Smoke Test Planning using Answer Set Programming

Tobias Philipp[1], Valentin Roland[1], Lukas Schweizer[2] *

[1] SINA Development and Verification Team, Division Defence & Space, secunet Security Networks AG, Essen (Germany)
[2] Computational Logic Group, Technische Universität Dresden, Dresden (Germany)

Received 10 November 2020 | Accepted 23 November 2020 | Published 10 February 2021

Abstract

Smoke testing is an important method to increase stability and reliability of hardware-dependent systems. Due to concurrent access to the same physical resource and the impracticality of the use of virtualization, smoke testing requires some form of planning. In this paper, we propose to decompose test cases in terms of atomic actions consisting of preconditions and effects. We present a solution based on answer set programming with multi-shot solving that automatically generates short parallel test plans. Experiments suggest that the approach is feasible for non-inherently sequential test cases and scales up to thousands of test cases.

Keywords

Planning, Answer Set Programming, Testing.

DOI: 10.9781/ijimai.2021.02.003

I. Introduction

Quick development-test-cycles are vital for the development of functional and reliable software. Thus, it is considered best-practice to frequently run a subset of regression tests, called smoke tests, to catch as many issues as possible as early as possible [1]. In contrast to regression testing, smoke tests lean towards minimizing the time spent on a test run, sacrificing coverage if necessary. To reduce overall run time, software tests are often run concurrently on isolated instances of the *system under test* (SUT). This works well if the software system can run on virtualized infrastructure, which can be easily scaled.

However, some systems depend on real hardware and thus virtualization is impossible. Furthermore, functional and performance characteristics of a virtualized system may significantly differ from operations on bare metal, which would render performance and load tests meaningless.

Smoke testing of hardware or hardware-dependent systems requires some form of planning: A running test inevitably changes the state of the SUT and thus the hardware state. Therefore, isolation of test cases is difficult, since multiple tests may depend on the same physical resources. Consequently, tests can neither be easily parallelized nor chained and some execution strategy is required.

A straightforward solution is to bring the SUT to a clean state after each test has run individually, e.g. by a cold reboot. For a number of reasons, this is not satisfactory: Resetting the SUT to a known state is often difficult or time-consuming. Issues, such as hidden assumptions on the concurrent use of different SUT features, may not occur with a strictly sequential execution.

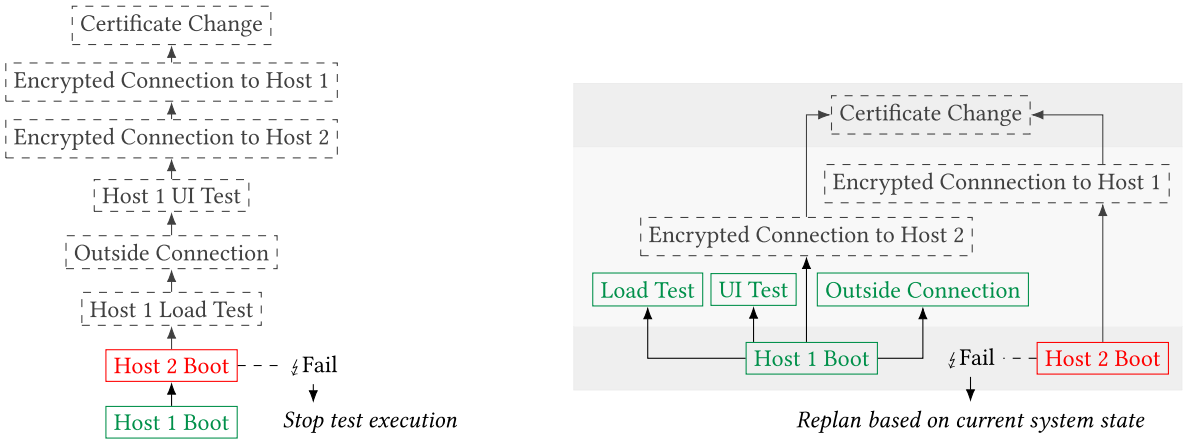
* Corresponding author.
E-mail addresses: tobias.philipp@secunet.com (T. Philipp), science@vroland.de (V. Roland), lukas.schweizer@tu-dresden.de (L. Schweizer).

