Automated Program Checking via Action Planning

Stefan Edelkamp
University of Bremen
Germany

Mark Kellershoff
Dortmund University of Technology
Germany

Damian Sulewski
University of Bremen
Germany

Abstract

In this paper we translate concurrent C/C++ code into PDDL. The system then runs heuristic search planners against the PDDL outcome to generate traces for locating programming bugs. These counter-examples result in an interactive debugging aid and exploit efficient planner in-built heuristics. Different aspects like parsing, generation of the dependency graph, slicing, abstraction, and property conversion are described. For data abstraction we provide a library, and for increased usability the tool has been integrated in Eclipse.

Introduction

Planning via model checking (Cimatti et al. 1997) considers the integration of verification technology into AI planners. For model checking via planning by considering the rising effectiveness of planning search heuristics in verification (Wehrle and Helmert 2009), a natural question is to apply planning technology directly. The effectiveness of translating model checking inputs into PDDL has been documented by a series of preceding papers, including the *communication protocols* (Edelkamp 2003), *Petri nets* (Edelkamp and Jabbar 2006), and *μ-calculus formulae* (Bakera et al. 2008), and *graph transition systems* (Edelkamp, Jabbar, and Lluch-Lafuente 2005).

In contrast to *model checking* (Clarke, Grumberg, and Peled 1999) that relies on a model of the system to be checked, *program checking* (Visser et al. 2003) aims at the automated verification of programs given its source code by analyzing the compiled executable. Typically, tools operate on top of a virtual machine that has been extended to simulate different execution branches. Verification units that consider checking the object code like *JPF* (Visser et al. 2000) *Steam* (Leven, Mehler, and Edelkamp 2004) and *MoonWalker* (de Brugh, Nguyen, and Ruys 2009), complement bounded software model checking tools like *CBMC* (Clarke, Kroening, and Lerda 2004).

This paper proposes the automated transformation of C/C++ sources into PDDL (Fox and Long 2003) to exploit refined guidance inherent to heuristic search planners. The rationale of applying heuristics is that *directed* model checkers (Edelkamp et al. 2008) quickly report short counter-examples. The rationale for a PDDL encoding are accurate planning heuristics (Helmert and Domshlak 2009).

The choice of the C/C++ is urged by its wide-spread use, rising thread parallelism in programs for the support of multi-core machines, and the lack of advanced bug-finding support. Tools like *valgrind* are able to find memory leaks, but not to validate concurrent programs. As we concentrate on the imperative core of C/C++, the results likely generalize to other programming languages like Java or Ruby.

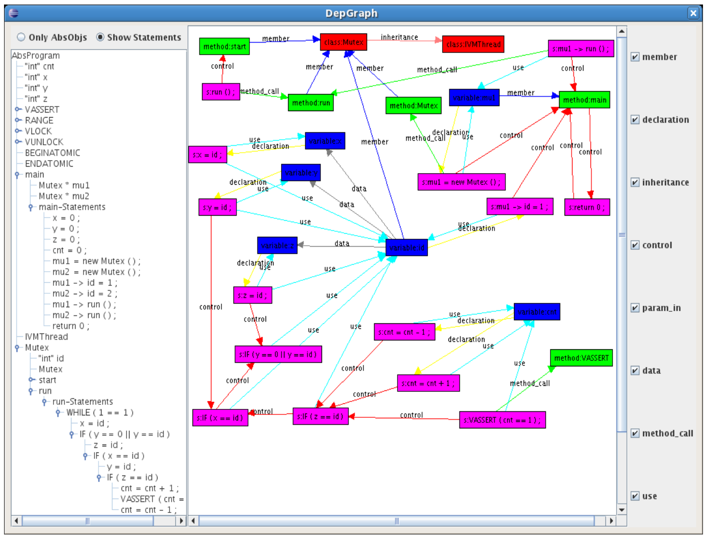
The transformation into Level 2 PDDL is able to directly uncover bugs. For the case a program cannot be analyzed completely, different abstractions apply. Besides *slicing* the program without loss of information, *data abstraction* converts infinite state variables to finite domain, and to Level 1 PDDL. Dependencies among variables are automatically detected by analyzing the parse of the source.

Annotated Parse and Dependency Graph

In program checking, sources are analyzed that have non-deterministic effects. Such non-determinism can be due to the interleaving of concurrent threads, unknown assignments to variables, program and user inputs, explicit choice points imposed by the programmer, or abstractions of deterministic programs.

