基于符号执行与混合约束求解的测试用例生成研究

周海将 吴军华
(南京工业大学计算机科学与技术系 江苏 南京 210009)

摘 要 由于现有基于符号执行的测试用例生成方法无法对关于字符串的测试用例生成提供有效支持,因此提出并实现了基于符号执行与混合约束求解测试用例自动化生成方法。该方法利用模型检测软件对被测软件源代码进行符号执行,生成关于字符串与数值的混合约束集,利用字符串-数值约束求解器对约束集进行求解,最终根据求解结果生成软件测试用例与不可达路径。实验结果表明,该方法较好地支持了关于字符串测试用例生成,且具有良好的效率与准确性。

关键词 测试用例生成 符号执行技术 混合约束求解 字符串

中图分类号 TP301 **文献标识码** A DOI:10.3969/j.issn.1000-386x.2016.06.006

ON TEST CASE GENERATION BASED ON SYMBOLIC EXECUTION AND HYBRID CONSTRAINT SOLVING

Zhou Haijiang Wu Junhua
(*Department of Computer Science and Technology, Nanjing Tech University, Nanjing 210009, Jiangsu, China*)

Abstract Since current test case generation method based on symbolic execution is unable to provide effective support to the test case generation with regard to character string, therefore we present an automatic test case generation method, which is based on symbolic execution and hybrid constraint solving, and implement it as well. The method uses the model checking software to make symbolic execution on the source code of the software to be tested and generates a mixed constraint set correlated to string and numeral. Then it uses the string-numerical constraints solver to calculate the constraint set. Finally, according to the calculation result it generates the software test case and the infeasible path. Experimental result shows that this method well support the generation of test case in regard to string and has good efficiency and accuracy.

Keywords Test cases generation Symbolic execution technology Hybrid constraint solving String

0 引 言

随着软件业的不断发展,主流软件中字符串操作的占比越来越大[1]。例如 Web 软件中后台逻辑处理部分通常需要与数据库进行交互,其中 SQL 语句的动态生成主要通过字符串操作完成。此外,前端界面用于显示的字符串变量与常量基本通过字符串操作动态地生成。而软件测试是保证软件质量的重要手段,由于早期的传统软件测试方法主要针对数值与逻辑计算,对于字符串变量和字符串操作不能有效的支持,因此关于字符串的测试用例生成技术逐渐成为国内外的研究热点之一[2]。

1 研究现状

近年来软件测试领域基于符号执行的测试研究成为热点。它是一种静态程序分析技术[3],主要原理是使用符号变量代替实际值模拟程序的执行,获取相应路径的约束,最终通过约束求解获得测试数据。如今该技术的瓶颈在于约束求解器的效率,尤其缺乏优秀的字符串求解器。关于约束求解技术,根据表示字符串的形式,字符串约束求解器主要分为基于位向量与基于有限状态自动机两种类型。

HAMPI[4]是基于位向量的字符串约束求解器,主要原理是将输入的字符串约束转换为是否为正则语言接受的形式,利用位向量表示,最后利用 STP 位向量求解器求解约束条件。但仅支持固定长度的字符串约束且不支持数值约束。Kudzu[5]基于位向量,改进了 HAMPI,提供了较多字符串操作的支持,并且有限支持数值约束。

文献[2]详细论述了基于有限状态自动机的字符串约束求解器技术。其中 Rex[7]基于有限状态机与 SMT 约束求解器,使用逻辑谓语表示自动机的转换。Rex 支持字符串约束与数值约束,但效率很低,远远慢于 HAMPI。文献[8]提出一种基于怠惰算法的约束求解技术,利用图表示字符串之间的约束,通过回溯搜索图寻求结果。该技术若有解时,效率很高;若无解则求解时间往往超出可接受范围。此外,该技术只能对字符串操作与正则语言提供有限的支持。