In this paper we propose to decompose test cases into atomic actions with preconditions and effects such that we can automatically infer a suitable test plan that is short and parallel. Each action is a small, specific instruction performed on the SUT, like “boot system X” or “connect to host Y”. Preconditions describe the state of the SUT in which an action is executable. Effects describe the changes of the state of the SUT after successful execution of the action. Furthermore, actions are associated with instructions determining their failure or success.

In our proposed method, this information is given declaratively to enable the automated generation of short parallel test plans. We demonstrate that solving the planning problem can be done using *Answer Set Programming* (ASP) and show that the system can solve planning problems with thousands of actions on non inherently-sequential problems. The main benefits of this approach include:

1. *Reduced overall execution times*: Consider Fig. 1 that compares test executions with automated planning to a predefined sequential test plan in which a secure network connection is established, user interface tests are performed and a certificate is exchanged. Note that the parallel plan consists of four time points whereas the sequential needs eight.
2. *Maximal test execution*: In case of failing tests, as many tests as possible can be executed by replanning: Assuming that the action “Host 2 Boot” fails, the predefined sequential test plan fails early, whereas one can construct a new test plan that allows the continued execution of tests that do not depend on host 2. Consequently, the state of the SUT is preserved, no reboots are necessary and we do not need to (re-)execute passed tests.
3. *Specification reuse*: Declarative definition of actions and a flexible planning framework allows reusing actions in regression tests, integration tests, fixed scenarios or component-wise testing trough different planning goals and constraints.

Due to the nature of testing we deal with uncertainty in the outcome of actions as well as in the SUT state itself, which captures the ideas of conformant planning [2]. Moreover, maximal test execution can



(a) Sequential execution.

(b) Parallel test execution with replanning.

Fig. 1. Comparison of test execution with sequential and parallel plans, in the scenario of the action "Host Boot 2" failing.

be seen as some form of oversubscription planning [3], which aims to achieve as many sub-goals as possible instead of a conjunctively defined (single) top-goal.

The paper is structured as follows: We briefly describe ASP in Section II, before a formal description of smoke test planning is then given in Section III. The ASP encoding and its correspondence to tests is then presented in Section IV. In Section V, we present an experimental evaluation on the basis of a family of benchmarks. Finally, we describe related work and conclude.

II. Preliminaries

ASP is a declarative modeling and problem solving framework that combines techniques of knowledge representation and database theory. Two of the main advantages of ASP are its expressiveness [4] and, when using non-ground programs, its advanced declarative problem modeling capability. Such programs must be transformed to a ground program prior to solving.

We review the basic notions of answer set programming [5] under the stable model semantics [6], and refer to introductory literature [4], [7], for a more comprehensive introduction.

We fix a countable set $\mathcal{U}$ of (domain) elements, also called constants; and suppose a total order $<$ over the domain elements. An atom is an expression $p(t_1, ..., t_n)$, where p is a predicate of arity $n \geq 0$ and each ti is either a variable or an element from $\mathcal{U}$. An atom is ground if it is free of variables. $B_{\mathcal{U}}$ denotes the set of all ground atoms over $\mathcal{U}$. A (normal) rule ρ is of the form

$$a \leftarrow b_1, \ldots, b_k, \ not\ b_{k+1}, \ldots, \ not\ b_m$$

with $m \geq k \geq 0$, where a, b_1, ..., b_m are atoms, and "not " denotes *default negation*. The head of ρ is the singleton set $H(\rho) = \{a\}$ and the body of ρ is $B(\rho) = \{b_1, ..., b_k, \text{not } b_{k+1}, ..., \text{not } b_m\}$. Furthermore, $B^+(\rho) = \{b_1, ..., b_k\}$ and $B^-(\rho) = \{b_{k+1}, ..., b_m\}$. A rule ρ is safe if each variable in ρ occurs in $B^+(r)$. A rule ρ is ground if no variable occurs in ρ. A fact is a ground rule with empty body. An (input) database is a set of facts. A (normal) *program* is a finite set of normal rules. For a program Π and an input database D, we often write Π(D) instead of $D \cup \Pi$. For any program Π, let U_Π be the set of all constants appearing in Π. $Gr(\Pi)$ is the set of rules $\rho\sigma$ obtained by applying, to each rule $\rho \in \Pi$, all possible substitutions σ from the variables in ρ to elements of U_Π.

