incompatibility in one viewpoint can result in additional constraints to the configuration space. Therefore we kept the example quite simple. Moreover, note that the methodology is not yet fully implemented: the functional dependency analysis exists but it is not connected to the timing analysis engine at the moment. We currently mainly focus on providing useful timing analysis feedback.

5 Related work

Design of complex systems of components using contracts, or interfaces, has been widely studied since it was introduced by de Alfaro and Henzinger in [1]. A great variety of interface theories have been developed, which mostly focus on incremental design.

There has been active research lately on contract theories dealing with multiple viewpoints. For example, Reineke et al. focus on issues such as consistency between viewpoints [13]: if two viewpoints have some degree of overlap, how is it possible to guarantee that they do not contradict each other? Persson et al. give a high-level classification of model-based approaches to multi-view systems, which also discusses additional challenges of multi-view modeling [11]: traceability of information between
views, reuse, automation, change propagation and extensibility. In [10] Panunzio et al. give a more
applied contribution as they propose a component model for separation of concerns backed by case
studies originating from the European Space Agency. However, none of these results address dynamicity.

Another line of work related to this paper regards languages for contracts. The BCL language [5]
has now been used in many safety-critical industrial applications. An alternative is the contract language
from [4] which is based on temporal logic. Note however that our interest is less in the language aspects
than in the methodology.

Changing constrained systems are in the focus of the FRESCOR project [16], in which contracts are
used to ensure Quality of Service on the network-layer. However, the project focuses on bandwidth and
latency constraints usually found in multimedia applications. In this context contracts may be broken
(at times), enabling the test of novel configurations during runtime. This is not acceptable when updates
have an impact also on safety-critical functions such as driver assistance systems in a car. The EPOC
project [9] proposed for the first time a contract-based admission control framework for safety-critical
systems, but it was restricted to timing aspects and could not handle multiple viewpoints.

6 Conclusion

It has become increasingly difficult to test and verify software updates for embedded systems, in partic-
ular because of the complex platform dependencies which exist between software components. In this
paper we have presented a methodology to split the necessary effort between lab and field.

Performing in-field formal analysis to determine whether an update may be safely applied is chal-
 lenging because it requires an automated process able to handle multiple viewpoints such as functional
correctness, timing, etc. We have proposed an original methodology based on contract negotiation which
enables this. In particular, instead of addressing all viewpoints together we rely on viewpoint-specific
analysis engines, each of which uses a model that is best adapted to the particular viewpoint it is dealing
with. A second specificity of our approach is that we rely on added capabilities of these viewpoint-
specific analysis engines.

Finally, we have illustrated the benefits of our approach on a realistic scenario from the automotive
domain, which we unfold from the specification of contracts until a safe update has been found.

References

(2001), http://doi.acm.org/10.1145/503209.503226


contract-based specification and design. In: Formal Methods for Components and Objects, 6th International
225 (2007), http://dx.doi.org/10.1007/978-3-540-92188-2_9


guage for embedded systems. In: Proceedings of the 2014 IEEE Emerging Technology and Factory Automation,


Monitoring Assumptions in Assume-Guarantee Contracts

Oleg Sokolsky  Teng Zhang  Insup Lee  Michael McDougall
University of Pennsylvania  Philadelphia, PA, USA
GrammaTech, Inc.  Ithaca, NY, USA

Pre-deployment verification of software components with respect to behavioral specifications in the assume-guarantee form does not, in general, guarantee absence of errors at run time. This is because assumptions about the environment cannot be discharged until the environment is fixed. An intuitive approach is to complement pre-deployment verification of guarantees, up to the assumptions, with post-deployment monitoring of environment behavior to check that the assumptions are satisfied at run time. Such a monitor is typically implemented by instrumenting the application code of the component. An additional challenge for the monitoring step is that environment behaviors are typically obtained through an I/O library, which may alter the component’s view of the input format. This transformation requires us to introduce a second pre-deployment verification step to ensure that alarms raised by the monitor would indeed correspond to violations of the environment assumptions. In this paper, we describe an approach for constructing monitors and verifying them against the component assumption. We also discuss limitations of instrumentation-based monitoring and potential ways to overcome it.

1 Introduction

Behavioral specifications of software components are often written as assume-guarantee contracts $A \Rightarrow G$, where the assumption $A$ describes constraints on acceptable behaviors of the environment and the guarantee $G$ relates well-formed environment behaviors to component behaviors (as in, e.g., [3]). This approach allows us to develop components without complete knowledge of their environment. However, in an open environment, error-free execution cannot be fully assured before deployment. While the guarantee can be verified at design time, the assumption describes the property of the environment and thus has to be checked at run time. A natural approach is to complement pre-deployment verification of the contract with post-deployment monitoring of the assumption. The combination of contract verification and monitoring, effectively, turns component specification into $A \Rightarrow G \land \neg A \Rightarrow \text{Alarm}$.

Our approach to monitoring is based on instrumenting the application code of the component.\(^1\) We expect that the application code makes use of some I/O library to communicate with its environment. We assume that the library supports an API with well-known semantics of each API call. At the same time, we assume that the implementation of the I/O library is a black box, and therefore we can apply instrumentation in the application code, but not in the library code. As a result, we do not monitor the assumption directly, but mediated by the I/O API. We refer to the property being monitored on the application code as the internal assumption of the component. This separation between the internal assumption and the contract assumption brings up the question whether the monitor, constructed from the application code perspective, will correctly detect deviations from the assumption. Answering this question necessitates the second pre-deployment verification step in our approach. At the very least, the monitor should not raise any alarms when inputs satisfy the assumption.

---

\(^*\)This work is supported in part by DARPA BRASS program under contract FA8750-16-C-0007 and by ONR SBIR contract N00014-15-C-0126.

\(^1\)Since monitoring is from the system perspective, we refer to environment observations as inputs.
The contribution of this paper is a way for constructing internal assumption monitors from the source code of the component application code and verifying their correctness with respect to the contract assumption. On the other hand, we show that not all deviations from the format can be detected by monitoring the internal assumption, and that the component may be robust to some deviations. A precise characterization of limitations of such monitors is left for future work.

We specify contract assumptions as extended finite state machines (EFSM) that reflect temporal relationships between inputs, as well as semantic constraints on inputs, captured as predicates on current and past values of inputs. We construct the monitors based on the framework for checking compatibility between producers and consumers, developed by Evan Driscoll in his doctoral work [4, 5]. His approach is based on modeling input and output data formats as visibly pushdown automata [1] enhanced with semantic constraints. The automata, and to some extent semantic constraints, can be extracted directly from the application code using static analysis techniques.

In Driscoll’s approach, conformance is based on checking language inclusion between models of producer and consumer data formats. We use a slight extension of this approach to verify conformance between the assumption in the contract and the monitor automaton. We then monitor inputs of the component with respect to the same automaton.

There exists work in the literature on software verification with respect to assume-guarantee contracts, for example, [6]. We do not consider this aspect of the problem further in this paper. Instead, we concentrate on the problem of deriving monitors from the application code and exploring their relationship to the assumption of the component.

The rest of the paper is organized as follows. Section 2 introduces the case study that we use as a running example throughout the paper. Section 3 describes the construction of monitors from the application source code and verification of monitors with respect to the assumption of the contract. Section 4 offers a discussion of the power of resulting monitors in its relation to the I/O API chosen by the application code developer.