综上所述,要实现基于符号执行软件测试自动化,就应改进约束求解技术的效率与增强适用性。

收稿日期:2015-01-30。**周海将**,硕士,主研领域:软件测试。**吴军华**,副教授。

2 符号执行与约束求解

符号执行[3]是一种可靠的程序分析技术,基本原理是:在源代码中,将程序中变量值替换为表示为未知但固定的符号标记,模拟程序执行的过程。符号标记在静态数学意义上用来表示一些未知的但固定的值,一个程序在特定的执行过程中会有与其相关的多个不同的符号标记。主要思想是从控制流图的入口处用符号标记代替具体的输入值来模拟程序符号化的执行过程。一般的软件测试用具体值作为测试用例输入来执行程序,所进行的计算是具体逻辑运算,符号执行使用符号标记来执行程序,所进行的是代数运算,符号执行即一般测试的扩充。符号执行是程序验证和程序测试的折中,一方面它具有普通测试所具有的测试方法,通过运行被测程序验证程序的可靠性;另一方面,符号执行沿着一条路径的一次执行积累的约束条件代表了一大类普通测试的集合,验证程序接受此类输入后是否正确,同时可以发现程序中不可行的路径。

约束问题由三个基本元素组成:变量、变量域与约束。约束求解即在变量域中找到特定变量值使之与各变量之间均满足约束。混合约束问题是传统约束问题的扩展,即变量域为多个,本文分为数值变量域与字符串变量域。约束求解技术主要包括数值法、符号计算法、区间分析法、启发式算法等。

3 基于符号执行与混合约束求解的测试用例生成技术

3.1 符号执行框架

本文提出了一个基于符号执行技术与混合约束求解的测试用例自动化生成方法框架,总体来说包括两个部分:符号执行生成约束部分;对约束集求解得到测试输入与不可达路径部分。主要步骤为:

(1) 将源程序进行扩展,利用符号执行工具对已扩展的源代码进行符号执行,遍历源程序的每个路径,生成每个与程序终点相对应的路径条件。路径条件定义为与该程序终点对应的约束。这样的过程最后得到的就是路径条件集合,即约束条件集合。

(2) 利用约束求解方法对约束条件集合进行约束求解,得到关于每个程序终点的约束的具体值,即程序测试输入。若约束求解无解,即程序终点不可达,说明该路径是错误的,返回该错误路径。符号执行框架的结构如图 1 所示。

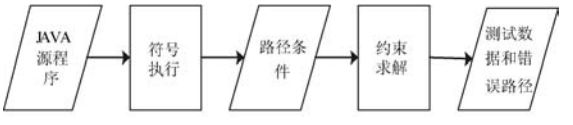

图 1 基于符号执行与混合约束求解测试用例生成框架

本文对基于符号执行与混合约束求解测试用例生成框架给出以下定义:

定义 1 定义执行该路径得到的路径条件为 PC(Path Condition),PC 积累了该路径每一条语句执行必须满足的约束。不可达路径的执行路径定义为 P_U。

定义 2 约束集 C:定义由对源程序进行符号执行得到的路径集合为 $C\{c_1,\cdots,c_i,\cdots,c_n\}$,称 C 为约束条件集合,其中 c_i 表示某个具体的路径条件表达式,n 为约束条件的总数。

定义 3 若是关于数值的路径条件,定义为数值约束 PC_N;若是关于字符串的约束,定义为字符串约束 PC_S;若 PC 中既包含数值约束又包含字符串约束,称该 PC 为混合约束条件 PC_H,对于混合约束条件集合为 C_H。

定义 4 由 PC_N 求解得到的值为关于数值约束条件的测试输入,定义为 T_N;由 PC_S 求解得到的值为关于字符串约束条件的测试输入,定义为 T_S;相应的 T_N 与 T_S 的集合即关于该执行路径的测试输入 T。

定义 5 由对约束条件集合进行求解的过程定义为 S,其中 S 的输入是约束条件集合,输出为测试输入集合与不可达路径条件。表达式为:

$$S(C) = T \text{ AND } P_U$$

其中 $C = \{c_1,\cdots,c_i,\cdots,c_N\}$;$T = \{T_1,\cdots,T_i,\cdots,T_N\}$,$T_i = T_{Ni}$ AND T_{Si}。

3.2 路径条件生成

本文将通过图 2 的例子详细论述符号执行的过程。图 2 表示的是根据字符串取值的不同采取不同处理,目的是对字符串进行辨识以确定 S 的含义,并决定采取哪些方法处理 S。

