

Computing Specification-Sensitive Abstractions for Program Verification

Tianhai Liu¹, Shmuel Tyszberowicz³, Mihai Herda¹, Bernhard Beckert¹, Daniel Grahl¹, and Mana Taghdiri²

¹ Karlsruhe Institute of Technology, Germany

² Horus software GmbH, Germany

³ The Academic College Tel Aviv Yaffo, Israel

Abstract. To enable scalability and address the needs of real-world software, deductive verification relies on modularization of the target program and decomposition of its requirement specification. In this paper, we present an approach that, given a Java program and a partial requirement specification written using the Java Modeling Language, constructs a semantic slice. In the slice, the parts of the program irrelevant w.r.t. the partial requirements are replaced by an abstraction. The core idea of our approach is to use bounded program verification techniques to guide the construction of these slices. Our approach not only lessens the burden of writing auxiliary specifications (such as loop invariants) but also reduces the number of proof steps needed for verification.

1 Introduction

Motivation. The power of deductive program verification has increased considerably over the last decades. To enable scalability and address the needs of real-world software, deductive verification relies on modularization of the target program. This requires annotating sub-procedures with formal auxiliary specifications (method contracts, loop invariants, etc.). To discover useful specifications that are fulfilled by the annotated sub-procedure and also meet the requirements of the calling procedures is, unfortunately, a difficult and error-prone effort (cf. [6, Chap. 5]). To ease the burden, verification engineers routinely break a complex requirement specification into conjunctions of *partial* specifications, i.e., they decompose not only the implementation but also the specification. Then, usually, only parts of the implementation are relevant for proving a partial property, and only partial and less complex auxiliary specifications are needed. To make use of that advantage, the verification engineer needs to identify the slice of the implementation relevant to the partial property. The main contribution of this paper is an automated method for computing such program slices defined by partial specifications.

Our approach. Given a Java program and a partial requirement specification, written using the Java Modeling Language (JML) [22], we construct a *semantic*

slice (an abstract program). In the slice, the program parts that are irrelevant to the partial requirements are replaced by an abstraction (i.e., they are not completely removed), whereas the rest of the program (i.e., the relevant parts) remains unchanged. (In the rest of the paper we use the terms semantic slice and abstract program interchangeably.) As said above, verifying slices requires fewer auxiliary specifications (as the abstractions have less details), and their correctness—by their construction—implies the correctness of the original program w.r.t. the partial specification under consideration. As a result, our method liberates the verification engineers from finding the relevant slice manually.

Figure 1 illustrates the structure of our novel approach. The core idea is to use bounded program verification techniques to guide the construction of slices. Bounded program verification systems (such as JForge [14], Jalloy [27], and InspectJ [23]) do not require auxiliary specifications. They translate, based on user-provided bounds (that, e.g., limit the number of objects or the number of loop iterations), the analyzed program and its *negated* requirement specification into a satisfiability problem—an SMT [3] formula consisting of a set of constraints, and try to find a solution to that problem. If a solution to that satisfiability problem is found, then that is a counterexample to the correctness of the original program, and no further analysis is required. If no solution is found, the partial property holds—but only w.r.t. the bounds, thus a deductive verification—an unbounded program verification, is still needed.

Before continuing with the deductive verification we compute the slice of the program relevant to the partial requirements. The computation is based on the *unsatisfiable core* (unsat core)—a subset of constraints that is unsatisfiable, obtained during the unsatisfiability proof for the bounded problem. Then we minimize the unsat core to ensure that the proof requires all its elements. The Java program statements that are related to the constraints in the unsat core (by the construction of constraints from the Java code) are known to be relevant for the bounded proof of the requirement specification. We generate a semantic slice by over-approximating the behaviors of the other statements.

Finally, if the semantic slice can be verified using deductive program verification, which requires auxiliary specifications, the original program satisfies the specification as well (by the construction of the slice). Otherwise, we use counterexample-guided refinement to refine the abstraction and repeat the deductive verification.

The semantic slice is generated based on a particular bounded proof. Therefore it (i) may be too abstract, and thus deductive verification is not possible, and (ii) may exclude unnecessary, yet helpful, details, hence deductive verification may require more effort. But, as our evaluation shows, in practice the slice is sufficiently precise.

Our approach not only lessens the burden of writing auxiliary specifications but also eases the deductive verification: less proof steps are needed. Besides, by the *small-scope hypothesis* [20], if the program does not satisfy its specification, in many cases that will be detected during the bounded-verification phase of our approach, avoiding unnecessary attempts at deductive verification.

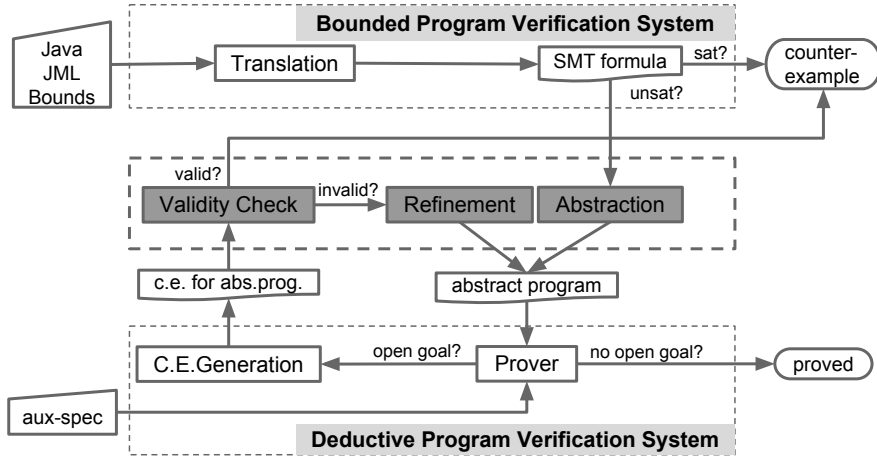


Fig. 1: Structure of our approach.

We have built a prototype tool, *AbstractJ*, that implements the abstraction as well as the refinement and validity check, and we have performed several experiments to evaluate the benefits of our approach.

2 Motivating Examples

We use two examples (Figs. 2 and 3) to demonstrate our approach. To specify the Java modules we employ JML, a behavioral interface specification language. We shortly explain the JML clauses used in the examples; for more details see [22]. The specification is written between `/*@` and `*/`. The `ensures` clause specifies properties that are guaranteed to hold at the end of the method call, and `\result` refers to the value returned by the method. (Both clauses refer to the case that the method terminated normally.) The `diverges` clause is used to specify when a method may either loop forever or not return normally to its caller. Writing `diverges true` means that non-termination is allowed for the method. The `assignable` clause provides the locations that can be assigned to during the execution of the method (frame conditions). The clause `assignable \strictly_nothing` denotes that the relevant methods neither modify heap locations nor allocate objects, whereas `assignable \nothing` allows object allocations; `assignable \everything` enables the method both to modify any heap location and to allocate objects.

The program in Fig. 2(a) computes the number of prime numbers between two given integers `x` and `y` (exclusive). The first line denotes that if the number of prime numbers is larger than 0, then `x < y`. Carefully inspecting the code, a verification engineer will notice that the `ensures` clause becomes false only when `x >= y`. In that case, the outer loop (Fig. 2(a), statement 2) is never executed and the variable `size` remains equal to 0. However, using traditional static slicing techniques, all statements (Fig. 2(a), statements 1-9) will be relevant w.r.t.