2 Case study

In this section, we introduce the case study that motivates our approach. Consider a simple program for calculating travel distance from GPS coordinates. Input for the calculator is a stream of data points along the path. Input can be streamed directly from a sensor, from a pre-processor that, e.g., cleans up the data and performs noise reduction, or from a pre-recorded file.

The informal specification of the input format is as follows: 1) input is a sequence of data points; 2) each point is represented as a newline-terminated string; 3) a point consists of three whitespace-separated numbers, which provide timestamp, latitude, and longitude; 4) timestamp is a non-negative integer, latitude is a floating point number in range [-90,90], longitude is a floating point number in range [-180,180]; 5) points are in the non-decreasing order of timestamps. The specification can be captured
Listing 1: Simple distance calculator

```c
int main(int argc, char * argv[]) {
    int time;
    float lat, lon, last_lat, last_lon;
    float total_dist = 0.0;
    _Bool is_first_pt = 1;

    while (1) {
        int items_read = 0;
        items_read = scanf("%d", &time);
        if (items_read <= 0) break;
        items_read = scanf("%f", &lat);
        if (items_read <= 0) break;
        items_read = scanf("%f", &lon);
        if (items_read <= 0) break;
        ...
    }
    ...
}
```

by an extended finite state machine (EFSM) shown in Figure 1. Note that, for simplicity, we do not represent white space separators in the format specification, but keep the newline terminator to be able to illustrate our point.

Listing 1 shows a possible implementation of the calculator (giving only the input-handling code). It is easy to see that some of the specification is not directly monitorable on the code. In particular, the parser does not see line breaks, because the `scanf` function silently consumes white space. Similarly, the code does not see the the input data as points, but rather as independently read in values. The code makes an implicit assumption that values read upon invocations $3i + 1$, $3i + 2$, and $3i + 3$ of the `scanf` function constitute the $i$th point. This assumption happens to be satisfied by the input format. For comparison, consider a different implementation of the same calculator, shown in Listing 2. This version relies on the assumption that each point appears on its own line. It is also correct with respect to the format specification. Each of the two versions is robust to some deviations from the input format specification and vulnerable to others. We will show in Section 4 that the two versions are vulnerable to very different kinds of deviations.

## 3 Monitor construction and verification

### 3.1 Preliminaries and problem statement

In this section, we define assume guarantee contracts for components in terms of extended finite state machines.

Given a set of typed channels $C$, a trace $t$ over $C$ is a sequence $\langle c_1(v_1), c_2(v_2), ... \rangle$, where $c_i \in C$ and $v_i$ is the value transmitted over $c_i$ in step $i$. Let $T_C$ be the set of all traces over $C$. Given a trace $t \in T_C$, $t \downarrow C' \in T_{C'}$ is a projection of $t$ on $C'$ defined as a maximal subsequence of $t$, in which every $c_i \in C'$. 
Listing 2: Alternative calculator implementation

```
int main(int argc, char * argv[]) {
    int time;
    char* buffer = NULL;
    size_t bufsiz;
    float lat, lon, last_lat, last_lon;
    float total_dist = 0.0;
    _Bool is_first_pt = 1;

    while (1) {
        getline(&buffer, &bufsize, stdin);
        if (items_read == -1) break;
        items_read = sscanf(buffer, "%d%f%f
", &time, &lat, &lon);
        if (items_read < 3) break;
        free(buffer);
        buffer = NULL;
        ...}
    }
```

Given a set of variables V, a valuation of V is a mapping that associates a type-correct value to each variable V. Let \( \mathcal{G}^V \) be the set of all valuations for V.

A component \( \mathcal{C} \) is a tuple \( \langle I, O, P \rangle \), where I is a set of typed input channels, O a set of typed output channels, and \( P(I, O) \) is the component logic that consumes data from input channels and computes values to transmit on output channels. Semantics of a component is given by a set of traces \( T^\mathcal{C} \subseteq T_{I,U} \).

An extended finite state machine (EFSM) is a tuple \( \langle S, s_0, C, V, D_0, T \rangle \), where S is a set of locations with the designated start location \( s_0 \), an alphabet \( C \) is a set of channels, V is a set of variables, and \( D_0 \) is the initial valuation of V. T is the transition relation. Each transition, denoted \( s_1 \xrightarrow{a} s_2 \), is equipped with a guard g, which is a predicate over \( V \cup \{ c \} \) and an update a, which is a function \( \mathcal{G}^V \cup \{ c \} \rightarrow \mathcal{G}^V \).

A state of EFSM \( M = \langle S, s_0, C, V, D_0, T \rangle \) is \( \langle s, D \rangle \), where \( s \in S \) and \( D \in \mathcal{G}^V \). Semantics of M is given by a set of runs. A run of M is a sequence \( \langle s_0, D_0 \rangle, c_0(v_0), \langle s_1, D_1 \rangle, c_1(v_1), \ldots \), such that, for each i, \( s_i \xrightarrow{c_i} s_{i+1} \in T, D_i \cup \{ c_i \mapsto v_i \} \in g_i \), and \( D_{i+1} = a_i(D_i \cup \{ c_i \mapsto v_i \}) \). That is, valuation \( D_i \) in the source state together with the value on channel \( c_i \) satisfy the guard \( g_i \), and the action \( a_i \) transforms \( D_i \) into \( D_{i+1} \). A trace is the projection of a run on the channels, abstracting away the information about locations. The set of traces of M is denoted \( T^M \).

Given a component \( \mathcal{C} = \langle I, O, P \rangle \), an assume-guarantee contract \( \langle A, G \rangle \) consists of two EFSMs: the assumption EFSM A with the alphabet I and the guarantee EFSM G with the alphabet \( I \cup O \). \( \mathcal{C} \) satisfies the contract if, for every trace \( t \in T^\mathcal{C} \), \( t \downarrow I \in T^A \Rightarrow t \in T^G \).

We model component logic \( P(I, O) \) as a composition of three EFSMs: \( M_l \otimes M_p \otimes M_0 \).\(^2\) \( M_p \) represents application code of the component. Channels in \( M_p \) represent values returned by input API calls (denoted

\(^2\)Intuitively, \( \otimes \) is standard product with synchronization over common channels. We do not define it formally here, since it is never directly computed in our approach.
as \( I' \) and values passed to output API calls (respectively, \( O' \)). \( M_I \) represents input API calls, transforming environment inputs \( I \) into returned inputs \( I' \). \( M_O \), similarly, transforms \( O' \) into environment outputs \( O \).

**Problem statement.** An internal assumption of the application code in a component \( \mathcal{C} = \langle I, O, P \rangle \) with the contract \( \langle A, G \rangle \) is an EFSM over the set of channels \( I' \). Given EFSMs \( M_P, M_I, \) and \( A \), our goal is to construct the internal assumption \( M_A \) to satisfy the following two requirements: (1) every trace of \( M_P \), projected on \( I' \), is a trace of \( M_A \) and (2) every trace of \( A \) is a trace of \( M_I \otimes M_A \). We treat EFSM \( M_A \) as a monitor that raises an alarm whenever a given trace is not a trace of \( M_A \). In the next section, we show how to construct \( M_A \) and \( M_I \otimes M_A \).

### 3.2 Monitor construction and verification

**Monitor construction.** We use the approach of [5] to extract monitor skeletons — transitions of the EFSM. The algorithm given in [5] produces visibly pushdown automata [1]. However, to simplify presentation, we limit our attention to state machines in this paper, effectively assuming that the application code does not have procedure calls except for external API calls to black-box libraries.