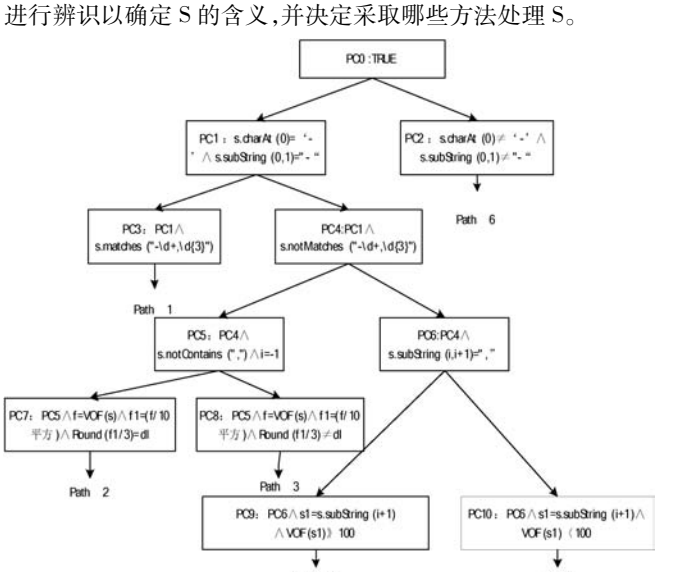

图 2 路径条件的分支树

如图 2 所示,PC0 表示的是符号执行的起点,Path1 – Path6 表示的是符号路径的终点。符号 $L(q)$ 代表字符串 q 的长度。符号执行过程中,首先检测输入字符串是否以负号为开头,即检测字符串 s 代表的是否为负数;接着检测字符串是否符合正则表示式的格式;然后检测字符串 s 中是否含有逗号,若无逗号,字符串 s 将被转换为长精度类型,否则将 s 中逗号后面的数字赋值整型变量 x;最后检测 x 是否小于 100。这个代码在一般的 Web 程序中比较常见,首先对输入字符串进行格式检测,接着将字符串转换为数字,最终在分支中对字符串进行处理。在每个路径终点都生成了约束条件的交集,即通过符号执行得到的约束条件集合 C,其中 C 中元素为:

$c_1 = \{$s. charAt(0) == '-'$\}$

$c_2 = \{$ s. charAt(0) == '-', s. matches("-\d+,\d{3}")$\}$

$c_3 = \{$s. charAt(0) == '-', s. notmatches("-\d+,\d{3}", s. notCtains(","), i == -1, f == VOF(s), f1 == (f/10²), round(f1/3) = dl$\}$

$c_4 = \{$s. charAt(0) == '-', s. notmatches("-\d+,\d{3}", s. notCtains(","), i == -1, f == VOF(s), f1 == (f/10²), round(f1/3) ≠

dl}

c_5 = {s. charAt(0) == '-',s. notmatches("-\d+,\d{3}",s. subString(i,i+1) == ","sl = s. substring(i+1),VOF(sl) >=100 }

c_6 = {s. charAt(0) == '-',s. notmatches("-\d+,\d{3}",s. subString(i,i+1) == ","sl = s. substring(i+1),VOF(sl) <100 }

在下一步中通过对 C 进行约束求解,则可以生成测试数据与不可达路径。由于 C 中的元素 c_i 既包含 PC_N 又包含 PC_S,因此,约束求解器就需要具有对 PC_N 和 PC_S 两种约束条件的求解能力。当前的约束求解算法对字符串约束不能提供有效支持,对此本文提出并实现了数值—字符串混合约束求解器[9]。

3.3　混合约束求解算法

混合约束集合包含数值约束与字符串约束。本文对此的解决方法为:通过将两种约束集进行分离,运用不同的约束求解器进行处理。数值约束条件选用已有的 Coral[10] 方法和 Yices[11] 方法处理,其中线性数值约束使用 Yices 方法处理,简单非线性数值约束使用 Coral 方法处理。对于字符串约束条件使用本文设计的基于有限状态机(FSM)约束求解器。具体算法为:

步骤 1　初始化约束求解器。加载由符号执行步骤得到的约束条件集合 C_H。对 C_H 中的所有元素 c_i 遍历执行以下步骤。

步骤 2　利用 Coral 与 Yices 数值约束求解器对 c_i 中的 P_N 部分进行求解,若无解则输出该执行路径,并退出;若有解,保存 T_s 并执行下一步。

步骤 3　将 C_H 中的 C_s 部分加载到求解器中,并对其进行字符串求解。若有解则输出 T_s 与 T_n。即该执行路径的测试输入;若无解,执行以下步骤。

步骤 4　首先判断求解时间是否超出限制,若超出则放弃,返回求解时间并退出;反之,根据 RULE 库替换可以替换的字符串约束条件为相应的数值约束条件,转到步骤 1 循环。