<pre> /*@ ensures \result>0==>x<y; @ diverges true; @ assignable \everything;*/ int numberOfPrime(int x, int y){ 1 int size = 0; 2 for(int i=x; i<y; i++){ 3 boolean isPrime = true; 4 for(int j=2; j<i; j++){ 5 if (i%j==0){ 6 isPrime=false; 7 break; 8 } 9 if (isPrime) 10 size++; 11 } 12 if(size > 0){ 13 int[] a = new int[y-x]; 14 } 15 return size; 16 } </pre>	<pre> /*@ ensures \result>0==>x<y; @ diverges true; @ assignable \everything;*/ int numberOfPrime(int x, int y){ 1 int size = 0; 2 for(int i=x; i<y; i++){ 3 size = pure_int(); 4 } 5 pure_allocArrayInt(); 6 return size; 7 } 8 9 10 11 12 13 14 15 16 17 18 19 20 21 22 23 24 25 26 27 28 29 30 31 32 33 34 35 36 37 38 39 40 41 42 43 44 45 46 47 48 49 50 51 52 53 54 55 56 57 58 59 60 61 62 63 64 65 66 67 68 69 70 71 72 73 74 75 76 77 78 79 80 81 82 83 84 85 86 87 88 89 90 91 92 93 94 95 96 97 98 99 100 101 102 103 104 105 106 107 108 109 110 111 112 113 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136 137 138 139 140 141 142 143 144 145 146 147 148 149 150 151 152 153 154 155 156 157 158 159 160 161 162 163 164 165 166 167 168 169 170 171 172 173 174 175 176 177 178 179 180 181 182 183 184 185 186 187 188 189 190 191 192 193 194 195 196 197 198 199 200 201 202 203 204 205 206 207 208 209 210 211 212 213 214 215 216 217 218 219 220 221 222 223 224 225 226 227 228 229 230 231 232 233 234 235 236 237 238 239 240 241 242 243 244 245 246 247 248 249 250 251 252 253 254 255 256 257 258 259 260 261 262 263 264 265 266 267 268 269 270 271 272 273 274 275 276 277 278 279 280 281 282 283 284 285 286 287 288 289 290 291 292 293 294 295 296 297 298 299 300 301 302 303 304 305 306 307 308 309 310 311 312 313 314 315 316 317 318 319 320 321 322 323 324 325 326 327 328 329 330 331 332 333 334 335 336 337 338 339 340 341 342 343 344 345 346 347 348 349 350 351 352 353 354 355 356 357 358 359 360 361 362 363 364 365 366 367 368 369 370 371 372 373 374 375 376 377 378 379 380 381 382 383 384 385 386 387 388 389 390 391 392 393 394 395 396 397 398 399 400 401 402 403 404 405 406 407 408 409 410 411 412 413 414 415 416 417 418 419 420 421 422 423 424 425 426 427 428 429 430 431 432 433 434 435 436 437 438 439 440 441 442 443 444 445 446 447 448 449 450 451 452 453 454 455 456 457 458 459 460 461 462 463 464 465 466 467 468 469 470 471 472 473 474 475 476 477 478 479 480 481 482 483 484 485 486 487 488 489 490 491 492 493 494 495 496 497 498 499 500 501 502 503 504 505 506 507 508 509 510 511 512 513 514 515 516 517 518 519 520 521 522 523 524 525 526 527 528 529 530 531 532 533 534 535 536 537 538 539 540 541 542 543 544 545 546 547 548 549 550 551 552 553 554 555 556 557 558 559 560 561 562 563 564 565 566 567 568 569 570 571 572 573 574 575 576 577 578 579 580 581 582 583 584 585 586 587 588 589 590 591 592 593 594 595 596 597 598 599 600 601 602 603 604 605 606 607 608 609 610 611 612 613 614 615 616 617 618 619 620 621 622 623 624 625 626 627 628 629 630 631 632 633 634 635 636 637 638 639 640 641 642 643 644 645 646 647 648 649 650 651 652 653 654 655 656 657 658 659 660 661 662 663 664 665 666 667 668 669 670 671 672 673 674 675 676 677 678 679 680 681 682 683 684 685 686 687 688 689 690 691 692 693 694 695 696 697 698 699 700 701 702 703 704 705 706 707 708 709 710 711 712 713 714 715 716 717 718 719 720 721 722 723 724 725 726 727 728 729 730 731 732 733 734 735 736 737 738 739 740 741 742 743 744 745 746 747 748 749 750 751 752 753 754 755 756 757 758 759 760 761 762 763 764 765 766 767 768 769 770 771 772 773 774 775 776 777 778 779 780 781 782 783 784 785 786 787 788 789 790 791 792 793 794 795 796 797 798 799 800 801 802 803 804 805 806 807 808 809 810 811 812 813 814 815 816 817 818 819 820 821 822 823 824 825 826 827 828 829 830 831 832 833 834 835 836 837 838 839 840 841 842 843 844 845 846 847 848 849 850 851 852 853 854 855 856 857 858 859 860 861 862 863 864 865 866 867 868 869 870 871 872 873 874 875 876 877 878 879 880 881 882 883 884 885 886 887 888 889 890 891 892 893 894 895 896 897 898 899 900 901 902 903 904 905 906 907 908 909 910 911 912 913 914 915 916 917 918 919 920 921 922 923 924 925 926 927 928 929 930 931 932 933 934 935 936 937 938 939 940 941 942 943 944 945 946 947 948 949 950 951 952 953 954 955 956 957 958 959 960 961 962 963 964 965 966 967 968 969 970 971 972 973 974 975 976 977 978 979 980 981 982 983 984 985 986 987 988 989 990 991 992 993 994 995 996 997 998 999 1000 1001 1002 1003 1004 1005 1006 1007 1008 1009 1010 1011 1012 1013 1014 1015 1016 1017 1018 1019 1020 1021 1022 1023 1024 1025 1026 1027 1028 1029 1030 1031 1032 1033 1034 1035 1036 1037 1038 1039 1040 1041 1042 1043 1044 1045 1046 1047 1048 1049 1050 1051 1052 1053 1054 1055 1056 1057 1058 1059 1060 1061 1062 1063 1064 1065 1066 1067 1068 1069 1070 1071 1072 1073 1074 1075 1076 1077 1078 1079 1080 1081 1082 1083 1084 1085 1086 1087 1088 1089 1090 1091 1092 1093 1094 1095 1096 1097 1098 1099 1100 1101 1102 1103 1104 1105 1106 1107 1108 1109 1110 1111 1112 1113 1114 1115 1116 1117 1118 1119 1120 1121 1122 1123 1124 1125 1126 1127 1128 1129 1130 1131 1132 1133 1134 1135 1136 1137 1138 1139 1140 1141 1142 1143 1144 1145 1146 1147 1148 1149 1150 1151 1152 1153 1154 1155 1156 1157 1158 1159 1160 1161 1162 1163 1164 1165 1166 1167 1168 1169 1170 1171 1172 1173 1174 1175 1176 1177 1178 1179 1180 1181 1182 1183 1184 1185 1186 1187 1188 1189 1190 1191 1192 1193 1194 1195 1196 1197 1198 1199 1200 1201 1202 1203 1204 1205 1206 1207 1208 1209 1210 1211 1212 1213 1214 1215 1216 1217 1218 1219 1220 1221 1222 1223 1224 1225 1226 1227 1228 1229 1230 1231 1232 1233 1234 1235 1236 1237 1238 1239 1240 1241 1242 1243 1244 1245 1246 1247 1248 1249 1250 1251 1252 1253 1254 1255 1256 1257 1258 1259 1260 1261 1262 1263 1264 1265 1266 1267 1268 1269 1270 1271 1272 1273 1274 1275 1276 1277 1278 1279 1280 1281 1282 1283 1284 1285 1286 1287 1288 1289 1290 1291 1292 1293 1294 1295 1296 1297 1298 1299 1300 1301 1302 1303 1304 1305 1306 1307 1308 1309 1310 1311 1312 1313 1314 1315 1316 1317 1318 1319 1320 1321 1322 1323 1324 1325 1326 1327 1328 1329 1330 1331 1332 1333 1334 1335 1336 1337 1338 1339 1340 1341 1342 1343 1344 1345 1346 1347 1348 1349 1350 1351 1352 1353 1354 1355 1356 1357 1358 1359 1360 1361 1362 1363 1364 1365 1366 1367 1368 1369 1370 1371 1372 1373 1374 1375 1376 1377 1378 1379 1380 1381 1382 1383 1384 1385 1386 1387 1388 1389 1390 1391 1392 1393 1394 1395 1396 1397 1398 1399 1400 1401 1402 1403 1404 1405 1406 1407 1408 1409 1410 1411 1412 1413 1414 1415 1416 1417 1418 1419 1420 1421 1422 1423 1424 1425 1426 1427 1428 1429 1430 1431 1432 1433 1434 1435 1436 1437 1438 1439 1440 1441 1442 1443 1444 1445 1446 1447 1448 1449 1450 1451 1452 1453 1454 1455 1456 1457 1458 1459 1460 1461 1462 1463 1464 1465 1466 1467 1468 1469 1470 1471 1472 1473 1474 1475 1476 1477 1478 1479 1480 1481 1482 1483 1484 1485 1486 1487 1488 1489 1490 1491 1492 1493 1494 1495 1496 1497 1498 1499 1500 1501 1502 1503 1504 1505 1506 1507 1508 1509 1510 1511 1512 1513 1514 1515 1516 1517 1518 1519 1520 1521 1522 1523 1524 1525 1526 1527 1528 1529 1530 1531 1532 1533 1534 1535 1536 1537 1538 1539 1540 1541 1542 1543 1544 1545 1546 1547 1548 1549 1550 1551 1552 1553 1554 1555 1556 1557 1558 1559 