The idea behind the extraction algorithm is to replace calls to I/O APIs with transitions labeled by the type of data item returned by each call. Following [5], we limit our attention to C types \( \text{int} \) and \( \text{float} \). We also allow API calls to produce multiple data items. Many standard C library calls are parameterized by a *format string*. If multiple data items are produced, we introduce a sequence of transitions to match items in the format string in the order of their appearance. For example, a call to `scanf()` with the format string \("%d%f"\) introduces two transitions: \( \text{int} \rightarrow \text{float} \). We refer to this sequence as the *skeleton fragment* of the API call.

The algorithm for constructing the monitor skeleton operates on the control flow graph of the component application code. If the node in the graph corresponds to a call to an I/O API procedure, the transition from the node is replaced with the skeleton fragment of the API call. All other transitions become \( \varepsilon \)-transitions and are removed from the skeleton in the standard way.

Once the skeleton is constructed, it is turned into an EFSM of the internal assumption by adding state variables, guards, and updates representing semantic constraints. In general, this is a manual process, since not all constraints are explicitly present in the code. The main states of the process are as follows. The first step is to turn type labels into channel labels. We conjecture that, in many cases, we may be able to use names of variables in the application code that receive the data items. If the channel value is used in semantic constraints that involve comparison of multiple values, we add an action that saves the received value in a variable of the EFSM. The channel value is used in semantic constraints that involve comparison of multiple values, we add an action that saves the received value in a variable of the EFSM. We also add a guard that is a conjunction of predicates representing semantic constraints that use the channel value. If the predicates involve other values, they are taken from the respective variables of the EFSM.

Figure 2, a) shows the skeleton extracted from the code of the case study from Listing 1, while Figure 2, b) shows the internal assumption EFSM after adding semantic constraints of the format. Note that the same EFSM would be constructed for the code in Listing 2.

At run time, we use the internal assumption EFSM as constructed above to monitor inputs read by the component application code. Monitoring can be performed using runtime verification tools such as [2, 7, 8].

Before monitoring, we still need to check that the internal assumption EFSM is correct with respect to the assumption of the component. In the approach of [5], extracted representation of the input format is verified against a similarly extracted output format of the producer by checking language containment:
the language of the producer automaton should be included in the language of the consumer automaton.

In our case, we perform similar verification of the extracted internal assumption against the EFSM representing the assumption of the contract. Before we do this, we need to modify the internal assumption EFSM to account for the semantics of I/O API calls.

**Modeling I/O API effects.** We extend the approach of [5] with the ability to represent effects of I/O APIs used by the application code. An important effect of an I/O API is that it can make some aspects of the input format unobservable. Here, we consider two kinds of I/O API and compare their effects. First, consider the `scanf()` call used by the code in Listing 1. Each call is parameterized by a format string, which is a sequence \(\langle t_1, t_2, \ldots \rangle\) of types. The procedure skips over white space, including newline characters, preceding each item in the format string. To represent this effect, we add a self-loop transition labeled by the `newline` symbol\(^3\) to each state that occurs within the sequence.

Now consider a different way to input data into the application code. Input is first read into a string using the `getline` call, which is then processed using the `sscanf` call. For the purpose of this paper, we consider this pattern as a single API call. In this case, we know that the white space skipped by the call does not involve any line breaks, thus, there is no need to add the self-loop transition before each item in the format string. However, any data item in the string beyond those listed in the format string is not read. However, we know that the line break has been read when the call returns\(^4\). Thus the sequence of transitions corresponding to a format string is extended by a self-loop transition for each non-newline symbol, and the sequence is extended with the newline-labeled transition at the end. Figure 3 shows extended skeleton fragments corresponding to the “%d%f” format string for the two cases.

Another important effect of the I/O API is type conversion performed by I/O routines. For example, “%f” directive will read an integer and convert it to the `float` type. To capture this effect, we can make more complex skeleton fragments, containing alternative transitions. We do not model type conversions in this paper.

---

\(^3\)Recall from Section 2 that we are representing line breaks in the input format but not other kinds of white space.

\(^4\)For simplicity, we do not consider the end-of-file case here.
Monitor verification. In order to obtain the EFSM of the internal assumption that incorporates effects of the I/O API we repeat the construction outlined in the algorithm above. Except now, in the skeleton extraction algorithm outlined above, instead of the skeleton fragment of the call, we use the extended fragments as described above.

Now we can check language inclusion between the two EFSMs. This would ensure that no input that satisfies the assumption of the component will trigger a violation of the internal assumption of the code. Internal assumptions for both versions of the calculator are correct in this respect.

4 Discussion

Monitor construction. In this paper, we explored an approach to extract monitor skeletons from the application code using static analysis. The extraction process, which inserts a state machine fragment for every I/O API call, allows us to incorporate API effects for monitor verification with respect to the assumption, by inserting a different state machine fragment. The extraction process also identifies code locations that need to be instrumented in order to perform monitoring at run time. Note that this process differs from most approaches to runtime verification [2, 7, 8], where monitors are developed independently of the code, based on requirements. It is still possible, of course, to construct monitors independently. The modification of the monitor to incorporate API effects for monitor verification would have to be done manually, too.

Detection power. What we need to consider is the detection power of monitors described above. The first point to notice is that, since monitors are extracted from the code, there can be no structural violations. By this we mean the case when an event $e$ is observed, but the monitor state machine does not expect $e$ to happen in the current state. Thus, the monitor can detect only violations of semantic constraints. If constraints are too weak, deviations will go undetected.

As mentioned above, monitors do not observe all deviations from the input format specification. However, the application code may be robust to some of the changes, effectively shielded by the choice of the I/O API. Other changes lead to incorrect computations.

To illustrate this with our case study, consider two deviations from the input format specification. One is to introduce a new field in each point. This may correspond to a case where the sensor in the system is upgraded, and now produces altitude readings in addition to latitude and longitude. We refer to this deviation as D1. Another deviation, D2, is the removal of line breaks, which makes sense in a streaming context. The first implementation of the calculator is robust to D2, and D1 is promptly detected by the monitor: the code attempts to read the altitude field as the next timestamp and then the timestamp as the latitude value, and one of the associated semantic constraints is likely to be violated. Thus, not only is the error detected, but we get a stream of alarms that indicates that it is not an isolated format glitch, but a systematic problem. The second implementation, on the other hand, is robust to D1: the whole point is read in, but the altitude field is not part of the format string and is never processed. However, D2 is not detected, all points are read together as a single line, and only the first point is input. No semantic constraints are violated, but the calculator produces a wrong result. With stronger semantic constraints, this deviation could be detected. For example, the input format could contain a field at the end of the stream, specifying the number of points in the input. We would be able to check, then, that no points have been lost.

Ideally, we would like to design monitors that detect all deviations that make the code produce a wrong result and do not raise alarms when the component is robust to an observed deviation. To achieve
this, we need to characterize (1) which deviations can be detected and which cannot be, and (2) which deviations are harmful and which are not. To address question (1), we can look at the language difference between the two EFSMs. That is, compute the set of runs that are rejected by the component assumption, but are accepted by the internal assumption of the application code. These runs correspond to deviations from the format that are not detected. The answer to question (2) seems harder.

Finally, we note that an alternative to instrumenting application code would be to instrument the I/O library, which we assumed to be a black box in this paper. Such instrumentation would allow the monitor to have an unmediated view of input data, allowing us to check the input format specification directly. On the other hand, instrumentation would have to be done at a much lower level of abstraction and require intimate understanding of how the library works.