As C/C++ is a rather complex language (Stroustrup 1994), we adapted JavaCC by Sreenivasa Viswanadha (published in 1997) to parse the input. The parser yields an abstract syntax tree, which we present as a navigational aid to the programmer, and which is used for further processing. For verification, the C/C++ code is annotated with a small set of commands for its controlled execution:

- `VLOCK(<variable>)` restricts the access to the variable `<variable>` in the currently invoked thread.
- `VUNLOCK(<variable>)` releases the lock to the variable `<variable>`.
- `BEGINATOMIC()` dictates that the current thread cannot be suspended.
- `ENDATOMIC()` terminates the atomic block selection within a thread.
- `VASSERT(<condition>)` tracks `<condition>` to be satisfied each time the program reaches it.
- `RANGE(<variable>,<low>,<high>)` offers non-deterministic choices to a program.



```
AbsProgram
├ "int" cnt
├ "int" x
├ "int" y
├ "int" z
├ VASSERT
├ RANGE
├ VLOCK
├ VUNLOCK
├ BEGINATOMIC
├ ENDATOMIC
├ main
│  ├ Mutex * mu1
│  ├ Mutex * mu2
│  └ main-Statements
│     ├ x = 0 ;
│     ├ y = 0 ;
│     ├ z = 0 ;
│     ├ cnt = 0 ;
│     ├ mu1 = new Mutex ( ) ;
│     ├ mu2 = new Mutex ( ) ;
│     ├ mu1 -> id = 1 ;
│     ├ mu2 -> id = 2 ;
│     ├ mu1 -> run ( ) ;
│     ├ mu2 -> run ( ) ;
│     └ return 0 ;
├ IVMThread
└ Mutex
   ├ "int" id
   ├ Mutex
   ├ start
   └ run
      └ run-Statements
         └ WHILE ( 1 == 1 )
            ├ x = id ;
            └ IF ( y == 0 || y == id )
               ├ z = id ;
               └ IF ( x == id )
                  ├ y = id ;
                  └ IF ( z == id )
                     ├ cnt = cnt + 1 ;
                     ├ VASSERT ( cnt =
                     └ cnt = cnt - 1 ;
```

Figure 1: Dependency graph of a C/C++ program.

For transforming the source code, the input has to be made avaailable in a dictionary data structure, while supporting the conversion from infix (as used in C/C++) to prefix notation (as used in PDDL). We implemented a hierarchy of containers. Its structure represents the scopes of a program and is exploited for constructing the object-oriented *dependency graph*. An example is shown in Fig.1 together with parts of the parse tree to its left.

The state of a C/C++ program includes information like assignments to global and local variables, as well as stack and dynamic memory contents. We assume variables of type boolean, integer and real. The situation before the execution of a program is called initial state, and the state of a program at its termination is called (valid) end state. Additionally to the variable assignments, a state contains information about the program counter, denoting which transition has been or will be executed. If the program is multi-threaded, a program counter is maintained for every running thread including the thread for *main*. For the conversion, we assume that a static analysis, applied after parsing the code, can detect the number of threads running concurrently.

The Translation into PDDL

The core motivation of translating the source of a program into PDDL is to use planner in-built heuristic to drive the exploration process towards falsifying a property, e.g. in form of a deadlock, a failed assertion/global invariance, or an array access violation.

The output pleases the first two levels from the PDDL hierarchy (Fox and Long 2003). In PDDL Level 1, states are collections of true facts. It allows quantification over domain objects, disjunctive and negated preconditions, as well as conditional effects. In PDDL Level 2, real-valued fluents are available. Preconditions of actions cover arithmetic expressions over the variables, while the effects can additionally modify value assignments.

Conversion of Variables For a C/C++ variable *declaration*, like *int a*, we reserve a PDDL variable *int_a* (allowing real value assignments). Since a program can contain several variables with name a, every PDDL variable is suffixed with an additional id, such that for our case we infer *int_a_1*, as it is the first (and only) appearance of a that is converted. As a can appear in different threads, we provide an additional parameter to the PDDL predicate, yielding the expression *(int_a_1 ?t - thread)* to represent the variable declaration of a. For variable *int b* the conversion is analogous. In short terms, variable conversion is a mapping that assigns a planning variable to each program variable.

Variable Assignments For translating an *assignment* to a variable into PDDL, we construct actions, which convert the state in the planning model in the same way it does within the program. As we have the parse of the expression available, the conversion from in- to prefix notation is immediate. The parameter is the thread, while the effect changes the planning state equivalent to the assignment *a=1000;*:

```
(:action SimpleAssignment
:parameters (?t - thread)
:precondition (<predecessor has finished>)
:effect (and (assign (int_a_1 ?t) 1000 )
     (<this action has finished>)
     (not (<predecessor has finished>))))
```

Control Flow We use predicates to model line numbers. Every action includes as a precondition that the predecessor (line) has finished its execution. For example, in a sequential execution line 20 has to finish before line 21 is processed. To avoid ambiguities, every line number is attached to the file in which the line is contained. It is also parameterized with the thread that is invoked. Since in PDDL every possible action is checked for execution, preconditions have been enlarged to select the actions that are currently activated.