An interpretation $I \subseteq B_{\mathcal{U}}$ *satisfies* a ground rule ρ iff $H(\rho) \cap I \neq \emptyset$ whenever $B^+(\rho) \subseteq I$, $B^-(\rho) \cap I \neq \emptyset$. I satisfies a ground program Π, if each $\rho \in \Pi$ is satisfied by I. A non-ground rule ρ (resp., a program Π) is satisfied by an interpretation I if I satisfies all groundings of ρ (resp., $Gr(\Pi)$). $I \subseteq B_{\mathcal{U}}$ is an *answer set* (also called *stable model*) of Π if it is the subset-minimal set satisfying the *Gelfond-Lifschitz reduct* $\Pi^I = \{H(\rho) \leftarrow B^+(\rho) \mid I \cap B^-(\rho) = \emptyset, \rho \in Gr(\Pi)\}$ [6]. For a program Π, we denote the set of its answer sets by $\mathcal{AS}(\Pi)$.

We make use of further syntactic extensions, namely integrity constraints and choice rules, which both can be recast to ordinary normal rules as described in [8]. An *integrity constraint* is a rule ρ where $H(\rho) = \emptyset$, intuitively representing an undesirable situation; i.e. it has to be avoided that $B(\rho)$ evaluates positively. Choice rules are of the form $l\ \{a : a_1, ..., a_i\}\ u$, where a is an atom and $a_j = p_j$ or $a_j = not\ p_j$, for a_j an atom, $1 \leq j \leq i$, l and u are non-negative integers, and the expression $\{a : a_1, ..., a_n\}$ denotes the set of all ground instantiations of a, governed through $\{a_1, ..., a_n\}$. Intuitively, an interpretation satisfies a choice rule, if $l \leq N \leq u$ holds, where N is the cardinality of any subset of $\{a : a_1, ..., a_i\}$. In our encodings, we further use $-a$, to denote the classical negation of an atom a, though '–' is not an operator present in the introduced language and merely represents syntactic sugar[1].

Programs in this paper are given in the input language of Clingo [9], a state-of-the-art system combining an ASP grounder and solver. However, rule head and body are separated with ← instead of :- for improved readability.

III. Smoke Tests as Planning Problems with Unknowns

In the following, we describe the smoke test planning and optimization problem in terms of a planning problem in which the state of SUT properties may be unknown and actions can be executed in parallel. As executing many actions in a short time span is crucial, we consider *parallel plans*. Here, we consider a new semantics based on the ∀*-step semantics* [10], in which actions may be executed simultaneously, as long as they can be executed in any sequential order resulting in a unique state.

A. The System State as Fluents

We describe the state of the SUT in terms of *fluents* that correspond to atomic properties of the system. The set of fluents $\mathcal{F}$ is a designated subset of the set of terms $\mathcal{T}$. For instance, the fluent system_up(host1) states that the machine host1 is running. In a *system state* S, a fluent f can hold, not hold or it may be unknown, whether f holds or not. This is indicated by unary operators +, - and unknown for *holds*, *does not hold* and *unknown*, respectively. If f is a fluent, then + f, - f and

[1] Note that $-a$ merely represents a special predicate name, and together with an integrity constraint $\leftarrow -a, a$, the behavior of classical negation can be emulated.

unknown f are fluent literals. For instance, the fluent literal -system_up(host1) denotes that host1 is not running. The set of all *fluent literals* is referred to as $\mathcal{FL}$. Formally, we represent the *state* S_t of the *SUT at time t* as a finite set of fluent literals. Initially, the presence of all fluents is unknown, thus $S_0 = \{\text{unknown } f \mid f \in \mathcal{F}\}$.