1560 1561 1562 1563 1564 1565 1566 1567 1568 1569 1570 1571 1572 1573 1574 1575 1576 1577 1578 1579 1580 1581 1582 1583 1584 1585 1586 1587 1588 1589 1590 1591 1592 1593 1594 1595 1596 1597 1598 1599 1600 1601 1602 1603 1604 1605 1606 1607 1608 1609 1610 1611 1612 1613 1614 1615 1616 1617 1618 1619 1620 1621 1622 1623 1624 1625 1626 1627 1628 1629 1630 1631 1632 1633 1634 1635 1636 1637 1638 1639 1640 1641 1642 1643 1644 1645 1646 1647 1648 1649 1650 1651 1652 1653 1654 1655 1656 1657 1658 1659 1660 1661 1662 1663 1664 1665 1666 1667 1668 1669 1670 1671 1672 1673 1674 1675 1676 1677 1678 1679 1680 1681 1682 1683 1684 1685 1686 1687 1688 1689 1690 1691 1692 1693 1694 1695 1696 1697 1698 1699 1700 1701 1702 1703 1704 1705 1706 1707 1708 1709 1710 1711 1712 1713 1714 1715 1716 1717 1718 1719 1720 1721 1722 1723 1724 1725 1726 1727 1728 1729 1730 1731 1732 1733 1734 1735 1736 1737 1738 1739 1740 1741 1742 1743 1744 1745 1746 1747 1748 1749 1750 1751 1752 1753 1754 1755 1756 1757 1758 1759 1760 1761 1762 1763 1764 1765 1766 1767 1768 1769 1770 1771 1772 1773 1774 1775 1776 1777 1778 1779 1780 1781 1782 1783 1784 1785 1786 1787 1788 1789 1790 1791 1792 1793 1794 1795 1796 1797 1798 1799 1800 1801 1802 1803 1804 1805 1806 1807 1808 1809 1810 1811 1812 1813 1814 1815 1816 1817 1818 1819 1820 1821 1822 1823 1824 1825 1826 1827 1828 1829 1830 1831 1832 1833 1834 1835 1836 1837 1838 1839 1840 1841 1842 1843 1844 1845 1846 1847 1848 1849 1850 1851 1852 1853 1854 1855 1856 1857 1858 1859 1860 1861 1862 1863 1864 1865 1866 1867 1868 1869 1870 1871 1872 1873 1874 1875 1876 1877 1878 1879 1880 1881 1882 1883 1884 1885 1886 1887 1888 1889 1890 1891 1892 1893 1894 1895 1896 1897 1898 1899 1900 1901 1902 1903 1904 1905 1906 1907 1908 1909 1910 1911 1912 1913 1914 1915 1916 1917 1918 1919 1920 1921 1922 1923 1924 1925 1926 1927 1928 1929 1930 1931 1932 1933 1934 1935 1936 1937 1938 1939 1940 1941 1942 1943 1944 1945 1946 1947 1948 1949 1950 1951 1952 1953 1954 1955 1956 1957 1958 1959 1960 1961 1962 1963 1964 1965 1966 1967 1968 1969 1970 1971 1972 1973 1974 1975 1976 1977 1978 1979 1980 1981 1982 1983 1984 1985 1986 1987 1988 1989 1990 1991 1992 1993 1994 1995 1996 1997 1998 1999 2000 2001 2002 2003 2004 2005 2006 2007 2008 2009 2010 2011 2012 2013 2014 2015 2016 2017 2018 2019 2020 2021 2022 2023 2024 2025 2026 2027 2028 2029 2030 2031 2032 2033 2034 2035 2036 2037 2038 2039 2040 2041 2042 2043 2044 2045 2046 2047 2048 2049 2050 2051 2052 2053 2054 2055 2056 2057 2058 2059 2060 2061 2062 2063 2064 2065 2066 2067 2068 2069 2070 2071 2072 2073 2074 2075 2076 2077 2078 2079 2080 2081 2082 2083 2084 2085 2086 2087 2088 2089 2090 2091 2092 2093 2094 2095 2096 2097 2098 2099 2100 2101 2102 2103 2104 2105 2106 2107 2108 2109 2110 2111 2112 2113 2114 2115 2116 2117 2118 2119 2120 2121 2122 2123 2124 2125 2126 2127 2128 2129 2130 2131 2132 2133 2134 2135 2136 2137 2138 2139 2140 2141 2142 2143 2144 2145 2146 2147 2148 2149 2150 2151 2152 2153 2154 2155 2156 2157 2158 2159 2160 2161 2162 2163 2164 2165 2166 2167 2168 2169 2170 2171 2172 2173 2174 2175 2176 2177 2178 2179 2180 2181 2182 2183 2184 2185 2186 2187 2188 2189 2190 2191 2192 2193 2194 2195 2196 2197 2198 2199 2200 2201 2202 2203 2204 2205 2206 2207 2208 2209 2210 2211 2212 2213 2214 2215 2216 2217 2218 2219 2220 2221 2222 2223 2224 2225 2226 2227 2228 2229 2230 2231 2232 2233 2234 2235 2236 2237 2238 2239 2240 2241 2242 2243 2244 2245 2246 2247 2248 2249 2250 2251 2252 2253 2254 2255 2256 2257 2258 2259 2260 2261 2262 2263 2264 2265 2266 2267 2268 2269 2270 2271 2272 2273 2274 2275 2276 2277 2278 2279 2280 2281 2282 2283 2284 2285 2286 2287 2288 2289 2290 2291 2292 2293 2294 2295 2296 2297 2298 2299 2300 2301 2302 2303 2304 2305 2306 2307 2308 2309 2310 2311 2312 2313 2314 2315 2316 2317 2318 2319 2320 2321 2322 2323 2324 2325 2326 2327 2328 2329 2330 2331 2332 2333 2334 2335 2336 2337 2338 2339 2340 2341 2342 2343 2344 2345 2346 2347 2348 2349 2350 2351 2352 2353 2354 2355 2356 2357 2358 2359 2360 2361 2362 2363 2364 2365 2366 2367 2368 2369 2370 2371 2372 2373 2374 2375 2376 2377 2378 2379 2380 2381 2382 2383 2384 2385 2386 2387 2388 2389 2390 2391 2392 2393 2394 2395 2396 2397 2398 2399 2400 2401 2402 2403 2404 2405 2406 2407 2408 2409 2410 2411 2412 2413 2414 2415 2416 2417 2418 2419 2420 2421 2422 2423 2424 2425 2426 2427 2428 2429 2430 2431 2432 2433 2434 2435 2436 2437 2438 2439 2440 2441 2442 2443 2444 2445 2446 2447 2448 2449 2450 2451 2452 2453 2454 2455 2456 2457 2458 2459 2460 2461 2462 2463 2464 2465 2466 2467 2468 2469 2470 2471 2472 2473 2474 2475 2476 2477 2478 2479 2480 2481 2482 2483 2484 2485 2486 2487 2488 2489 2490 2491 2492 2493 2494 2495 2496 2497 2498 2499 2500 2501 2502 2503 2504 2505 2506 2507 2508 2509 2510 2511 2512 2513 2514 2515 2516 2517 2518 2519 2520 2521 2522 2523 2524 2525 2526 2527 2528 2529 2530 2531 2532 2533 2534 2535 2536 2537 2538 2539 2540 2541 2542 2543 2544 2545 2546 2547 2548 2549 2550 2551 2552 2553 2554 2555 2556 2557 2558 2559 2560 2561 2562 2563 2564 2565 2566 2567 2568 2569 2570 2571 2572 2573 2574 2575 2576 2577 2578 2579 2580 2581 2582 2583 2584 2585 2586 2587 2588 2589 2590 2591 2592 2593 2594 2595 2596 2597 2598 2599 2600 2601 2602 2603 2604 2605 2606 2607 2608 2609 2610 2611 2612 2613 2614 2615 2616 2617 2618 2619 2620 2621 2622 2623 2624 2625 2626 2627 2628 2629 2630 2631 26</pre>
--	---

```

class Key {} class Value {}
class Map {
  /*@ nullable */ Key[] keys;
  /*@ nullable */ Value[] values;
  /*@ ensures (\exists int i;0<=i&&
    @ i<values.length;values[i]==v);
    @ diverges true;
    @ assignable \everything; */
  void put(Key k, Value v){
1   int pos = getIndex0f(k);
2   if (pos>=0)
3     values[pos] = v;
4   else {
5     addKey(k);
6     addValue(v);
7   }
8   int getIndex0f(Key k){
9     int r = -1;
10    for(int i=0;i<keys.length;i++)
11      if (keys[i] == k)
12        r = i;
13    return r;
14  }
15  void addKey(Key k){
16    Key[] oldKs = keys;
17    keys = new Key[keys.length+1];
18    keys[keys.length - 1] = k;
19    for (int i=0;i<oldKs.length;i++)
20      keys[i] = oldKs[i];
21  }
22  void addValue(Value v){
23    Value[] oldVs = values;
24    values=new Value[values.length+1];
25    values[values.length - 1] = v;
26    for (int i=0;i<oldVs.length;i++)
27      values[i] = oldVs[i];
28  }
29 }
}

class Key {} class Value {}
class Map {
  /*@ nullable */ Key[] keys;
  /*@ nullable */ Value[] values;
  /*@ ensures (\exists int i;0<=i&&
    @ i<values.length;values[i]==v);
    @ diverges true;
    @ assignable \everything; */
  void put(Key k, Value v){
1   int pos = pure_int(k);
2   if(pure_boolean())
3     values[pos] = v;
4   else {
5     impure_keys(k);
6     addValue(v);
7   }
8   void addValue(Value v){
9     Value[] oldVs = values;
10    values=new Value[values.length+1];
11    values[values.length - 1] = v;
12    for (int i=0;i<oldVs.length;i++)
13      values[i] = pure_Value();
14  }
15  /*@ assignable \strictly_nothing;
16  native int pure_int();
17  /*@ assignable \strictly_nothing;
18  native boolean pure_boolean();
19  /*@ assignable this.keys;
20  native void impure_keys();
21  /*@ assignable \strictly_nothing;
22  native /*@nullable*/ Value
23      pure_Value();
24  }
}