The action for the first line in the main program includes *(start T0)* as a precondition triggered by the initial state, since it does not have a direct predecessor. The first instruction of a method also contains a label that it has been called.

Consider the following simple example program fragment

```
#include "Thread.h"
[...]
class Example:public Thread {
public:
 Example();
 void run();
};
Example::Example(){}
  void Example::run(){
    int a;
    int b;
    a=1000;
    b=20;
    VASSERT(a < b);
 }
```

After calling *run*, we have two concurrent threads: *main* (thread t_0), and the thread (t_1) that has been called. If we omit the details for thread invocation, the remaining program logic has to include the variables a and b, their order and the constraints imposed. Hence, the PDDL equivalent for the assignments *a=1000* and *b=20* is as follows.

```
(:action Example_cpp_Line_20
:parameters (?t - thread)
:precondition (Example_cpp_Line_19 ?t)
:effect (and (assign (int_a_1 ?t) 1000 )
    (Example_cpp_Line_20 ?t)
    (not (Example_cpp_Line_19 ?t))))
(:action Example_cpp_Line_21
:parameters (?t - thread)
:precondition (Example_cpp_Line_20 ?t)
:effect (and (assign (int_b_2 ?t) 20)
    (Example_cpp_Line_21 ?t)
    (not (Example_cpp_Line_20 ?t))))
```

An *if-statement* exists in two different variants (with or without else block). A model without an else-branch is not directly convertible in PDDL, as failing the if-condition would not increase the program counter. We observe that every action has access to its immediate predecessor, and every action has at most two successors in case of a branch and two predecessors in case of a join. A *while-statement* is an if-statement featuring a backward jump.

For if-statements we introduce a *virtual-else* branch. The if-statement itself would vanish as the conditions are imposed as additional preconditions to the actions. But without modeling the if-statements explicitly, there is a subtle problem in modeling nested if-statements. Consider the small extension of the running example in Fig. 2. If one uses one action per instruction, then implementing correct precedences among the if statements is tricky, i.e., to connect an else-branch to the corresponding if. Therefore, we decided to include an additional flag and an additional action for starting and ending an if- or else- part.

```
void Example::run(){
  int a;
  int b;
  a=1000;
  b=20;
  if (a > b) {
    if (a > 20) {
        a=20;
    } else {
        a=0;
    }
  } else {
    a=1000;
  }
  VASSERT(a < b);
}
```

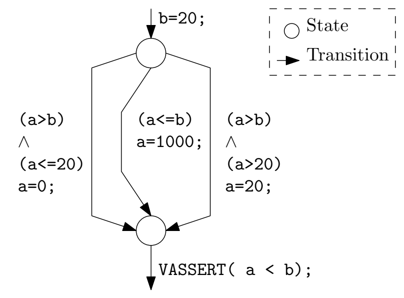


Figure 2: A nested if-statement and induced control flow.

Fig. 3 relates the source of a simple while-statement to the according automaton, that is used to monitor the flow of control in the PDDL code.

Model Checking Statements An *assert-statement* is split into two parts. One branch considers the violation of the assertion, in which case the predicate *assertionviolation* is set, the other branch continues with the flow of instructions.

```
void Example::run(){
  int a;
  int b;

  a=1000;
  b=20;
  while (a > b) {
      a=20;
      b=1000;
  }
  VASSERT(a < b);
}
```

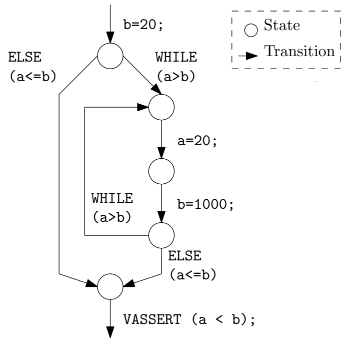


Figure 3: A while program and induced control flow.

When searching for the violation of safety properties, this predicate is included as a goal condition.

The *lock-statement* denotes that a thread requires exclusive access to a variable. The first thread that locks the variable has top priority, such that all upcoming accesses to the same variable are rejected. The PDDL model is extended in the sense that the actions include a precondition for locked variables, while avoiding multiple locks. A proper locking mechanism yields checks of invalid end states. A supplementary action wait is generated, that indicates that a thread waits for a resource. If all threads are blocked, a deadlock has occurred, an a goal achiever is triggered.