The function fluents: $\mathcal{P}(\mathcal{FL}) \rightarrow \mathcal{P}(\mathcal{F})$ maps a set of fluent literals to the set of fluents occurring in that set. The function lits : $\mathcal{P}(\mathcal{F}) \rightarrow \mathcal{P}(\mathcal{FL})$ maps a set of fluents F to the corresponding set of possible fluent literals $\{+f, -f, \text{unknown } f \mid f \in F\}$.

B. Test Cases as Actions

Smoke test cases are usually specified at varying levels of complexity and abstraction. A test describing a user interface workflow is semantically different from testing a single button press. To reason about SUT behavior for test planning, we need a unified, fine grained view on how the system state is changed. Thus, we decompose test cases into *actions*. Such actions represent small, self-contained operations. In the context of the test planning problem, we consider only the information relevant for planning.

Definition 1 (Action). An *action* a is a tuple (P, E, M), where

- $P \subseteq \mathcal{FL}$ is a finite set of *preconditions*, which do not contain fluent literals of the form unknown f,
- $E \subseteq \mathcal{FL}$ is a finite set of *effects*, and
- $M \subseteq \mathcal{F}$ is a finite set of *modifications*, which is the set of fluents an action may change if the action *fails*.

The set of all actions is denoted by $\mathcal{A}$. An action (P, E, M) is executable in state S if $P \subseteq S$, i.e. its preconditions are satisfied.

Definition 2 (Successor State). The *successor state* S_{i+1} of S_i after executing a finite set of actions A, denoted by $S_i \xrightarrow{A} S_{i+1}$ is defined as follows:

$$S_{i+1} = \left(S_i \setminus \text{lits}\left(\bigcup_{a \in A} \text{fluents}(E_a)\right)\right) \cup \bigcup_{a \in A} E_a$$

C. Plan Semantics

For all permutations $(a_1, ..., a_n)$ of a set of actions A_t, their sequential application $S_i \xrightarrow{a_1} S_{i+1} \xrightarrow{a_2} ... \xrightarrow{a_n} S_G$ must always result in a unique state S_G.

Definition 3 (Plan). A plan P for a set of actions $\mathcal{A}$ is a finite sequence of slots $P = \langle A_0, ..., A_n \rangle$.

Intuitively, all actions must be executable when applied, are mutually non-interfering, and each action a occurs at least once in a slot A_t for some $t \in \{0 \ldots n\}$.

Definition 4 (Non-interference). Let $a_1 = (P_1, E_1, M_1)$ and $a_2 = (P_2, E_2, M_2)$ be two actions. Then, a_1 and a_2 are *non-interfering*, if the following mutually holds:

1. $\text{fluents}(E_1) \cap \text{fluents}(E_2) = \varnothing$,
2. $\{+f \mid f \in P_1\} \cap \{-f, \text{unknown } f \in E_2\} = \varnothing$,
3. $\{-f \mid f \in P_1\} \cap \{+f, \text{unknown } f \in E_2\} = \varnothing$, and
4. $(\text{fluents}(P_1) \cup \text{fluents}(E_1) \cup M_1) \cap M_2 = \varnothing$.

Intuitively, the conditions describe the following: Condition 1. excludes concurrent modification of system properties. From the point of view of classical planning, this constraint seems too restrictive as it disallows executing actions in parallel although any order of execution results in the same final state [10]. However, in the context of smoke testing, actions represent state transitions of physical systems and fluents describe its properties. Therefore, a change of the presence of a fluent indicates concurrent access to physical resources that may be unsafe. Conditions 2. and 3 ensure that no action affects the precondition of other, concurrently executed actions. This assumes that a fluent literal which is present in the precondition and effects of an action always holds during the execution of an action. If this is not the case, the fluent must also appear in the set of *modifications* of that action. Condition 4 guarantees that two actions running in parallel do not modify fluents which are modified by another action. Thus, their parallel execution is independent of each other, in particular in case of failing actions. These conditions guarantee the prevention *false positive* or *false negative* test results.