```

(a) Original program

(b) Abstract program

Fig. 3: A data-structure-rich program to put a key and a value to a map.

`keys` and `values`, respectively, and have the same index in the arrays. The method `put(k,v)` invokes the method `getIndex0f` to check whether `k` already exists in the map. If it exists, the old value is replaced by `v`; otherwise, the methods `addKey` and `addValue` reallocate the arrays `keys` and `values`, respectively, and add `k` and `v` to the new arrays. The `ensures` clause guarantees that the value `v` is in

this map (the `\exists` quantifier). By default, referenced variables are not null, thus the `nullable` clause enables also the `null` value. The abstract program is shown in Fig. 3(b), where the methods `getIndexOf` and `addKey` are irrelevant to the property of interest, thus their call sites (Fig. 3(a), statements 4, 5) are abstracted, such that this fact is directly exposed to the verification engineers. The abstract method `put` contains half of the statements of the original program, relieving the user from the burden of writing some auxiliary specifications. The methods `getIndexOf` and `addKey` are abstracted, thus loop invariants are not needed for them. It is difficult to discover this fact without non-trivial efforts. Besides, the abstraction indicates that the loop (Fig. 3(a), statements 19-20) only modifies locations $[0 \dots (\text{values.length} - 2)]$ of the `values` array, whereas the concrete behaviors to modify the other slots can be left out completely in the loop invariants. The abstract program has been proved with KeY using 3879 rules and 4 auxiliary specifications, while the original program requires 14684 rules and 14 auxiliary specifications.

3 Techniques

We now explain the principle techniques of our approach. We describe: program translation, program abstraction, validity check of counterexamples and refinement of abstract programs, and runtime exception handling. See Fig. 1.

We focus on analyzing object-oriented programs, and currently support a basic subset of Java that does not include floating point numbers, concurrency, and user-defined exceptions. We support a class hierarchy definition without interfaces and abstract classes. A detailed program syntax can be found in a previous work (cf. [23, Sect. 2]). We currently support a basic subset of JML that does not include model fields and exceptional behaviors.