For *atomic blocks*, in the PDDL model 2 new predicates are inserted: *atomic* denoting that the execution is in atomic mode, and *isatomic ?t - thread* denoting which thread is actually atomic. Most ordinary actions are extended by the following precondition $atomic \Rightarrow (isatomic\ ?t)$. The end of the block generates an action without further specialized preconditions that deletes *atomic* and (*isatomic ?t)*.

Complex Statements For *indexed variable access* first the index is determined then the access to the array is executed. As PDDL does not provide a mechanism to index variables with numbers, we allow the user to adjust upper bounds provided by the parser (in PDDL 3.1, indirect variable access is available but only a few planners support the extension). The conversion of *C++-objects* into PDDL is possible, if the initialization uses the new-operator and gets assigned to a unique name. The *new-statement* induces the reservation of a PDDL object with a reference to this object; the variables of the class contain an additional parameter, whose type is the class name.

Methods PDDL models cannot generate objects dynamically. The only *methods* that are currently supported are those that have $integer^* \rightarrow integer$ or $integer^* \rightarrow void$ in their signature.

Methods are converted in actions that are triggered by setting a special predicate. The parameters are found on the method-stack, and the solutions are found in a special solution register, accessible from the calling action, similar to what is done in an ordinary executable. Actions are indexed s.t. more than one call is possible.

	Mutex			Producer-Consumer			BubbleSort10			8-Puzzle		
Program	l	s	t	t	s	t	l	s	t	l	s	t
StEAM DFS	66	742	90	122	353	59	594	1184	124	36096	70366	6552
StEAM BF	30	1630	198	29	4383	690	594	1774	88	86	7836	269
FF EHC	27	747	6	20	23	2	-	-	-	-	-	-
FF Best-First	27	251	4	20	3405	10	-	-	-	-	-	-
FF DA EHC	28	39	28	20	33	2	661	670	1312	115	932	2585
MFF DA EHC	28	39	108	20	34	56	661	670	8938	361	56257	1177572
MFF DA BF	27	922	877	20	3405	1609	661	73294	79674	99	4113	96291

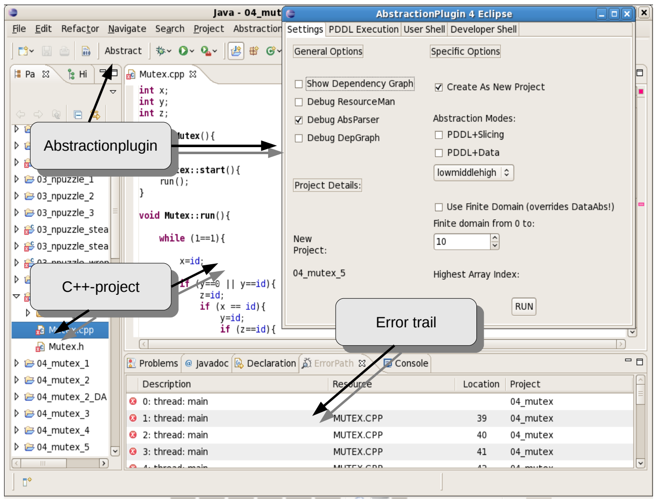
Figure 4: Results for benchmarks; l denotes the length of the counter-example, s the number of states, t the CPU time in milliseconds, DA data abstraction, BF best-first search, and EHC enforced hill climbing applied in FF/MFF (MetricFF).

Abstraction We mainly support *data abstraction* (Merino et al. 2002). In the *neg-pos-zero* abstraction, for example, integers are projected to three values of being either positive negative, or equal to zero. If two negative or two positive values are multiplied, the result is determined, while for mixed multiplication, different options are possible. An alternative is an *odd-even* abstraction with obvious semantics.

Numerical abstractions have be implemented using abstraction libraries. The interface serves as a macro that is automatically extended to enrich the initial state and the PDDL operators to realize abstraction. The dependency graph then helps to deduce the set of all variables that are affected.