5 Conclusion

We presented an approach for combining static, pre-deployment verification of assume-guarantee contracts for software components with dynamic, post-deployment monitoring of the contract assumption. We identified a semantic gap between the contract assumption and the internal assumption of the application code of the component. Unlike the contract assumption, which cannot be monitored directly, the internal assumption can be monitored by instrumenting application code. We showed that, due to the semantic gap, not all violations of the assumption can be detected. Our future work will concentrate on automated characterization of the semantic gap for a given contract assumption and I/O API used by the application code, in order to identify which critical deviations from the contract assumption can be missed by the monitor.

References

Preliminary Results Towards Contract Monitorability

Annalizz Vella
CS, ICT, University of Malta
annalizz.vella.10@um.edu.mt

Adrian Francalanza
CS, ICT, University of Malta
adrian.francalanza@um.edu.mt

This paper discusses preliminary investigations on the monitorability of contracts for web service descriptions. There are settings where servers do not guarantee statically whether they satisfy some specified contract, which forces the client (i.e., the entity interacting with the server) to perform dynamic checks. This scenario may be viewed as an instance of Runtime Verification, where a pertinent question is whether contracts can be monitored for adequately at runtime, otherwise stated as the monitorability of contracts. We consider a simple language of finitary contracts describing both clients and servers, and develop a formal framework that describes server contract monitoring. We define monitor properties that potentially contribute towards a comprehensive notion of contract monitorability and show that our simple contract language satisfies these properties.

1 Introduction

Web services [7, 6] typically consist of two types of computing entities. Servers offer ranges of sequences of service interactions to clients, which in turn interact with these services and occasionally reach a state denoting client satisfaction. The service interactions offered by a server typically follow some predefined structure that may be formalised as a contract [6, 7, 15, 3]. Dually, the service interactions invoked by a client may also be expressed within the same formalism.

The contract calculus defined in [15, 2, 5] is an abstract formalism equipped with an operational semantics that provides an implementation-agnostic, high-level description of client-server interactions; this permits formal reasoning about web services, such as whether a client is compatible with a server or whether a server is able to satisfy the service interactions requested by the client. Such reasoning may, for instance, be used by clients for dynamic service discovery, where a client decides to interact with a server whenever the contract it advertises satisfies the requirements of the client.

Example 1.1. Consider the contract below describing the behaviour of an internet banking server:

\[
\text{login}((\text{valid}.(\text{query} \cdot 0 + \text{transfer} \cdot 0)) \oplus (\text{invalid} \cdot 0))
\]

It states that the server first expects a login service interaction followed by either a valid or invalid service invocation; the operator \(\oplus\) denotes that the server decides autonomously whether to invoke valid or invalid in response. If it branches to the latter, it terminates all interactions, denoted by 0. However, if it internally decides to invoke the service interaction valid, it then offers a choice (denoted by the symbol +) of service interactions: it either accepts (account balance) query interactions or else (fund) transfer interactions. A contract describing the behaviour of a possible bank client is given below:

\[
\text{login}((\text{invalid}.\text{reason} \cdot 1 + (\text{expired} \cdot 1) + (\text{valid}.\text{query} \cdot 1))
\]

After a login service invocation, this client expects either of three responses: an invalid interaction prompting another service request that asks for a reason why the login was invalid, a login expired
invocation or else a valid login interaction that is followed by invoking a query service request. All these alternative sequences leave the client in a satisfied state. By analysing the resp. contracts, one can deduce that interactions on the valid service following a client login interaction necessarily lead to a query interaction, which then leaves the client satisfied. One can also discern that invalid interactions lead to a deadlock, whereby the client asks for a reason service that is not offered by the server. One can also note that the expired option offered by the client is never chosen by the server.

Within this framework, there still remains the question of whether a service behaviour actually adheres to the contract it advertises. In general, static techniques (such as session-based type systems [9], or state-based model-checking of compliance, must or fair testing inclusion [15, 2, 5]) are used to verify before deployment whether a server implementation respects the contract that describes it. However, there are cases where this solution is not applicable. For instance, the client may decide not to trust the static verifier used by the server. Alternatively, in a dynamic setting where service components are downloaded and installed at runtime, pre-deployment checks cannot be made on the server implementation since some components only become available for inspection at runtime. There are also cases whereby a server does not come equipped with a formal description at all.

In these circumstances, a client can check that a server respects an advertised (or expected) contract by analysing the behaviour exhibited by the server at runtime. There are a number of cases where such a solution is adopted [4, 12], making use of dynamic monitoring, possibly in conjunction with other verification techniques. This monitoring of systems may be seen as an instance of Runtime Verification (RV) [13], a lightweight formal verification technique used to check the current execution of a program by verifying it against some properties. In a typical setup, the monitor observing the running system raises a flag when a conclusive verdict is reached, denoting that the property being checked for is either satisfied or violated.

An important question in any RV setup is that of the monitorability of the specification language considered. Indeed, it is generally the case that not all aspects of a specification can be monitored for and determined at runtime, as shown in [8, 1, 11] for specification languages such as LTL and the modal μ-calculus. In this work, we start to investigate the monitorability of contracts which, in turn, sheds light on the viability and expressiveness of the dynamic contract checking setup discussed above. In contrast to earlier work on monitorability, we do not rely on an external formal logic for specifying the properties expected by a server contract, e.g., a satisfaction relation \( p \models \phi \) where \( \phi \) would be a formula from a logic defined over server contract \( p \) through the semantic relation \( \models \). Instead, we use the subcontract server relation \( q \sqsubseteq_{\text{SRV}} p \) defined in [15] as a refinement semantic relation where \( q \) is an abstract description of the expected properties of a server contract \( p \), thus using the contract language itself as a specification language. Within this setting, we investigate whether our monitoring mechanism is expressive enough to verify whether a server \( p \) indeed refines an abstract description \( q \).

The rest of the paper is structured as follows. Section 2 overviews our contract language and defines our notion of contract satisfaction. Section 3 introduces our monitoring setup and Section 4 relates verdicts reached by our monitored computations to the contract satisfactions discussed in Section 2. Section 5 concludes by discussing related and future work.

## 2 Servers, Clients and Satisfaction

Figure 1 describes the syntax and semantics of (finite) servers and clients. Let \( a, b, c, d, \ldots \in \text{NAMES} \) be a set of names denoting interaction addresses. Let \( \overline{\cdot} \) be a complementation operation on these names where we refer to the complement of \( a \) as \( \overline{a} \); the operation is an involution, where \( \overline{\overline{a}} = a \). The set of actions
Preliminary Results Towards Contract Monitorability

Syntax

\[ p, q \in \text{SRV} ::= 0 \quad \text{(inaction)} \quad | \quad \alpha.p \quad \text{(prefixing)} \]
\[ | \quad p + q \quad \text{(external choice)} \quad | \quad p \oplus q \quad \text{(internal choice)} \]

\[ r, s \in \text{CL1} ::= 0 \quad | \quad \alpha.r \quad | \quad r + s \quad | \quad r \oplus s \quad | \quad 1 \quad \text{(success)} \]

Dynamics

\[
\begin{array}{c}
\text{ACT} \\
\alpha.p \xrightarrow{\alpha} p \\
\text{SELL} \\
p \xrightarrow{\mu} p' \quad p + q \xrightarrow{\mu} p' \\
\text{CHOL} \\
p \oplus q \xrightarrow{\tau} p \\
\text{CHOR} \\
p \oplus q \xrightarrow{\tau} q
\end{array}
\]

Interaction

\[
\begin{array}{c}
\text{ASYS} \\
\tau \xrightarrow{r} q \quad | \quad r \parallel p \xrightarrow{\tau} r \parallel q \\
\text{ASYC} \\
r \xrightarrow{\tau} s \quad | \quad r \parallel p \xrightarrow{\tau} s \parallel p \\
\text{SYN} \\
r \xrightarrow{\tau} s \quad | \quad p \xrightarrow{\alpha} q
\end{array}
\]

Figure 1: Server and Client Syntax and Semantics

\( \alpha \in \text{ACT} = (\text{NAMES} \cup \{\overline{a} \mid a \in \text{NAMES}\}) \) includes all names and their complement. Let \( \tau \) be a distinct action not in \( \text{ACT} \) denoting internal unobservable activity, where we let \( \mu \in \text{ACT} \cup \{\tau\} \).