3.1 Translation

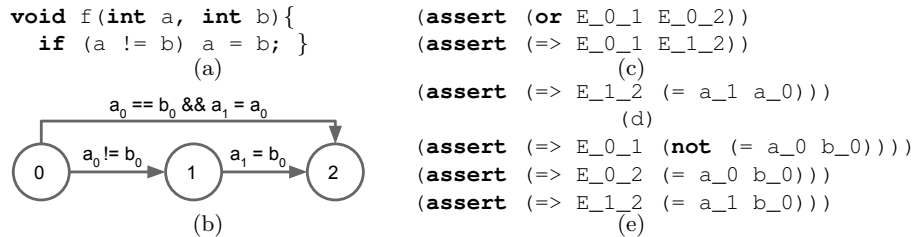


Fig. 4: Encoding: (a) a sample Java code, (b) computation graph, (c) control constraints, (d) frame conditions, (e) data constraints.

We explain the translation techniques of bounded program verification (the “Translation” box in Fig. 1). Based on user-provided bounds we translate a Java program and its JML requirement specifications into an SMT formula. Some code

transformations are performed on the analyzed program before the translation: Loops are unrolled the number of times defined in bounds; methods are inlined into their call sites; constructors are split into object allocation and initialization; and all variables and fields are renamed such that they are assigned at most once. The preprocessed program (called *bounded program* in the paper) is represented using a *computation graph* [27], a directed acyclic graph that has a single entry node and a single exit node. The nodes of the graph represent the control points in the bounded program, and the edges represent the state transitions. Figure 4 provides a simple example. The computation graph of the program in Fig. 4(a) is shown in Fig. 4(b), where variable names are indexes; the initial index is 0, and the index is incremented every time the variable is updated. Figure 4(c) gives the SMT constraints encoding the control flow. An SMT formula consists of logical conjunction-connected SMT constraints (enclosed in the `assert` command). Basic constraint are combined using the boolean operators `and`, `or`, `not`, and `=>` (implies). We introduce a boolean variable `E.i_j` to represent an edge from node `i` to node `j`; the data constraints in Fig. 4(e) provide the correct semantics for state transitions; the frame condition in Fig. 4(d) explicitly prevents variables to be unspecified. Variables (fields) in JML expressions are replaced by the appropriate variables (fields) in the pre-/post-state of the bounded program. More details can be found in a previous work [23].

3.2 Abstraction

When an SMT formula is unsatisfiable, an SMT solver capable of generating proofs is used to find a proof of invalidity, i.e., an *unsat core*. Minimization is performed on the core returned by an SMT solver to ensure the core is locally minimal: removing any single constraint from the core renders it satisfiable. (The algorithm is presented later in this section.) Let the set C denote the inconsistent constraints extracted from the SMT formula; i.e., C encodes the reason that no post-state violates the requirement specification. To discover which statements are responsible for a constraint in the *unsat core*, we maintain a *constraint map* $M := \{C \mapsto S\}$ to store the connection between the constraints C and statements S . When generating data constraints (e.g., Fig. 4(e)), the mapping from a data constraint $c \in C$ to the statement $s \in S$ (where the constraint is generated) is added to the constraint map M . Figure 5 presents the rules used for updating the constraint map M . Rules R_1 and R_2 shows data constraints are directly mapped to the simple assignment statements. We translate the assignments $\mathbf{e.f} = \mathbf{e}$ to two constraints: $e.f' = e$ and $\forall T o, o \neq e \Rightarrow o.f' = o.f$ (T represents the type of e), where only the former is used to update the constraint map M (R_3). The translations of the create statement and the array update statement are handled in the same way (R_4 and R_5 respectively). The rule R_6 shows that the constraints translated from branch conditions are mapped to the branch (or loop) statement. The loop condition is negated after the last iteration, the rule R_7 maps the negation of the loop condition to the loop statement. The statements mapped by the constraints of C are the relevant statements w.r.t. the property under consideration for user-provided bounds.

R₁ : $\mathcal{T}_d[\mathbf{T} \ v = \mathbf{e};, E]$	\rightarrow	$M' = M \cup \{(E \Rightarrow (v_0 = \mathcal{T}_e[e])) \mapsto S\}$
R₂ : $\mathcal{T}_d[\mathbf{v} = \mathbf{e};, E]$	\rightarrow	$M' = M \cup \{(E \Rightarrow (v' = \mathcal{T}_e[e])) \mapsto S\}$
R₃ : $\mathcal{T}_d[\mathbf{e.f} = \mathbf{e};, E]$	\rightarrow	$M' = M \cup \{(E \Rightarrow (e.f' = \mathcal{T}_e[e])) \mapsto S\}$
R₄ : $\mathcal{T}_d[\mathbf{e} = \mathbf{new} \ \mathbf{T};, E]$	\rightarrow	$M' = M \cup \{(E \Rightarrow (e' = \mathcal{T}_e[\mathbf{newT}])) \mapsto S\}$
R₅ : $\mathcal{T}_d[\mathbf{e}[e] = \mathbf{e};, E]$	\rightarrow	$M' = M \cup \{(E \Rightarrow (e[\mathcal{T}_e[e]]) = \mathcal{T}_e[e])) \mapsto S\}$
R₆ : $\mathcal{T}_d[\mathbf{if} \ (\mathbf{e}), E_t, E_f]$	\rightarrow	$M' = M \cup \{(E_t \Rightarrow \mathcal{T}_e[e]) \mapsto S, (E_f \Rightarrow \mathcal{T}_e[!e]) \mapsto S\}$
R₇ : $\mathcal{T}_d[\mathbf{assume} \ (\mathbf{e}), E]$	\rightarrow	$M' = M \cup \{(E \Rightarrow \mathcal{T}_e[e]) \mapsto S\}$

Fig. 5: The rules for updating the constraint map M to M' . The new variables (or fields) are marked with apostrophes. \mathcal{T}_d and \mathcal{T}_e represent the translation of program statements and expressions respectively. S denotes a program statement and E represents the edge for the statement. E_t and E_f denote the outgoing edges of branch statements.

R₁ : $\mathcal{A}[\mathbf{T} \ v = \mathbf{e};]$	$:=$	$\mathbf{T} \ v = \mathbf{pure.T}();$
R₂ : $\mathcal{A}[\mathbf{v} = \mathbf{e};]$	$:=$	$\mathbf{v} = \mathbf{pure.T}();$
R₃ : $\mathcal{A}[\mathbf{e.f} = \mathbf{e};]$	$:=$	$\mathbf{e.f} = \mathbf{pure.T}();$
R₄ : $\mathcal{A}[\mathbf{if} \ (\mathbf{e})]$	$:=$	$\mathbf{if} \ (\mathbf{pure.T}())$
R₅ : $\mathcal{A}[\mathbf{while} \ (\mathbf{e})]$	$:=$	$\mathbf{while} \ (\mathbf{pure.T}())$
R₆ : $\mathcal{A}[\mathbf{return} \ \mathbf{e};]$	$:=$	$\mathbf{return} \ \mathbf{pure.T}();$

Fig. 6: The rules for transforming original statements to abstract statements. The transformation is denoted by \mathcal{A} . The concrete Java statements on the left are replaced by the abstract statements on the right. The `pure.T` methods return unspecified values.