Note that by defining non-interference and *executability* in this fashion, we model uncertainty in the SUT state as the third fluent state unknown. This is different to approaches like conformant planning [2], [11], where uncertainty in the initial state is modeled as a set of possible states. While one could construct such a *belief-state* for a state S, the initial state is not the only source of uncertainty. Even when successful, actions can introduce uncertainty through their effects: For instance, consider an action which re-starts the SUT. While some properties of the SUT will be known, some components may be in an uninitialized state, thus the related fluents are made unknown in the action's effects.

D. Planning Goal

In contrast to classical planning problems in which a desired goal state is to be reached, the goal in this paper is to find a shortest parallel plan that executes all actions. We call such a plan *complete*. In the case that there is no such plan as many actions as possible should be executed, which can be seen as some form of oversubscription planning [3]. While the system state determines which actions can be executed at any point in time, there is *no specified goal state*. The planning goal is a property of the plan itself, not of the system under test.

Example 1. Consider an arrangement of two devices, where a subset of the activities shown in Fig. 1 are tested: 1. The devices connect to each via encrypted network connections, 2. each device connects to the outside network, and 3. the devices stay connected if the encryption parameters of their connection are changed. Consider the following set of fluents that describe the relevant system properties $\mathcal{F}$ and the set of actions $\mathcal{A}$ as described in Table I:

$\mathcal{F}$ = {system_up(host1), system_up(host2)}
∪ toutside_up(host1), outside_up(host2)u
∪ tenc_conn_up(host1), enc_conn_up(host2)u
∪ tconn_to(host1, host2), conn_to(host2, host1)u

We can obtain the plan $P_e = \langle A_0, ..., A_3 \rangle$ with the following considerations:

- In state $S_0 = \{\text{unknown } f \mid f \in \mathcal{F}\}$, we can see that only boot(host1) and boot(host2) are *executable* in A_0. Since precondition and modifications are empty, 2, 3 and 4 are satisfied. As both actions each only affect one of the hosts, respectively, they are also *non-interfering*. Thus, we can schedule both actions in A_0 = {boot(host1), boot(host2)}
- With $S_0 \xrightarrow{A_0} S_1 \supseteq$ {+system_up(host1), unknown enc_conn_up(host1)}, the actions connect_en-crypted(host1), connect_encrypted(host2), connect_out-side(host1) and connect_outside(host2) get executable. Through checking for non-interference, we can see that they can all be scheduled for A_1.
- In S_2, change_conn(host1) as well as change_conn(host2) are executable. However, they mutually interfere due to conditions 1 and 4. Thus, we need to schedule them sequentially, for instance as A_2 = {change_conn(host1)} and A_3 = {change_conn(host2)}.
- The resulting plan $P_e = \langle A_0, ..., A_3 \rangle$ is a shortest plan.

TABLE I. Representation of the Example in Terms of Actions. The Variables *H1*, *H2* ϵ {host1, Host2} Are Placeholders for Any Concrete Host, Where *H1* and *H2* Are Distinct

Action	Precondition *P*	Effects *E*	Modifications *M*
boot(H1) – Boot up *H1*.	∅	+ system_up(H1) unknown enc_conn_up(H1) unknown outside_up(H1) unknown conn_to(H1, H2)	∅
connect_outside(H1) – H1 initiates an (unencrypted) connection to an outside network.	+ system_up(H1)	+ outside_up(H1)	∅
connect_encrypted(H1) – *H1* initiates an encrypted connection to the other host *H2*.	+ system_up(H1)	+ conn_to(H1, H2) + enc_conn_up(H1)	∅
change_conn(H1) – *H1* initiates a change of encryption parameters, e.g. renews a certificate.	+ system_up(H1) + enc_conn_up(H1) + conn_to(H1, H2) + conn_to(H2, H1)	+ conn_to(H1, H2) + conn_to(H2, H1)	enc_conn_up(H1) enc_conn_up(H2) conn_to(H1, H2) conn_to(H2, H1)

E. Replanning

Consider a plan $\langle A_0, ..., A_{t-1}, A_t, A_{t+1}, ..., A_\ell \rangle$, the corresponding states $S_0, ..., S_\ell$ and suppose action $e = (P, E, M)$ in A_t fails. In the case that $E = \varnothing$ and $M = \varnothing$, the SUT state is consistent with the plan. Otherwise, actions in the slots A_{t+1}, ..., A_ℓ could be non-executable and therefore, a new plan involving as many actions as possible that have not been executed is generated.