Servers, \( p, q \in \text{SRV} \), consist of either the terminated server \( 0 \), a prefixed server \( \alpha.p \) that first engages in interaction \( \alpha \) and then behaves as \( p \), an external choice \( p + q \) that can either behave as \( p \) or \( q \) depending on the interactions it engages in, or an internal choice \( p \oplus q \) that autonomously decides to either behave as \( p \) or \( q \). Clients, \( r, s \in \text{CL1} \), have a similar structure but may also consist of the term \( 1 \) denoting contract fulfilment. The semantics of both servers and clients are given in terms of a Labelled Transition System (LTS) where the labelled transition relation \( p \xrightarrow{\mu} q \) is defined as the least relation satisfying the rules in Figure 1; the definition of the transition relation for clients \( r \xrightarrow{\mu} s \) is analogous and thus elided. The definition is standard and follows that of related languages such as CCS [14]. For instance, the term \( \alpha.p \) transitions with (action) label \( \alpha \) to the continuation \( p \); if \( p \) can engage in an interaction on \( \mu \) and transition to \( p' \), then an external choice term involving \( p \), e.g., \( p + q \) may also transition to \( p' \) after exhibiting action \( \mu \); by contrast, an internal choice involving \( p \), e.g., \( p \oplus q \) may transition to \( p \) without exhibiting an external action (\( \tau \) is used).

Servers and clients may be composed together to form a system, \( r \parallel p \), so as to engage in a sequence of interactions. Interactions are also defined as an LTS over systems, through the rules ASYS, ASYC and SYN in Figure 1. As is standard, silent transitions by either server or client allow them to transition autonomously in a system. However, a client transition on an external action must be matched by a server transition on the (dual) co-action for the transition to occur in the resp. system, denoting client-server interaction. Computations are sequences of system transitions \( r_0 \parallel p_0 \xrightarrow{\tau} \ldots \xrightarrow{\tau} r_n \parallel p_n \), denoted as \( r_0 \parallel p_0 \Rightarrow r_n \parallel p_n \); the sequence may be potentially empty, \( n = 0 \), where no transitions are made, in which case we have \( r_0 = r_n \) and \( p_0 = p_n \). A computation \( r_0 \parallel p_0 \Rightarrow r_n \parallel p_n \) is maximal whenever
A maximal computation, \( r \parallel p \Rightarrow s \parallel q \), is successful, whenever the client’s contract is fulfilled, meaning that \( s = 1 \). A service \( p \) satisfies a client \( r \), denoted as \( \text{sat}(p, r) \), when every maximal computation rooted at \( r \parallel p \) is successful.

**Example 2.2.** The server \( p = \overline{\tau}.0 + (b.a.0 \oplus c.0) \) may either transition as \( p \xrightarrow{\tau} 0 \) using rules \text{ACT} and \text{SELL} from Figure 1, or silently transition as \( p \xrightarrow{\tau} b.a.0 \) or \( p \xrightarrow{\tau} c.0 \) via rules \text{CHOL}, \text{CHOR} and \text{SELR} from Figure 1. It satisfies the client \( r = \overline{\tau}.1 + \overline{\tau}.1 \), denoted as \( \text{sat}(p, r) \), because the only maximal computations possible are the following

\[
\begin{align*}
    r \parallel p & \xrightarrow{\tau} r \parallel b.a.0 \xrightarrow{\tau} 1 \parallel a.0 \\
    r \parallel p & \xrightarrow{\tau} r \parallel c.0 \xrightarrow{\tau} 1 \parallel 0
\end{align*}
\]

both of which are successful. By contrast, server \( p \) does not satisfy client \( \overline{\tau}.1 \), denoted as \( \neg \text{sat}(p, \overline{\tau}.1) \), nor does it satisfy the clients \( \overline{\tau}.1 + \overline{\tau}.0 + \overline{\tau}.1 \) and \( \overline{\tau}.c.1 + \overline{\tau}.1 \). In each case, we can show this through the unsuccessful maximal computations below.

\[
\begin{align*}
    \overline{\tau}.1 \parallel p & \xrightarrow{\tau} \overline{\tau}.1 \parallel c.0 \\
    \overline{\tau}.1 + \overline{\tau}.0 + \overline{\tau}.1 \parallel p & \xrightarrow{\tau} \overline{\tau}.1 + \overline{\tau}.0 + \overline{\tau}.1 \parallel b.0 \xrightarrow{\tau} 0 \parallel 0 \\
    \overline{\tau}.c.1 + \overline{\tau}.1 \parallel p & \xrightarrow{\tau} \overline{\tau}.c.1 + \overline{\tau}.1 \parallel b.0 \xrightarrow{\tau} c.1 \parallel 0
\end{align*}
\]

The satisfaction predicate \( \text{sat}(\cdot, \cdot) \) induces a natural preorder amongst servers.

**Definition 2.3** (Server Preorder [15]). A server \( p \) is a subcontract of server \( q \), denoted as \( p \sqsubseteq_{\text{SRV}} q \), whenever, for all clients \( r \), \( \text{sat}(p, r) \) implies \( \text{sat}(q, r) \). Dually, \( q \) is referred to as a supercontract of \( p \).

Intuitively, \( p \sqsubseteq_{\text{SRV}} q \) of Definition 2.3 means that we can substitute a server \( p \) by a server \( q \), safe in the knowledge that any client satisfied by \( p \) would not be affected.

**Example 2.4.** Definition 2.3 allows us to establish a number of useful server (in)equalities such as

\[
\begin{align*}
    \overline{\tau}.0 \oplus b.0 & \sqsubseteq_{\text{SRV}} \overline{\tau}.0 \\
    b.a.0 + b.c.0 & \sqsubseteq_{\text{SRV}} b.(a.0 \oplus c.0) \\
    b.(a.0 \oplus c.0) & \sqsubseteq_{\text{SRV}} b.a.0 + b.c.0
\end{align*}
\]

but also justify subtle cases where substituting one server for another might break client satisfaction. For instance, we have \( 0 \not\sqsubseteq_{\text{SRV}} a.0 \) because for the client \( (1 \oplus 1) + \overline{\tau}.0 \) we have \( \text{sat}(0, (1 \oplus 1) + \overline{\tau}.0) \) since \( (1 \oplus 1) + \overline{\tau}.0 \parallel 0 \xrightarrow{\tau} 1 \parallel 0 \) is the only maximal computation (which is also successful), but also have \( \neg \text{sat}(a.0, (1 \oplus 1) + \overline{\tau}.0) \) due to the unsuccessful computation \( (1 \oplus 1) + \overline{\tau}.0 \parallel a.0 \xrightarrow{\tau} 0 \parallel 0 \).