在算法中提到的 RULE 库是为了提高混合约束条件求解效率而增加的。主要实现的原理:字符串约束部分可以转换为数值约束,数值约束求解效率一般优于字符串约束期间效率,实现部分数值约束与字符串约束互相的转换将提高求解效率。

3.4　字符串约束求解改进

本文使用有限状态机(FSM)表示一个符号字符串变量,即所有变量可能的取值组成了可以被这个 FSM 识别的语言。实现是基于文献[12]提出的自动机。FSM 的转换(边)拥有一个范围取值。当一个自动机求精时,不仅需要改变结点和边,边上的额外取值范围也会变化。例如图 3 表示的是接受正则表达式"-\d+,\d{3}"的自动机。

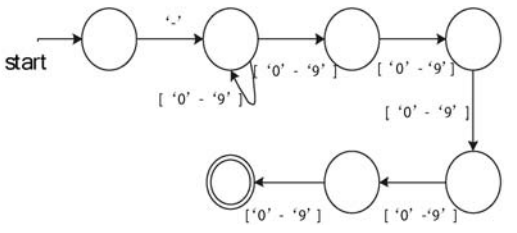

图 3　接受 -\d+,\d{3} 的自动机

本文通过使用有限状态自动机来实现字符串的操作。例如,字符串 $s1$ 和字符串 $s2$ 的连接操作转换为 $s2$ 的自动机附加到 $s1$ 自动机的操作。该自动机支持了很多的字符串操作,但是因为其设计是用于静态分析,本文扩展该自动机不支持的字符串操作。例如,substring(2,4)方法的操作返回的是从索引位置 2 到索引位置 4 的子字符串,如图 4 所示。基本方法是:首先从开始状态转移两个步骤;然后将访问到的节点作为开始节点;接着将从开始节点两步转换中所有的状态作为接受状态;最后,我们交互该自动机并接受所有长度为 2 的自动机用来得到最终的自动机版本。除此之外还有其他需要扩展,比如替换字符与替换子字符串等。

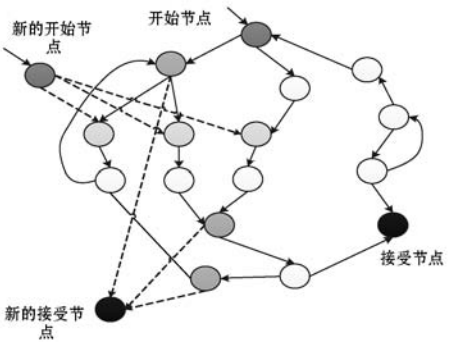

图 4　在自动机上操作 substring(2,4)操作

字符串约束描述的是字符串之间的关系。FSM 并不直接对字符串约束求解,因此需要根据字符串约束加强对字符串的求精。这个过程包括:(1)自动机求精;(2)不动点计算;(3)优化收敛速度。例如,对于 s. charAt(0) = '-'约束,将字符串 s 与接收首字符为负号的自动机交互。后来,因为字符串 s 的部分通过 parseInt 方法转换为整型变量,将字符串 s 的该部分与建模所有有效整数交互;同样需要对字符串 s 的根据更新部分对字符串 s 进行求精。例如,将字符串 s 的自动机与以某种形式结尾部分的自动机交互。这部分工作将一直进行,直到没有什么求精是可能的,并且得到不动点,即得到一个可能的字符串值。

本文通过 RULE 库来实现字符串约束与数值约束之间的互相转换,因此字符串约束和数值约束对于共享的符号变量取值必须一致,即共享的符号变量在数值约束与字符串约束中的值相同。在符号变量同步的过程中,数值约束求解器 N 会给出一些候选值,提交给字符约束求解器 S 判断该符号变量值是否冲突。若无冲突,则符号变量取值被 S 和 N 都接受;否则 S 会返回 N 引起冲突的约束变量值,请求 N 提交其他候选符号变量取值。接着 N 取消冲突的符号变量值,使用启发式算法重新搜索另一个有效赋值,继续以上步骤。在交互的过程中只有共享变量的具体值可以从 N 提交到 S,N 也可以理解一些字符串冲突,并传递给 S。例如,数值约束 s. length() >5 使相应的自动机应该与接受长度大于 5 的自动机交互。