Replanning follows the same rules outlined in Section C, but with a different initial state. Intuitively, the new initial state is constructed by applying the successful actions as scheduled in previous plan, but making any fluent the failed action may have changed unknown. Formally, let $F_{\text{affected}} = \text{fluents}(E_e) \cup M_e$. Then, the new initial state is derived from the system state S_{t-1} before the action failed as follows:

$$S = \left(S_{t-1} \setminus \text{lits} \left(\bigcup_{a \in A_t} \text{fluents}(E_a) \cup F_{\text{affected}} \right) \right) \cup \{\text{unknown } f \mid f \in F_{\text{affected}}\} \cup \bigcup_{a \in A_t} E_a \setminus \text{lits}(F_{\text{affected}})$$

F. Finding Maximal Runnable Subsets

A complete plan may not exist, in particular in case of replanning as a failed test is excluded. For instance, an action that cannot be executed because all sequences of actions making its precondition true contain some action which has already failed. However, with the overall goal to quickly execute as many actions as possible, a *partial* plan can be constructed instead, which executes *as many outstanding actions as possible*. We can exploit the flexibility of a declarative approach to obtain such a maximal partial plan, by maximizing a score function instead of solving for satisfiability of a set goal. As a score function for a plan P of length n, we use the number of distinct outstanding actions $g(P)$ in P.

$$g(P) = \left| \bigcup_{t=1}^{n} \{a | a \in A_t, a \text{ not previously executed}\} \right|$$

IV. An ASP Program Using Multi-Shot Solving

For solving the smoke test planning problem, we phrase the planning and replanning problem in terms of an Answer Set Program and use Clingo [9] for solving, a state-of-the-art system combining an ASP grounder and solver supporting incremental solving [12] and assumptions. Incremental grounding allows to extend the logic program after initial grounding, by adding parameterized *subprograms*. Assumptions are realized with *external atoms* that allow to change their truth value after grounding. Together, these mechanisms allow to reuse a single ground program over the entire testing process, supplying initial conditions and goal condition via external atoms. We use the atoms as presented in Table II to encode the planning program which is divided into four *subprograms*: the instance program, the base program, the transition program, and the goal program.

TABLE II. ASP Predicates and Their Meaning: I Denotes May Atoms That May Be Added Incrementally, E Denotes May External Atoms

Atom	I	E	Description
apply(A,t)	✓		applies action A at time t
demands(A,F,true)			action A requires + F
demands(A,F,false)			action A requires - F
adds(A,F)			action A has positive effect + F
deletes(A,F)			action A has negative effect - F
invalidates(A,F)			action A makes F unknown F
modifies(A,F)			action A may modify fluent F
pc_changes(A, F)			action A changes fluent F
interfere(A1, A2)			actions A1 and A2 mutually interfere
available(A)			action A can be used for in a plan
has_failed(A)		✓	the execution of A has failed
need_to_plan(A)		✓	action A must occur in the plan
add(F, t)	✓		fluent F is made true at time t
del(F, t)	✓		fluent F is made false at time t
inv(F, t)	✓		fluent F is made unknown at time t
holds(F, 0)	✓	✓	fluent F holds initially
holds(F, t)	✓		$t > 0$, fluent F holds at time t
was_applied(A, t)	✓		action A has been applied at least once at t
goal_horizon(t)	✓	✓	the current horizon is t

1. Instances

As the basis of the planning program, we represent actions and the initial state as ASP facts. This part of the program is specific to the supplied set of actions and, in case of replanning, the current SUT state.