### 3 Monitors and Monitored Computations

Figure 2 describes the monitoring framework used to analyse servers purporting to adhere to some advertised contract. It defines the syntax of these monitors, which follow the general structure used in earlier works [11, 1] whereby monitors may reach any one of the three verdicts \( \text{VERD} \), namely acceptance, rejection, or the inconclusive verdict. In addition to the basic prefixing patterns used in [11, 10], we here also use action complementation, \( \underline{\alpha} \), to denote any action apart from \( \alpha \). As in [11, 10], a monitor is allowed to branch, \( m + n \), depending on the actions observed at runtime. We also find it convenient to express a merge monitor operator that facilitates the composition of monitor specifications, \( m \times n \).

The semantics of a monitor is given in terms of the LTS defined by the rules in Figure 2. This is best viewed as the evolution of a monitor in response to a (finite) execution trace \( t \in \text{ACT}^* \), consisting of a sequence of actions \( \alpha_1, \ldots, \alpha_n \). Verdicts are irrevocable when reached, and do not change upon viewing
Preliminary Results Towards Contract Monitorability

Syntax

\[ v, u \in \text{VERD} ::= Y \quad \text{(acceptance)} \quad | \quad N \quad \text{(rejection)} \]
\[ | \quad \text{end} \quad \text{(inconclusive)} \]
\[ \theta \in \text{PATTERNS} ::= \alpha \quad \text{(action)} \quad | \quad \alpha \quad \text{(complement)} \]
\[ m, n \in \text{MON} ::= v \quad \text{(verdict)} \quad | \quad \theta \cdot m \quad \text{(interaction)} \]
\[ | \quad m + n \quad \text{(choice)} \quad | \quad m \times n \quad \text{(conjunction)} \]

Dynamics

\[ \text{MVER} \quad v \quad \alpha \rightarrow \quad v \]
\[ \text{MACT} \quad \alpha \cdot m \quad \alpha \rightarrow \quad m \]
\[ \text{MNACT} \quad \beta \neq \alpha \quad \alpha \cdot m \quad \beta \rightarrow \quad m \]
\[ \text{MSELL} \quad m \quad \alpha \rightarrow \quad m' \quad m + n \quad \alpha \rightarrow \quad m' \]
\[ \text{MSELR} \quad n \quad \alpha \rightarrow \quad n' \quad m + n \quad \alpha \rightarrow \quad n' \]
\[ \text{MCONJ} \quad m \quad \alpha \rightarrow \quad m' \quad n \quad \alpha \rightarrow \quad n' \quad m \times n \quad \alpha \rightarrow \quad m' \times n' \]

Instrumentation

\[ \text{iMON} \quad p \quad \alpha \rightarrow \quad p' \quad m \quad \alpha \rightarrow \quad m' \quad m < p \quad \alpha \rightarrow \quad m' < p' \]
\[ \text{iTER} \quad p \quad \alpha \rightarrow \quad p' \quad m \quad \alpha \rightarrow \quad m' \quad m < p \quad \alpha \rightarrow \end{\quad \text{p} \quad m < p \quad \tau \rightarrow \quad m < p' \]
\[ \text{iASY} \quad p \quad \tau \rightarrow \quad p' \quad m < p \quad \tau \rightarrow \quad m < p' \]

Figure 2: Monitors and Instrumentation

Further actions in the trace (rule MVER). Prefixing releases the guarded monitor when the expected pattern is encountered (rules MACT and MNACT). The rules MSELL and MSELR describe left and right monitor branching as expected, whereas rule MCONJ describes the synchronous evolution of merged monitors.

A monitored server contract consists of a server \( p \) that is instrumented with a monitor \( m \), denoted as \( m < p \). The behaviour of monitored contracts is defined as an LTS through the rules stated in Figure 2, and relies on the resp. LTSs of the monitor and the server. Rule iMON states that if a server can transition with action \( \alpha \) and the monitor can follow this by transitioning with the same action, then in an instrumented server they transition in lockstep. However, if the monitor cannot follow such a transition the instrumentation forces it to terminate with an inconclusive verdict, end, while the process is allowed to proceed unaffected; see rule iTER. Finally, rule iASY allows a contract to evolve independently from the monitor when performing silent \( \tau \)-moves (which are unobservable to the monitor). We refer to a sequence of transitions from a monitored contract as a monitored computation and use the standard notation \( m < p \Rightarrow m' < p' \) that abstracts over \( \tau \)-moves in trace \( t \).

A few comments are in order. First, we highlight the fact that in the operational semantics for monitored systems of Figure 2, the monitor does not have access to the internal state of the server generating the trace, and its observations are limited to the execution that the server chooses to exhibit at runtime. This is meant to model the RV scenarios mentioned in Section 1, where the source of the executing system cannot be analysed: from the point of view of the runtime monitoring and verification, the server description is merely used to generate traces. Second, we note that, in a monitored server setup, any visible behaviour is instigated by the server, relegating the instrumented monitor to a passive role.
that merely follows the server actions. Stated otherwise, the server drives the behaviour in a monitored system and dictates the execution path that the monitor can analyse at runtime.

In what follows, we explain how monitors work through a series of examples. The exposition focuses on monitors that produce rejection verdicts, but the discussion can be extended to acceptance verdicts in a straightforward manner.

**Example 3.1.** The monitor \( \overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N \) checks for violations from contracts that are expected to adhere to (i.e., be supercontracts of) the contract \( \overline{\alpha} {\cdot} b {\cdot} 0 \). In fact, the monitor reaches a rejection verdict whenever a contract either emits an action that is not \( \overline{\alpha} \) at runtime, \( \overline{\alpha} {\cdot} N \), or else emits an action that is not \( b \) following action \( a \), \( \overline{\alpha} {\cdot} b {\cdot} N \). Consider the server \( \overline{\alpha} {\cdot} c {\cdot} 0 \); when instrumented with our monitor we can observe the following monitored computation whereby the monitor reaches a rejection verdict, \( N \).

\[
(\overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N) \triangleleft (\overline{\alpha} {\cdot} c {\cdot} 0) \quad \xrightarrow{\pi} \quad b {\cdot} N \triangleleft c {\cdot} 0 \quad \xrightarrow{c} \quad N \triangleleft 0
\]

By contrast, when the server \( \overline{\alpha} {\cdot} b {\cdot} 0 \) is instrumented with the monitor, no rejection verdict is reached; in particular, the final transition below is derived using rule \( \text{TER} \) because \( \overline{\alpha} {\cdot} b {\cdot} N \nvdash b {\cdot} 0 \).

\[
(\overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N) \triangleleft (\overline{\alpha} {\cdot} b {\cdot} 0) \quad \xrightarrow{\pi} \quad b {\cdot} N \triangleleft b {\cdot} 0 \quad \xrightarrow{b} \quad \text{end} \triangleleft 0
\]

We emphasise the fact that monitor termination through rule \( \text{TER} \) is crucial to avoid unwanted detections. Consider a variant of the earlier monitor, \( \overline{\alpha} {\cdot} b {\cdot} N \), which now reports violations whenever it observes the trace consisting of the action \( \overline{\alpha} \) followed by the action \( b \). When composed with the system \( \overline{\alpha} {\cdot} c {\cdot} b {\cdot} 0 \) we observe the following monitored computation.