在迭代过程中,我们将字符串约束求解器[14,15]和数值约束求解器得到的约束交换,这样就可以提高收敛速度。例如,对于字符串约束 $s1$ = $s2$. trim(),数值约束求解器会给出一个数值约束给出 $L(s1) <= L(s2)$,其中 $L()$ 方法返回的是字符串对象的长度。

在方法中,类似上例的交互约束由 RULE 库建模,RULE 库包含一般 Java 程序常见的交互规则,表 1 展示了该库的部分规则。

表 1　RULE 库中部分 RULE 规则

字符串约束条件	转换之后的数值约束条件
s3 = s1. concat(s2)	L(s3) = L(s1) + L(s2)
s2 = s1. trim()	L(s2) <= L(s1)

续表 1

字符串约束条件	转换之后的数值约束条件
s2 = s1. substring(i,j)	I > =0 ∧j < L(s2) ∧ i < = j ∧ L(s2) = j - i
i = s. lastIndexOf(c)	s. substring(i,i + 1). equal(c)
s. charAt(0) = ' - ' ∧n = VOF(s)	n > 0
s. charAt(0) = ' - ' ∧n = VOF(s)	n < 0
VOF(s) = n	s. equal("n")
VOF(s) < 0	s. charAt(0) = ' - '
VOF(s) > c	L(s) > = log(c) ∧ s. charAt(0)! = ' - '
L(s. trim()) = c	L(s) > = c

为了解释数值与字符串约束迭代求解的过程,对于图 2 中的 path 4,下面给出了 path 4 的路径条件,包含了已经增加的额外的约束。为了对该约束求解,数值约束求解器将得到一组有效的数值变量赋值,$L(s) = 2$, $L(s1) = 1$, $i = 0$, $x = 100$。然后将这组赋值提交给字符串约束求解器。但是因为 x = parseInt(s1) 条件不满足,所以字符串求解器没有找到有效赋值。这就要求 N 进行重新计算直到 s1 的长度大于等于 3,或者求解器利用 RULE 库从"x = parseInt(s1) ^ 》100"得到一个额外的 L(s1) > =3约束。此外从"s1 = s. substring(i + 1)"得到 L(s) > = L(s1)。有了额外的约束,求解器可以很快得到新的一组有效变量赋值,例如 s = " - ,100", s1 = "100", i = 1, x = 100。

Numeric	String
(1 : L(s) > 0)	s:charAt[0] = `- '
(2 : i = 1 _ 0 ¶ i < L(s))	i = s. lastIndexOf(',')
(3 : i + 1 + L(s1) = L(s))	s1 = s. substring(i + 1)
(4 : L(s1) > 0)	x = parseInt(s1)
i = ≠ -1 ^ x》100	¬ (s. matches(- \d + ,\d{3}))

当字符串约束求解器 S 不能得到一个有效赋值时,额外的约束被传递给数值约束求解器并始新的迭代计算。最终,混合约束迭代求解器可以得到一个有效的解决。

在符号执行得到的混合约束可能包含非线性数值约束条件,对非线性约束求解,正常的处理不仅效率不高,而且容易导致约束求解失败。对此本文通过将非线性约束条件转换为多个线性约束组合来避免直接求解效果不佳。例如 round() 方法,该方法的作用是对于给定的浮点数变量返回最接近的整型值。对于 round() 方法,本文将其转换为"((e -0.5 < x1) ^ (e + 0.5 > = x1)) ^((e -0.5 < =x2) ^ (e + 0.5 > x2)) ^(result = (if (e > 0) x1 x2))"。其中 *e* 为实数,*x*1、*x*2 和 *result* 是引进的整型变量,*result* 代表的 *round*(*e*) 的返回值。同样地,将其他的一些约束进行类似的转换,用以处理非线性数值约束。

4 实验部分

本文使用的是两个富含字符串操作的 Java 源程序进行对比试验,使用 JPF 的符号执行版本[13]对实验样本进行符号执行两个程序的详细情况见表 2。由于程序代码量过大,因此仅关注逻辑处理部分,因为这部分代表了程序中最复杂的字符串与数值计算与处理。测试环境为 HP PC 机 - Intel Core2 Duo CPU E7500,2.93 GHz,2 GB;Windows XP sp3。

表 2 实验程序数据

程序	描述	类个数	代码行数	核心代码行数	符号输入
A	B2B 订单系统	2801	1226K	3379	4(strings)
B	金融应用系统	3	1672	1157	3(strings)