An action a is represented as an action(a) atom. Its precondition, effects and modifications are expressed using the atoms demands(..), adds(..), deletes(..), invalidates(..), modifies(..), respectively. If the initial state S_0 is different from {unknown $f \mid f \in \mathcal{F}$}, it is specified through holds(f, 0) and -holds(f, 0) atoms. Since variables are capitalized in the input language of Clingo, variables A and F refer to singular actions and fluents in this context. They are distinct from A and F, which refer to sets thereof.

Note that we assume that a nop action is contained in the specification.

2. Base Program

The *base program* specifies available actions, required actions and pairs of interfering actions. Mutual interference of actions is calculated by first collecting the fluents modified by its postcondition.

```
pc_changes(A, F) ← adds(A, F).
pc_changes(A, F) ← deletes(A, F).
pc_changes(A, F) ← invalidates(A, F).
```

Then, interference is calculated as described in Section C.

```
interfere(A1, A2) ← demands(A1, F, _), pc_changes(A2, F).
interfere(A1, A2) ← pc_changes(A1, F), pc_changes(A2, F).
interfere(A1, A2) ← modifies(A1, F), modifies(A2, F).
interfere(A1, A2) ← demands(A1, F, _), modifies(A2, F).
interfere(A1, A2) ← pc_changes(A1, F), modifies(A2, F).
interfere(A1, A2) ← interfere(A2, A1).
```

An action is available if it did not fail in a previous plan:

```
available(A) ← action(A), not has_failed(A).
```

In case of *replanning*, actions which have already been executed successfully are not required in the new plan. Thus, they are excluded from the set of required actions. This is implemented through the external need_to_plan(A) atoms, whose truth values can be changed for replanning.

3. Transition Program

For a given slot $A_{t-1}1 \in P$ of a plan P, the *transition program* ensures that only a non-empty set of *executable*, *mutually non-interfering* actions is selected as A_t. It constitutes the ASP rules necessary to raise the planning horizon (and thus the maximal plan length) from $t-1$ to t and describes the new state S_t. The transition program is incrementally added.

1. At least one action is selected for execution in this slot, marked by apply(A, t). This may be the nop action, which is always executable.

   ```
   1 { apply(A, t) : available(A) }.
   ```

2. Each applied action is executable w.r.t. the previous state S_{t-1}.

   ```
   ← apply(A, t), demands(A, F, true), not holds(F, t−1).
   ← apply(A, t), demands(A, F, false), not −holds(F, t−1).
   ← apply(A, t), not available(A).
   ```

3. Applied actions are pairwise non-interfering.

   ```
   ← interfere(A1, A2), apply(A1, t), apply(A2, t), A1 < A2.
   ```

4. Describe the new state S_t based on the applied actions.

   ```
   add(F, t) ← apply(A, t), adds(A, F).
   del(F, t) ← apply(A, t), deletes(A, F).
   inv(F, t) ← apply(A, t), invalidates(A, F).

   not holds(F, t) ← inv(F, t).
   not −holds(F, t) ← inv(F, t).

   holds(F, t) ← add(F, t).
   holds(F, t) ← holds(F, t−1), not del(F, t), not inv(F, t).
   − holds(F, t) ← del(F, t).
   − holds(F, t) ← −holds(F, t−1), not add(F, t), not inv(F, t).
   ```

5. Track which actions have been applied so far with was_applied(A, t)-atoms.

   ```
   was_applied(A, t) ← apply(A, t).
   was_applied(A, t) ← was_applied(A, t−1).
   ```

4. Goal Program

The *goal program* characterizes the desired plan for a specific horizon h. A straightforward goal is to have all actions (except nop) planned in some slot A_t with $t \leq h$. Through external goal_horizon(h) atoms, goals are constructed in a way that they can later be *deactivated*, i.e. have no effect on the set of models of the ground program.

```
← not was_applied(A, h), need_to_plan(A), goal_horizon(h).
```

While this is sufficient in a narrow sense, practically, additional rules such as optimization goals, optional actions, timeout bounds or other planning constraints may be added.