These relevant statements are marked as *mustHave* statements and will not be abstracted. The other statements in the bounded program, that are named *mayHave* statements, are not necessary for bounded verification, but may be helpful in the deductive program verification. We generate an abstract program by over-approximating the behaviors of the statements in the original program when their transformed statements are *mayHave* statements—each transformed statement gets the location of its original statement. Thus all feasible executions of the original program are feasible in the abstract program, but not vice versa. Abstract programs are generated using the abstraction rules in Fig. 6. The original statement (on the left of Fig. 6), from which the *mayHave* statement has been transformed, is replaced with a statement (on the right of Fig. 6) that calls a JML-annotated *pure_T* method `/*@ assignable \strictly_nothing;*/ native /*@ nullable */ T pure_T();`. The JML `assignable` clause ensures that no memory location is changed by the pure method and that distinct unspecified values will be returned by the pure method, T represents an appropriate type required by the original statement, and the pure method returns an unspecified value of T which includes `null` as well. The Java keyword `native` is used to avoid implementations of the pure methods.

Using the rules in Fig. 6, the generated abstract programs provide to the verification engineers the information which statements are necessary for the

properties under consideration. Thus, writing auxiliary specifications could be easier. However, it may increase the proof complexity compared to the concrete programs. Typically, a deductive program verification system (e.g., KeY [1]) symbolic executes a program and applies various calculus rules to make a proof. During symbolic execution, the symbolic states of the original program are very likely more concrete than those of the abstract program. Therefore, the symbolic execution paths which are invalid for the concrete programs are traversed when proving the abstract programs. Furthermore, symbolic execution of an abstract statement may require more rules than its original statement.

We optimize the abstract program. In order for the abstract programs to have appropriate concrete states, statements that are unnecessary for bounded program verification, yet helpful for the deductive program verification, are marked as `mustHave` statements. For example, assignment statements where the expression on the right-hand side is an object allocation, constant, etc., and that their defined variables are used in some `mustHave` statements. When possible, we abstract a set S of `mayHave` statements into one single statement, thus reducing the number of abstract statements. This is available for any two nodes m and n in the computation graph g (see, e.g., Fig. 4(b)), where m dominates n , n post-dominates m , and *all* the statements in S whose edges are in the paths from m to n are `mayHave` statements. The new abstract statement calls an impure method `/* @assignable loc;*/ native T impure_T()`, where the JML specification denotes the memory locations modified by the statements. We compute the modifiable locations `loc` as a collection of the fields and variables that are updated in the `mayHave` statements.

Minimization of an unsat core. A locally minimal unsat core is useful for computing optimal abstractions. To the best of our knowledge, none of the SMT solvers guarantees its unsat core is locally minimal. We present an algorithm (Algorithm 1) that minimizes an unsat core by exhaustively checking whether a constraint is necessary for the unsatisfiability of the SMT formula. If the formula remains unsatisfiable when deactivating (negating) the constraints of a program statement, the constraints are not needed and their statement is a `mayHave` statement. The new unsat core returned by the SMT solver is the input for the next check. Otherwise, we reactivate the constraints and check the other constraints till all constraints are flipped.

Algorithm 1: Minimize an unsat core

Input: \mathcal{C} : unsatisfiable SMT constraints; $S \leftarrow \emptyset$: unnecessary constraints;
 $muc \leftarrow \emptyset$: locally minimal unsat core.
for $c \in \mathcal{C}$ **do**
 if $c \notin S$ **then**
 if (*check-sat* $((\mathcal{C} - c) \setminus S)$) *is UNSAT* **then**
 $muc \leftarrow \text{getUnsatCoreFromSolver}()$;
 $S \leftarrow S \cup ((\mathcal{C} - c) \setminus muc)$
return muc

3.3 Validity Check and Refinement

To check the validity of the counterexamples, new bounds are always required since the abstract program fulfills the property w.r.t. the old bounds. For a counterexample ce , the new bound of class C is $CPre_{ce} + \max(CPath_1, \dots, CPath_n)$, where $CPre_{ce}$ is the number of C instances in ce , $CPath_n$ is the number of allocations of C instances on the n -th program path, and function \max returns the maximum. To compute new loop unrolls, we transform a `while(cond){stmts;}` loop into `if(cond){stmts; if(!cond)var=var;}`, where `var` is a variable that is modifiable in `stmts`. This transformation prevents unrolling the loops that are irrelevant for the program correctness.

We compute a new SMT formula that is the conjunction of the translation of the counterexample and the translation of the original program for the new bounds. When the formula is satisfiable, then either the counterexample is valid, or the loop requires further iterations if the loop condition is still `true` after traversing the last iteration. In the latter case, we double the loop bounds and repeat the validity check. If the formula is unsatisfiable, we find the statements w.r.t. the counterexample using the techniques as shown in Sect. 3.2. In the bounded program, we highlight a `mayHave` statement as a `mustHave` statement when the statement is in the newly found statements. Finally, using the technique shown in Sect. 3.2, we generate a new abstract program for deductive verification.

3.4 Runtime Exceptions

For each property to be proved, verification systems also prove that no runtime exception is thrown. When more than one functional property has to be verified, the same proof steps for checking runtime exceptions are redone. Our approach separates the verification of functional properties from checking runtime exceptions: usually the statement `o = o.f`, e.g., is translated into $(o \neq null \Rightarrow o' = o.f) \vee (o = null \Rightarrow exc)$, where `exc` denotes runtime exception, whereas we translate it into $o \neq null \wedge o' = o.f$. To check that there are no runtime exceptions, we also inject guards into the code, such that if a guard passes an exception is thrown. We treat the possible exception types separately.

Figure 7 presents the code from Fig. 2(a) with one guard injected. We insert a guard (statements 11-13) which sets to true the flag `NASE` in the class `RTE` if a `NegativeArraySizeException` is about to be thrown (statement 14). Thus, when the program in Fig. 7 preserves the value of the exception flag (it is false when calling the method and when returning from it⁴), no `NegativeArraySizeException` is thrown in the original program, as the guard is checking the statement at line 14. All program parts not relevant to whether the exception is thrown are abstracted. In our approach, when there is no runtime exception and the functional properties have been fulfilled by the analyzed abstract programs, the original program is also verified.

⁴ The `requires` clause specifies the method's precondition.

```

    /*@ requires RTE.NASE = false;
       @ ensures RTE.NASE = false;
       @ diverges true;
       @ assignable \everything;*/
int numberOfPrime(int x, int y) {
    ...// statements 1-9 are omitted to save space.
10  if (size>0){
11      if (y-x < 0) {
12          RTE.NASE = true;
13          return;
14      }
15      this.a = new int[y-x];
16  }
17  return size;}

```

Fig. 7: The example from Fig. 2(a) with an injected guard.

4 Evaluation

The approach that we have presented (i) liberates verification engineers from finding the relevant program slices manually, and (ii) reduces the proof complexity especially for partial properties, for which most of the program slices are irrelevant.