\[
\overline{\alpha} {\cdot} b {\cdot} N \triangleleft \overline{\alpha} {\cdot} c {\cdot} b {\cdot} 0 \quad \xrightarrow{\pi} \quad b {\cdot} N \triangleleft c {\cdot} b {\cdot} 0
\]

\[
\xrightarrow{c} \quad \text{end} \triangleleft b {\cdot} 0
\]

\[
\xrightarrow{b} \quad \text{end} \triangleleft 0
\]

At transition (**) the server can perform an action, \( c \), that the monitor is not able to follow (i.e., it is not specified how the monitor should behave at that point should it observe action \( c \)). Accordingly, the semantics instructs the monitor to terminate (prematurely) with an inconclusive verdict. There are two instrumentation alternatives that could have been adopted, both of which are arguably wrong from a monitoring perspective. The first option would have been to prohibit the server from exhibiting action \( c \), which goes against the tenet that the monitor should adopt a passive role and not interfere with the execution of the program it monitors. The second option is arguably even worse: we could have let the server transition and left the monitor in its present state, i.e., \( b {\cdot} N \triangleleft c {\cdot} b {\cdot} 0 \xrightarrow{c} b {\cdot} N \triangleleft b {\cdot} 0 \), but then this would have led to an unspecified/erroneous detection at the next transition \( b {\cdot} N \triangleleft b {\cdot} 0 \xrightarrow{b} N \triangleleft 0 \).

**Example 3.2.** The server \( \overline{\alpha} {\cdot} b {\cdot} 0 \oplus c {\cdot} b {\cdot} 0 \) is not a supercontract of \( \overline{\alpha} {\cdot} b {\cdot} 0 \) according to Definition 2.3. Crucially, however, in an RV setting, monitor detection depends on the runtime behaviour exhibited by the server. This contrasts with other forms of verification which may be allowed to explore all the execution paths of a server under scrutiny.\(^1\)

\[
(\overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N) \triangleleft (\overline{\alpha} {\cdot} b {\cdot} 0 \oplus c {\cdot} b {\cdot} 0) \quad \xrightarrow{\pi} \quad (\overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N) \triangleleft (\overline{\alpha} {\cdot} b {\cdot} 0) \quad \xrightarrow{\pi} \quad b {\cdot} N \triangleleft b {\cdot} 0 \quad \xrightarrow{b} \quad \text{end} \triangleleft 0
\]

\[
(\overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N) \triangleleft (\overline{\alpha} {\cdot} b {\cdot} 0 \oplus c {\cdot} b {\cdot} 0) \quad \xrightarrow{\pi} \quad (\overline{\alpha} {\cdot} N + \overline{\alpha} {\cdot} b {\cdot} N) \triangleleft (c {\cdot} b {\cdot} 0) \quad \xrightarrow{c} \quad N \triangleleft b {\cdot} 0 \quad \xrightarrow{b} \quad N \triangleleft 0
\]

\(^1\)In the general case, a pre-deployment verification technique may also analyse infinite paths.
In the first monitored computation above, the server exhibits the behaviour described by the trace \( \overrightarrow{m} \), which prohibits the monitor from detecting any violations. However, the same server exhibits a different trace \( \equiv^{cb} \) in the second monitored computation which permits monitor detection. The rejection verdict is in fact reached after the first visible transition on action \( c \), and then preserved throughout the remainder of the computation.

Example 3.3. We can monitor for violations of the contract \( \overline{a}.b.0 + c.0 \) by composing two submonitors that monitor for the constituents. Specifically, since the monitor \( c.N + c.end \) checks for violations of contract \( c.0 \) and the minimally extended monitor \( \overline{a}.N + \overline{a}.(b.N + b.end) \) checks for violations of \( \overline{a}.b.0 \) as discussed in Example 3.1, we can construct the composite monitor \( (\overline{a}.N + \overline{a}.(b.N + b.end)) \times (c.N + c.end) \) to monitor for violations of \( \overline{a}.b.0 + c.0 \).

When the composite monitor is instrumented with the contract it is expected to monitor for, we note that it does not reach a rejection along every (parallel) submonitor:

\[
\left((\overline{a}.N + \overline{a}.(b.N + b.end)) \times (c.N + c.end)\right) \triangleleft b.0 + c.0 \quad \Rightarrow \quad b.N + b.end \times N \not\triangleleft b.0
\]

\[
\left((\overline{a}.N + \overline{a}.(b.N + b.end)) \times (c.N + c.end)\right) \triangleleft b.0 + c.0 \quad \Rightarrow \quad N \times end \not\triangleleft 0
\]

By contrast, the violating contract above generates a rejection along every submonitor.

Example 3.3 clearly suggests a definition of monitor rejection.

Definition 3.4 (Rejection). A monitor \( m \) is in a rejection state, denoted as \( \text{rej}(m) \), whenever it is of the form \( N \times \ldots \times N \). We overload this predicate to denote a server \( p \) being rejected by a monitor \( m \), defined formally as

\[
\text{rej}(p, m) \overset{def}{=} \exists t, p' \cdot m \triangleright p \triangleleft m' \triangleleft p' \text{ and } \text{rej}(m')
\]

Example 3.5. The monitor \( c.N + c.end \) rejects server \( b.0 \), \( \text{rej}(b.0, (c.N + c.end)) \) as well as server \( c.0 + b.0 \), \( \text{rej}((c.0 + b.0), (c.N + c.end)) \) because both may exhibit an execution trace that leads the monitor to a rejection state. By contrast, \( c.N + c.end \) does not reject server \( c.0 \). Recalling monitor \( m = (\overline{a}.N + \overline{a}.(b.N + b.end)) \times (c.N + c.end) \) from Example 3.3, we can also state that it rejects server \( b.0 \), \( \text{rej}(b.0, m) \).

4 Preliminary results towards Monitorability

Monitorability may be broadly described as the relationship between the properties of a logic specifying program behaviour and the detection capabilities of a monitoring setup instrumented over such programs. It is therefore parametric with respect to the logic and monitoring setup considered. In what follows, we sketch out preliminary investigations that focus on the monitor rejections defined in Section 3, and attempt to relate them to violations of the server preorder defined in Section 2.

We have already defined enough machinery to be able to state formally two important properties. Definition 4.1 states that a monitor \( m \) soundly monitors for a server contract \( p \) if and only if, whenever it rejects a server \( q \), it is indeed the case that \( q \) is not a supercontract of \( p \). In a sense, the dual of this is Definition 4.2, which states that a monitor \( m \) completely monitors for a server contract \( p \) if and only if every \( q \) that is not a supercontract of \( p \) is rejected by \( m \).
**Definition 4.1 (Rejection Sound).** \( \text{smon}(p,m) \overset{\text{def}}{=} \forall q \cdot \text{rej}(q,m) \implies p \not\sqsubseteq_{\text{SRV}} q. \)**

**Definition 4.2 (Rejection Complete).** \( \text{cmon}(p,m) \overset{\text{def}}{=} \forall q \cdot p \not\sqsubseteq_{\text{SRV}} q \implies \text{rej}(q,m). \)**

We can also extend these monitorability definitions to a specification language of contracts (i.e., a set of contracts).