测试中记录生成的是输入符号变量的个数,在程序有效结束状态下生成的测试用例个数,在符号执行中因异常抛出而未执行的路径个数。完成实验的时间、完成实验中迭代的次数见表 3 所示。最后在符号输入个数为四个时,得到了的测试用例覆盖率达到 80%。

表 3 程序 A 的实验结果

符号输入	真实路径	异常路径	时间	迭代次数
1	6	6	8 s	1
2	27	5	16 s	1
2	35	9	55 s	2
3	595	16	3 min 13 s	7
3	493	25	4 min 26 s	7
4	1971	95	10 min 59 s	23

由表 3 可以看出在符号输入个数为 4 时,得到了的测试用例覆盖率达到 80%。综上所述,该技术在合理的时间内能有效自动化地生成测试用例与不可达路径。

为了说明非线性约束求解的有效性,本文使用金融应用程序 B 进行单元测试。程序 B 包含很多对 Java 语言长精度类型的舍入操作。实验对非线性模式开启与否做了多次实验(如表 4 所示)。当开启非线性模式时,所有生成的路径条件由 Yices 求解,反之则由 Coral 求解。可以从表中明显看到,非线性模式下的具有明显优势。

表 4 程序 B 的实验结果

符号输入	开启非线性模式			关闭非线性模式		
	准确路径	异常路径	时间	准确路径	异常路径	时间
1	5	0	1 s	3	0	9 s
2	37	11	19 s	26	9	57 min 1 s
2	27	21	23 s	21	7	45 min 1 s
3	169	49	1 min 39 s	33	9	6 h 37 min

从表 3、表 4 可以看出,本文提出的基于符号执行与混合约束求解技术的测试自动化技术明显优于传统基于符号执行的测试自动化技术。不仅对字符串约束与数值约束提供支持,求解效率也有明显提升。

5 结 语

本文对符号执行技术、数值—字符串混合约束求解进行了研究,设计并实现了基于符号执行和混合约束求解的自动化测试技术。它能有效提高约束求解效率,并支持字符串约束与数值约束。实验表明,该方法能在可接受的时间范围内生成测试

(下转第 31 页)

下,用户使用本应用时,缓存检测到离线事件,直接访问设备上的缓存数据,进行界面展示。

项目场景二是联线数据的更新。在移动设备断网的情况下,对应用服务器端的数据进行更新,当网络恢复正常后,本应用自动下载最新数据,并更新设备上的缓存数据。

由于系统设计的这部分功能是通过后台自动进行的,所以没有界面可以展示,实测下来两个场景均顺利实现。

表 1 给出的是查询某品牌商品的销售量,将采用 H-OAAS 的应用同采用实时查询方式的应用进行对比。前者由于不依赖网络,具有更快的速度,后者在网络状况良好的情况下,接近前者的速度。

表 1　两种不同开发模式下的查询对比

	1 个月销售量	2 个月销售量	3 个月销售量
H-OAAS 查询时间	1.6 秒	2.2 秒	3.5 秒
实时查询时间	2.5 秒	3.8 秒	5 秒

在移动网络应用中,许多查询结果所得到的数据是重复的,重复的查询会给系统带来一定的压力。H-OAAS 的使用减轻了服务器端的压力,提高了前端性能。从表 1 中可以看出H-OAAS 的应用提高了用户请求的响应效率,同时由于采用数据离线存储的机制也大大减轻了服务器实时响应用户操作的压力。

5　结　语

基于当前移动应用在使用过程中的局限性,本文提出移动商务智能信息展现系统的应用框架,较好地解决了目前遇到的问题,使信息的分析结果能够随时随地的为用户提供支撑和服务。该系统目前已经在国内的一家大型企业中正式投入使用,为该企业的管理层提供整个企业的运营信息分析。

在后续的工作中,还将继续针对系统的核心功能进行深入的研究和优化:

1) 将继续研究 HTML5 插件的设计与实现,进一步拓展展现模块的跨平台能力,如对微软的手机操作系统(Windows Phone)的支持;

2) 为用户提供自定义界面的功能,由目前的由系统管理员对用户界面进行配置转变为由最终用户自行对展现界面进行配置;

3) 进一步研究离线数据的压缩与解压缩方式,降低数据传输时所需要的流量,提高移动设备对压缩数据的解压效率,提升系统的响应效率。

未来,还将在系统中细化权限机制,针对不同权限的用户提供行列级的权限管理功能。增加用户使用行为的记录功能,用于后续研究用户的使用习惯,使系统的功能更加完整,并为今后不断的提升系统的用户体验,打下良好的基础。

参　考　文　献

[1] 罗森,夏普. HTML5 用户指南[M]. 刘红伟,译. 北京:机械工业出版社,2011.