Programming Environment

The implementation used the following components: Eclipse 3.3 - Europa + CDT, the planner Metric-FF (Hoffmann 2003), and Java SDK 6 (includes Java-Script). The abstraction plugin that we have developed (see Fig. 5) consists of the GUI for parameterizing the algorithms, the parser, the dependency graph data structure, and the error trailer.



```
int x;
int y;
int z;
utex(){
ex::start(){
   run();
}
void Mutex::run(){
   while (1==1){
      x=id;
      if (y==0 || y==id){
         z=id;
         if (x == id){
            y=id;
            if (z==id){
```

[ ] Show Dependency Graph
[ ] Debug ResourceMan
[✓] Debug AbsParser
[ ] Debug DepGraph
[✓] Create As New Project
Abstraction Modes:
[ ] PDDL+Slicing
[ ] PDDL+Data
lowmiddlehigh
[ ] Use Finite Domain (overrides DataAbs!)
Finite domain from 0 to: 10
New Project: 04_mutex_5
Highest Array Index:

Description	Resource	Location	Project
0: thread: main			04_mutex
1: thread: main	MUTEX.CPP	39	04_mutex
2: thread: main	MUTEX.CPP	40	04_mutex
3: thread: main	MUTEX.CPP	41	04_mutex

Figure 5: Abstraction Plugin and Error Trailer in Eclipse.

Figure 4 compares the performance for some simple C++ benchmarks with the one of the C/C++ program checker StEAM (Mehler 2006), which systematically analyzes a program as an executable in object code and complies with in- and output. In the *BubbleSort* and *8-Puzzle*, the program checker is faster, while in planning only data abstraction solves the problem. In the communication protocol examples, the analysis via PDDL is superior.

We do not claimed to have a full translation of C/C++ to PDDL. For example given that current PDDL is inherently static (it does not allow dynamic object creation), there are obvious restrictions to the expressiveness of sources that can be processed (no dynamic mememory allocation, no incremental invocation/deletion of threads etc). Nonetheless, the results indicate that PDDL can yield exploration advances.

References

Bakera, M.; Edelkamp, S.; Kissmann, P.; and Renner, C. D. 2008. Solving solving μ-calculus parity games by symbolic planning. In *MoChArt*, 15–33.

Cimatti, A.; Giunchiglia, E.; Giunchiglia, F.; and Traverso, P. 1997. Planning via model checking: A decision procedure for AR. In *ECP*, 130–142.

Clarke, E. M.; Grumberg, O.; and Peled, D. A. 1999. *Model checking*. MIT Press.

Clarke, E. M.; Kroening, D.; and Lerda, F. 2004. A tool for checking ANSI-C programs. In *TACAS*, 168–176.

de Brugh, N. H. M. A.; Nguyen, V. Y.; and Ruys, T. C. 2009. Moonwalker: Verification of .net programs. In *TACAS*, 170–173.

Edelkamp, S., and Jabbar, S. 2006. Action planning for directed model checking of Petri nets. *Electronic Notes in Theoretical Computer Science (ENTCS)* 149(2):3–18.

Edelkamp, S.; Schuppan, V.; Bosnacki, D.; Wijs, A.; Fehnker, A.; and Aljazzar, H. 2008. Survey on directed model checking. In *MoChArt*, 65–89.

Edelkamp, S.; Jabbar, S.; and Lluch-Lafuente, A. 2005. Action planning for graph transition systems. In *ICAPS'05-Workshop on Verification and Validation of Model-Based Planning and Scheduling Systems*.

Edelkamp, S. 2003. Promela planning. In *SPIN*, 197–212.

Fox, M., and Long, D. 2003. PDDL2.1: An extension of pddl for expressing temporal planning domains. *JAIR* 20:61–124.

Helmert, M., and Domshlak, C. 2009. Landmarks, critical paths and abstractions: What's the difference anyway? In *ICAPS*.

Hoffmann, J. 2003. The Metric-FF planning system: Translating *ignoring delete lists* to numeric state variables. *JAIR* 20:291–341.

Leven, P.; Mehler, T.; and Edelkamp, S. 2004. Directed error detection in c++ with the assembly-level model checker steam. In *SPIN*, 39–56.

Mehler, T. 2006. *Challenges and Applications of Assembly-Level Software Model Checking*. Ph.D. Dissertation, University of Dortmund.

Merino, P.; del Mar Gallardo, M.; Martinez, J.; and Pimentel, E. 2002. αSPIN: Extending SPIN with abstraction. In *SPIN*, 254–258.

Stroustrup, B. 1994. *The C++ Programming Language – 2nd ed.* Addison-Wesley Publishing Company.

Visser, W.; Havelund, K.; Brat, G.; and Park, S. 2000. Java pathfinder - second generation of a java model checker.

Visser, W.; Havelund, K.; Brat, G.; Park, S.; and Lerda, F. 2003. Model checking programs. *Automated Software Engineering Journal* 10(2):203–232.

Wehrle, M., and Helmert, M. 2009. The causal graph revisited for directed model checking. In *SAS*, 86–101.