We have implemented the techniques introduced in the paper in a prototype tool, *AbstractJ*. We use InspectJ [23] as the bounded verification tool and KeY [1] as the deductive verification tool. The KeY system performs *symbolic execution* [21] of sequential Java programs, using various calculus rules. Program verification with KeY is usually done in *auto-active* style: the user interacts with the system only through provided auxiliary specifications, while the proof result is obtained automatically. The number of rule applications is our primary measure of proof complexity. We have used 5 benchmark programs, all taken from the related program verification literature and from the KeY repository. Each program has 2 to 6 partial properties to be verified. We have considered also two other approaches to evaluate the effectiveness of our approach (*abstraction*) in program verification. One approach, *baseline*, proves the original programs using KeY as usual. The other approach, *highlight*, is similar to the *abstraction* approach, but it only highlights the relevant program statements and retains the irrelevant statements rather than abstracting them. We have completed 21 verification tasks using each approach, and in total we have completed 63 (= 21 * 3) verification tasks in our experiments. We have written the auxiliary specifications as compact as possible and measured the auxiliary specifications as the number of the operands of JML expressions, JML constructs, and logical connectors, e.g., `loop_invariant`, `assignable`, `forall`, `&&`, etc.⁵ We used the SMT solver Z3 [25] to compute the unsat cores. For the experiments described in this

⁵ Different engineers may write different auxiliary specifications for the same programs. We have asked an experienced KeY engineer to prove the original programs and a relatively inexperienced KeY user to prove the abstract programs. They carefully

Table 1: Evaluation Results

method	properties	origin	baseline		highlight		abstraction		
		stmts	specs	rules	specs	rules	stmts	specs	rules
List. merge(list)	nullPointer	27	22	3578	12	3046	4	0	196
	indexBounds	43	59	4641	46	4434	33	46	3717
	negSize	31	13	4316	14	2723	16	6	1188
	leElems	22	14	2962	14	2962	13	6	1715
	subset	22	82	6299	56	5715	15	52	4404
Map. put(key,value)	nullPointer	32	28	4485	14	3780	9	0	512
	indexBounds	48	61	6154	54	5557	48	54	5488
	negSize	32	17	4084	12	3753	16	0	654
	oldKey	26	30	4295	30	4295	11	22	1725
	sameValues	26	27	9823	34	8494	12	26	4647
	kvMatched	26	50	7327	50	7327	26	50	8814
LRS. doLRS()	nullPointer	39	11	3022	8	2818	13	0	753
	indexBounds	43	44	5006	14	4545	30	14	4502
	foundOrNot	26	32	4155	14	2908	17	10	1255
Set. intersect(set)	nullPointer	48	23	10937	18	10226	25	6	5505
	negSize	38	17	14555	14	9963	23	10	4586
	indexBounds	58	57	19715	33	12287	51	33	6714
	emptySet	33	94	64807	46	13557	16	38	3875
	subset	33	142	RO	60	136225	16	52	11211
Graph. remove(nodes)	sameNodes	54	78	RO	60	14985	13	39	3923
	sameEdges	54	119	RO	83	RO	18	67	12334

paper, we have used the default minimal bounds of InspectJ—at most 3 objects and at most 3 loop iterations. All experiments⁶ have been performed on an Intel Core i5-2520M CPU with 2.50 GHz running on a 64-bit Linux.

To evaluate the effect of the *abstraction* approach on reducing the complexity of programs, we have compared the number of Java statements of original and abstract programs. The results are shown in Table 1. The column *method* shows the Java class and its method to be verified; the verified properties are listed in the column *properties*. The *nullPointer*, *indexBounds*, and *negSize* represent the runtime exceptions `NullPointerException`, `ArrayIndexOutOfBoundsException`, and `NegativeArraySizeException`, respectively. The *orgStmts* column displays the number of the original program statements.⁷ The column *stmts* shows the number of the program statements that have been generated by the *abstraction* approach. On average, 49.5% (median 50%, maximum 85.2%) of statements in the original programs have been abstracted by the *abstraction* approach. There are 2 properties (*indexBounds*, and *kvMatched* for the method *put*) for which

inspected and ensured that the annotations are compact enough w.r.t. the requirement specifications.

⁶ The complete experiments can be found at <http://asa.iti.kit.edu/458.php>.

⁷ The injected guard statements are treated as original statements when handling runtime exceptions.

the approach *abstraction* seems has no effect. A careful inspection reveals that one single concrete statement is abstracted. From the results, the abstract programs contain less, yet enough details for partial properties. The more partial the verified property, the fewer details the abstract programs have. Conservatively speaking, even in the case where the abstract programs are identical to the original programs, the *abstraction* approach assists verification engineers at exploring the relevant statements—all program statements that have not been abstracted are necessary for the properties under consideration. The *highlight* approach shows the relevant program statements to verification engineers, while the *abstraction* approach provides additional benefits: (i) automatic generation of auxiliary specifications for the irrelevant program statements, and (ii) *possible* reduction of proof complexity for partial properties. Besides, the *abstraction* can increase users confidence in the correctness of their programs, before starting deductive verification.

For a fair comparison of the amount of manually written auxiliary specifications, the *highlight* approach reused the auxiliary specifications that have been written manually in the *abstraction* approach (shown in the column *specs* of the column *abstraction* in Table 1). The *abstraction* approach generates annotations for the unnecessary program slices, for which the verification engineers need to write annotations using the *highlight* approach. On average, 37.2% (median 26.7%) of annotations for the highlighted programs have been automatically generated by the *abstraction* approach.

All properties in Table 1 have been proved using the *abstraction* approach. When using the approaches *highlight* and *baseline*, several properties are unprovable. The column *rules* provides the number of rule applications. Any rule application beyond our threshold of 2000000⁸ is denoted by *RO*. For 18 properties that have been proved by all approaches, the *abstraction* approach needed only 50.1% (median 55.2%) of the rules required by the *highlight* approach. It is not guaranteed that the *abstraction* approach requires less rule applications than the other two approaches for arbitrary properties. Besides of the reasons talked in Sect. 3.2, KeY creates branches for each abstract statement, to check its pre-/post-conditions.⁹ When the rule cost introduced by the abstract statements is lower than the cost of symbolic execution of the irrelevant original statements, only then the *abstraction* approach requires fewer rules than other approaches, by assuming they use same auxiliary specifications. In other words, the more partial the verified property, the less proof complexity of the abstract programs. The property *kvMatched* is an example for less partial property.

Although we used small bounds for InspectJ in the experiments, there are no refinement cases in Table 1. On the other hand, when the refinement of an abstract program is needed, the abstract program will contain much less details, thus it is easy to find the relevant program statements. The verification engineers are free to provide even higher bounds for InspectJ. Given the same input formula,

⁸ The time cost and memory consumption grow exponentially w.r.t. the rule applications. It required ~ 30 min and more than 4 GB memory for 2000000 rules.

⁹ The trivial pre-/post-conditions of each abstract statement requires ~ 20 -100 rules.

Z3 may find an unsat core that is different from the core found by other SMT solvers. AbstractJ may generate different abstract programs using other SMT solver, but the abstract programs will still contain less details than the concrete programs if the analyzed property is partial enough.

5 Related Work

Several methods have been proposed to split the program under analysis with respect to particular concerns. Traditional program slicing techniques (e.g., static/dynamic slicing) generate a group of accessible statement (a slice) w.r.t. variables of interest at particular locations. Due to the complexity of the specification expressions and various data structures in the analyzed programs, it is very difficult to find specification-sensitive slices correctly.

Conditioned slicing techniques [4, 8, 9, 12, 13, 17] have been widely applied to simplify programs with respect to the specifications. Comuzzi et al. [12] introduced predicates as a slicing criterion; the slice contains the statements affecting the predicates. That idea has been extended by introducing preconditions [9], symbolic execution [4], and program verification [13] into conditioned slicing techniques. Typically, conditioned slicing produces a group of all accessible statements w.r.t. the specification by symbolic execution with the inputs generated by a solver. The pre/postcondition (generally formulas of first-order logic) are expressed in terms of the (input) variables at program locations of interest. However, intensive human interaction is required to guide the symbolic execution by choosing a suitable criterion. GamaSlicer [13] verifies the program w.r.t. specifications before generating semantic-based slices. Nevertheless, it may not terminate with a conclusive result, since it targets an undecidable logic. Our approach ensures that the soundness of the proof depends only on the deductive verification.