[2] Peter Lubbers, Brian Albers, Frank Salim, et al. HTML5 高级程序设计[M]. 2 版. 李杰,柳静,刘淼,译. 北京:人民邮电出版社,2010.

[3] Reto Meier. Android 4 高级编程[M]. 3 版. 余建伟,赵凯,译. 北京:清华大学出版社,2013.

[4] 杨宏焱. 企业级 iOS 应用开发实战[M]. 北京:机械工业出版社,2013.

[5] Gene Backlin. iPhone&iPad 高级编程[M]. 岳虹,凌冲,译. 北京:清华大学出版社,2012.

[6] 关东升. iOS 网络编程与云端应用最佳实践[M]. 北京:清华大学出版社,2013.

[7] 施伟,王硕苹,郭鸣,等. 跨平台移动应用中间适配层设计与实现[J]. 计算机工程与应用,2014,50(16):39-44.

[8] 罗大晖,陈娟. 基于 HTML5 的 Web 离线应用研究与实现[J]. 计算机应用与软件,2012,29(12):262-264,305.

[9] Dewsbury R. Google Web Toolkit 应用程序开发[M]. 秦绪文,李松峰,译. 北京:机械工业出版社,2008.

[10] 陆晨,冯向阳,苏厚勤. HTML5 WebSocket 握手协议的研究与实现[J]. 计算机应用与软件,2015,32(1):128-131,178.

[11] Nathan Yau. 鲜活的数据:数据可视化指南[M]. 向怡宁,译. 北京:人民邮电出版社,2012.

(上接第 26 页)

用例与错误路径。文中的混合约束求解框架还有一些不足与需要改进的算法。混合约束求解算法的效率还存在提高的可能,优化可以采用各个击破的"on-the-fly"技术。此外该方法只能对 Java 源程序提供支持,以后可以扩展到其他主流编程语言例如 C++、C 等。

参　考　文　献

[1] 梅宏,王啸吟,张路. 字符串分析研究进展[J]. 软件学报,2012,24(13):37-49.

[2] Hooimeijer P, Veanes M. An evaluation of automata algorithms for string analysis[C]//Verification, Model Checking, and Abstract Interpretation. Springer Berlin Heidelberg, 2011: 248-262.

[3] King J C. Symbolic execution and program testing[J]. Communications of the ACM, 1976, 19(7): 385-394.

[4] Ganesh V, Kie ̇zun A, Artzi S, et al. HAMPI: A string solver for testing, analysis and vulnerability detection[C]//Computer Aided Verification. Springer Berlin Heidelberg, 2011: 1-19.

[5] Saxena P, Akhawe D, Hanna S, et al. A symbolic execution framework for javascript[C]//Security and Privacy (SP), 2010 IEEE Symposium on. IEEE, 2010: 513-528.

[6] Veanes M, De Halleux P, Tillmann N. Rex: Symbolic regular expression explorer [C]//Software Testing, Verification and Validation (ICST), 2010 Third International Conference on. IEEE, 2010: 498-507.

[7] Hooimeijer P, Weimer W. Solving string constraints lazily[C]//Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010: 377-386.

[8] Shannon D, Ghosh I, Rajan S, et al. Efficient symbolic execution of strings for validating web applications[C]//Proceedings of the 2nd International Workshop on Defects in Large Software Systems: Held in conjunction with the ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA 2009). ACM, 2009: 22-26.

[9] Souza M, Borges M, d'Amorim M, et al. CORAL: solving complex constraints for symbolic pathfinder [M]//NASA Formal Methods. Springer Berlin Heidelberg, 2011: 359-374.

[10] Dutertre B, De Moura L. The yices smt solver[OL]. 2006. Tool paper at http://yices. csl. sri. com/tool-paper. pdf.

[11] Christensen A S, Møller A, Schwartzbach M I. Precise analysis of string expressions[M]. Springer Berlin Heidelberg, 2003.

[12] Păsăreanu C S, Rungta N. Symbolic PathFinder: symbolic execution of Java bytecode[C]//Proceedings of the IEEE/ACM international conference on Automated software engineering. ACM, 2010: 179-180.