**Definition 4.3 (Language Rejection Monitorability).** A set of contracts \( C \) is:

- sound rejection-monitorable iff \( \forall p \in C \cdot \exists m \in \text{MON} \cdot \text{smon}(p,m) \)
- complete rejection-monitorable iff \( \forall p \in C \cdot \exists m \in \text{MON} \cdot \text{cmon}(p,m) \)
- rejection-monitorable iff \( \forall p \in C \cdot \exists m \in \text{MON} \cdot \text{smon}(p,m) \) and \( \text{cmon}(p,m) \)

We can readily argue in a formal manner that the contract language SRV of Figure 1 cannot be rejection-monitorable. Consider as an example \( \overline{\alpha}.0 + b.0 \in \text{SRV} \). If this language is rejection-monitorable, then there must exist a monitor \( m \) such that \( \text{smon}(\overline{\alpha}.0 + b.0,m) \) and \( \text{cmon}(\overline{\alpha}.0 + b.0,m) \). We argue towards a contradiction. From Section 2 we know that \( \overline{\alpha}.0 + b.0 \not\sqsubseteq_{\text{SRV}} \overline{\alpha}.0 \), and thus, by \( \text{cmon}(\overline{\alpha}.0 + b.0,m) \), it must be the case that \( \text{rej}(\overline{\alpha}.0,m) \). Now this rejection predicate holds if either \( m \) reaches a rejection state immediately or else reaches rejection after observing action \( a \). In either case, this monitor would also reject the contract \( \overline{\alpha}.0 + b.0 \) as well, which would make the monitor necessarily unsound, i.e., \(~\text{smon}(\overline{\alpha}.0 + b.0,m)\), since, by the reflexivity property of the preorder, we have \( \overline{\alpha}.0 + b.0 \sqsubseteq_{\text{SRV}} \overline{\alpha}.0 + b.0 \).

We deem sound rejection to be the minimum correctness requirement to be expected from the contract monitors we consider. Note, however, that the contract language SRV of Figure 1 is trivially sound rejection-monitorable via the monitor \( \text{end} \); this monitor never reaches a rejection state and thus trivially satisfies \( \text{rej}(p,\text{end}) \) for any \( p \in \text{SRV} \). However, we argue that this monitor, \( \text{end} \), is not very useful.

We attempt to go one step further and define an automated monitor synthesis function that returns a monitor for every server in the contract language SRV. We argue, at least informally, that these synthesised monitors are, in some sense, useful because they perform a degree of violation detections. Importantly, however, we show that these synthesised monitors are rejection sound, according to Definition 4.1.

**Definition 4.4 (Monitor Synthesis).** The function \([\cdot] : \text{SRV} \rightarrow \text{MON} \) synthesises a monitor from a server contract description, and is defined inductively on the structure of this contract as follows:

\[
\begin{align*}
[0] & \overset{\text{def}}{=} \text{end} & [\alpha.p] & \overset{\text{def}}{=} \overline{\alpha}.N + \overline{\alpha}.[p] \\
[p + q] & \overset{\text{def}}{=} [p] \times [q] & [p \oplus q] & \overset{\text{def}}{=} [p] \times [q]
\end{align*}
\]

A few comments on Definition 4.4 are in order. First, note that a number of the monitors considered earlier in Section 3 are in fact instances of this translation. For instance, we have

\[
[\overline{\alpha}.b.0] = \overline{\alpha}.N + \overline{\alpha}.(\overline{b}.N + b.\text{end}) \quad \text{and} \quad [\overline{\alpha}.b.0 + c.0] = (\overline{\alpha}.N + \overline{\alpha}.(\overline{b}.N + b.\text{end})) \times (\overline{c}.N + c.\text{end})
\]

from Example 3.3. Secondly, note that the monitor synthesis does not attempt to perform any detection violation for the contract \( 0 \). Since \( 0 \) is in some sense a bottom element in the preorder, no supercontract of \( 0 \) is allowed to perform any visible action. Thus, in cases where all the actions permissible in SRV are known up front as a finite set \( \{\alpha_1, \ldots, \alpha_n\} \), we can improve the precision of our synthesis through the alternative definition \( [0] \overset{\text{def}}{=} \overline{\alpha_1}.N + \ldots + \overline{\alpha_n}.N \) for the case where \( p = 0 \). Third, note that the synthesis for both internal and external choice constructs coincide which, in a sense, is due to the inherent discriminating limits of RV. Consider, by way of example, the monitor synthesises below:

\[
[\overline{\alpha}.0 + b.0] = (\overline{\alpha}.N + \overline{\alpha}.\text{end}) \times (\overline{b}.N + b.\text{end})) = [\overline{\alpha}.0 \oplus b.0]
\]
The server \(c.0\) is rejected by the monitor \(((a.N + a.end)) \times ((b.N + b.end))\) and accordingly it is neither a supercontract of \(7.0 + b.0\) nor of \(7.0 \oplus b.0\). However, the server \(7.0\) is not rejected by the monitor \(((a.N + a.0)) \times ((b.N + b.0))\); whereas it is correct to do so in the case of monitoring for the internal choice contract \(7.0 \oplus b.0\) because \(7.0 + b.0 \sqsubseteq_{SRV} 7.0\), it leads to lack of precision in the case of the external choice \(7.0 + b.0\) since \(7.0 + b.0 \not\sqsubseteq_{SRV} 7.0\). In spite of these limitations, we are able to show that our proposed monitor synthesis is sound.

**Theorem 4.5 (Synthesis Soundness).** For every server specification \(p \in SRV\), every server implementation \(q \in SRV\), and the monitor synthesis function \([-\] of Definition 4.4:

\[
\text{Whenever } \text{rej}(q, \llbracket p \rrbracket) \text{ then it is necessarily the case that } p \not\sqsubseteq_{SRV} q
\]

**Proof.** By structural induction on the server specification \(p\). \(\Box\)

5 Conclusion

We have presented preliminary investigations relating to the monitorability of contracts, high-level descriptions for web services. We developed a monitoring framework that complements the operational semantics of server contracts. We then focused on the rejection expressivity of the monitors within this framework and related it to cases where it is unsafe to replace one server (contract) with another. Within our simple framework, we were already able to identify limits with respect to monitor detection powers, and were able to diagnose problems with a proposed automated monitor synthesis procedure. We were also able to formally prove that, in spite of its limit, the monitor synthesis considered is, in some sense, correct (Theorem 4.5).

**Related and Future Work** The language of contracts for web services has been discussed in several other works prior to ours, such as [2, 5, 15, 7]; although conceptually simple, it has been shown to be expressive enough to capture the dynamicity of interactions specified by more elaborate contract descriptions. The server preorder considered in this paper captures the essence of the must preorder, studied in [3] and the compliance preorder, studied in [15, 7]; in our simplistic case of finite servers and clients, the two preorders coincide (modulo minor technical details regarding client satisfaction and computation success). Our notion of monitorability is inspired by that presented in [11], which relates process satisfaction of a branching-time logic, \(p \models \phi\), with detections of monitors synthesised from formulas in this logic, \(\llbracket \phi \rrbracket \triangleleft p\). The instrumentation relation considered in this paper is in fact an adaptation to the one used in [11].

For future work, we aim to achieve a more comprehensive study of monitorability than the preliminary one presented in Section 4. In particular, we plan to consider monitor acceptances as a verdict in addition to rejections, establish stronger results with respect to rejections and consider extended contract descriptions similar to [3, 7] that include recursion and the potential for infinite computation. This will lead to different notions of server refinements such as those resulting from compliance and fair testing preorders [5, 15]. It will be interesting to study whether any of the aforementioned server preorder variants are more monitorable than the others.

**Acknowledgements:** This research was partly supported by the project “TheoFoMon: Theoretical Foundations for Monitorability” of the Icelandic Research Fund.
References