The following three approaches tried to improve the verification process using bounded analysis. Bormer et al. [6] claim that verifying programs using the bounded model checker LLBMC [24] facilitates proving with VCC [11]. Annotations written in VCCs specification language are translated into assertions that can be checked by LLBMC. El Ghazi et al. [16] try to verify Alloy problems using deductive verification, after the Alloy analyzer [20]—based on bounded analysis, fails in finding a counterexample in bounds dictated by the machine. Kroening et al. [15] combine k -induction and inductive invariant method to facilitate program verification using significantly weaker annotations. These approaches do not aim to reduce the overhead of writing specifications. However, the k -induction frequently allows using weaker loop invariants than are required by the inductive invariant approach. Our approach can reduce the burden of specifications not only for loops.

Using unsat core is not new in bounded program verification. The authors of [26] used the unsat core to refine the method summaries in program verification. In [14], a code coverage metric is constructed by the program statements that are mapped from the unsat core.

Counterexample-guided abstraction refinement (CEGAR) has been widely used in program verification. To the best of our knowledge, the abstractions have been constructed mostly at the predicate level [2, 5, 7, 10, 18] and rarely at the function level [26]. Our approach constructs the abstractions at the levels including the ones mentioned above and statement level.

6 Conclusion and Future Work

We presented a novel method to compute specification-sensitive abstractions for program verification. The abstractions are constructed with the help of bounded program verification. The counterexample-guided refinement framework has been used to refine the abstractions. We exploited the characteristics of the unsat core to discover irrelevant statements. The novelty of our approach is to abstract the program statements that are irrelevant for the properties of interest, to help verification engineers to write auxiliary specifications. We described how to: encode programs, map program statements to constraints, generate abstractions based on abstraction rules, and refine the abstractions with new bounds computation. We evaluated our experiments on 5 programs that were already used in related papers and in the KeY repository. Initial results show that our approach generates suitable abstract programs for verification, and all abstract programs have been proved for all 21 properties, while the original programs have been proved for 18 properties. Our tool took off 50% of the user's workload in writing auxiliary specifications. Only about half of the proof rules used to prove the original program are needed for proving the abstract program.

We plan to apply our approach to larger programs, and investigate incorporating loop invariant generators, e.g., Invgen [19], to improve the automation of the approach.

Acknowledgement. This work has been partially supported by GIF (grant No. 1131-9.6/2011) and by DFG under project “DeduSec” within SPP 1496 “RS³” and by BMBF under project FIFAKS within the Software Campus program.

References

1. Ahrendt, W., Beckert, B., Bruns, D., Bubel, R., Gladisch, C., Grebing, S., Hähnle, R., Hentschel, M., Herda, M., Klebanov, V., Mostowski, W., Scheben, C., Schmitt, P.H., Ulbrich, M.: The KeY platform for verification and analysis of Java programs. In: VSTTE. LNCS, vol. 8471, pp. 55–71. Springer (2014)
2. Ball, T., Cook, B., Levin, V., Rajamani, S.K.: SLAM and static driver verifier: Technology transfer of formal methods inside Microsoft. In: iFM. pp. 1–20. LNCS, Springer (2004)
3. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB standard: Version 2.5. Tech. rep., The University of Iowa (2015)
4. Barros, J.B., Carneiro da Cruz, D., Rangel Henriques, P., Sousa Pinto, J.: Assertion-based slicing and slice graphs. *Formal Asp. Comput* 24(2), 217–248 (2012)

5. Beyer, D., Henzinger, T.A., Jhala, R., Majumdar, R.: The software model checker Blast. *STTT* 9(5-6), 505–525 (2007)
6. Bormer, T.: Advancing Deductive Program-Level Verification for Real-World Application: Lessons Learned from an Industrial Case Study. Ph.D. thesis, KIT (2014)
7. Chaki, S., Clarke, E.M., Groce, A., Jha, S., Veith, H.: Modular verification of software components in C. *IEEE Trans. Software Eng* 30(6), 388–402 (2004)
8. Chebaro, O., Kosmatov, N., Giorgetti, A., Julliand, J.: Program slicing enhances a verification technique combining static and dynamic analysis. In: SAC. pp. 1284–1291. ACM (2012)
9. Chung, I.S.: Program slicing based on specification. In: SAC. pp. 605–609. ACM (2001)
10. Clarke, E.M., Kroening, D., Sharygina, N., Yorav, K.: SATABS: SAT-based predicate abstraction for ANSI-C. In: TACAS. pp. 570–574. LNCS, Springer (2005)
11. Cohen, E., Dahlweid, M., Hillebrand, M., Leinenbach, D., Moskal, M., Santen, T., Schulte, W., Tobies, S.: VCC: A practical system for verifying concurrent C. In: TPHOLs. LNCS, vol. 5674, pp. 23–42. Springer (2009)
12. Comuzzi, J.J., Hart, J.M.: Program slicing using weakest preconditions. In: FME. LNCS, vol. 1051, pp. 557–575. Springer (1996)
13. Carneiro da Cruz, D., Rangel Henriques, P., Sousa Pinto, J.: GamaSlicer: an online laboratory for program verification and analysis. In: LDTA. ACM (2010)
14. Dennis, G.D.: A Relational Framework for Bounded Program Verification. Ph.D. thesis, MIT (2009)
15. Donaldson, A.F., Haller, L., Kroening, D., Rümmer, P.: Software verification using k-induction. In: SAS. pp. 351–368 (2011)
16. El Ghazi, A.A., Ulbrich, M., Gladisch, C., Tyszbrowicz, S., Taghdiri, M.: JKelloy: A proof assistant for relational specifications of Java programs. In: NFM. pp. 173–187 (2014)
17. Fox, C., Danicic, S., Harman, M., Hierons, R.M.: CONSIT: a fully automated conditioned program slicer. *Software: Practice and Experience* 34(1), 15–46 (2004)
18. Gupta, A., Popeea, C., Rybalchenko, A.: Predicate abstraction and refinement for verifying multi-threaded programs. *ACM SIGPLAN Notices* 46(1), 331–344 (2011)
19. Gupta, A., Rybalchenko, A.: Invgen: An efficient invariant generator. In: CAV. LNCS, vol. 5643, pp. 634–640. Springer (2009)
20. Jackson, D.: *Software Abstractions: Logic, Language, and Analysis*. The MIT Press (2012)
21. King, J.C.: Symbolic execution and program testing. *CACM* 19(7), 385–394 (1976)
22. Leavens, G.T., Baker, A.L., Ruby, C.: Preliminary design of JML: a behavioral interface specification language for Java. *ACM SIGSOFT SEN* 31(3), 1–38 (2006)
23. Liu, T., Nagel, M., Taghdiri, M.: Bounded program verification using an SMT solver: A case study. In: ICST. pp. 101–110. IEEE (2012)
24. Merz, F., Falke, S., Sinz, C.: LLBMC: Bounded model checking of C and C++ programs using a compiler IR. In: VSTTE. pp. 146–161. Springer (2012)
25. de Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: TACAS. LNCS, vol. 4963, pp. 337–340. Springer (2008)
26. Taghdiri, M., Jackson, D.: Inferring specifications to detect errors in code. *Automated Software Engineering* 14(1), 87–121 (2007)
27. Vaziri, M.: Finding Bugs in Software with a Constraint Solver. Ph.D. thesis, MIT (2004)