5. Externals

To use the same ground program for planning and replanning, some input must be supplied as external atoms:

1. *Initial Conditions*: The state of a fluent is expressed by the combination of holds/2 and -holds/2: holds(F, T) states that fluent F holds at slot T, -holds(F, T) states that fluent F does not holds at slot T, and if neither holds(F, T) not -holds(F, T) holds, the fluent F is unknown at slot T. The initial conditions describe which fluents hold *before* the first slot A_0 and are marked as external.

   ```
   #external holds(F, 0) : fluent(F).
   #external −holds(F, 0) : fluent(F).
   ```

2. *Required Actions*: When replanning, actions which have already been executed do not need to be re-run. There might be other practical reasons for not requiring some actions, such as further reducing the overall execution time for testing only selected features.

   ```
   #external need_to_plan(A) : action(A).
   ```

3. *Failed Actions*: Actions for which execution has failed have to be excluded from the set of available actions when replanning.

   ```
   #external has_failed(A) : action(A).
   ```

4. *Goals*: During solving, the search horizon is increased incrementally. As the ground program is reused for solving with different search horizons, a new goal is added. The old goal is deactivated by assigning goal_horizon(t) to false.

   ```
   #subprogram goal(t).
   #external goal_horizon(t).
   ← some_constraint(...), goal_horizon(t).
   ```

A. Solving Heuristics

As we do not know the required planning horizon, we guess a planning horizon before solving. Selecting a horizon which is too low results in the program being unsatisfiable. In contrast, an unnecessarily large horizon yields a large ground program, taking more time to preprocess and solve. Thus, the horizon should approach the plan length as quickly as possible, without overestimating it.

To balance these two aspects, we solve the planning problem in two phases: First, the search horizon is increased in exponential steps. In each step, additional *step*-subprograms are grounded. The old goal is deactivated and a new *goal*-subprogram for the new horizon h_n is added. This is repeated until the resulting ground program is *satisfiable*. Then, the minimal plan length is in the range $(h_{last}, h_n]$.

In the second phase, we proceed to find the lowest satisfiable horizon using binary search. Note that while the goal program is adjusted for every horizon, no additional *step*-subprograms are added, as they have already been ground in the first phase. This procedure is reminiscent to a parallel plan search algorithm proposed by Rintanen [10], however, we only solve one horizon at a time.

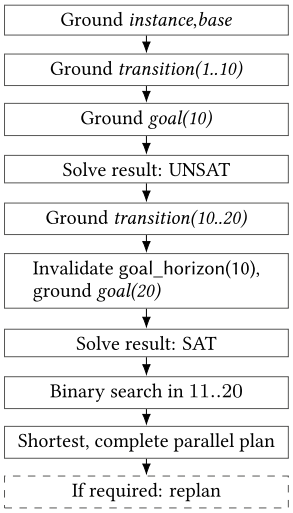


Fig. 2. Example of an incremental solving run.

An example of such an incremental solving run is presented in Fig. 2: After grounding the *instance* and *base* subprograms, the initial search horizon of 10 is increased until the instance is satisfiable at horizon 20. Then, the shortest plan is found through subsequent binary search.

A further improvement is to adapt the heuristics as follows:

#heuristic apply(A, t) : action(A). [−1, sign]

The addition of this line does not significantly reduce the runtime of Clingo, but reduces the number of executed actions in a plan.

V. Experimental Results

In the following, we report on experimental results based on a generated family of four benchmarks[2] that have been measured in a virtualized GNU/Linux environment (Debian stretch) with four 2.2GHz CPU cores (Intel Xeon Silver 4114) and 16GB of main memory.

[2] ASP instances of some generated benchmarks are provided on https://github.com/vroland/smoke-testing-benchmarks.

Version 5.4.1 of the Clingo solver was used in parallel mode 4, split with Python 3.7.3.

The benchmark families, as described in Fig. 3, have a certain structure which we believe is realistic and include corner cases:

Tree-Structured: Every action a_i depends on a fluent that is made true by $a_{\lfloor \frac{i}{2} \rfloor}$. The resulting plans are highly parallel, with plan lengths of $\lceil \log_2 N \rceil$ for N actions.

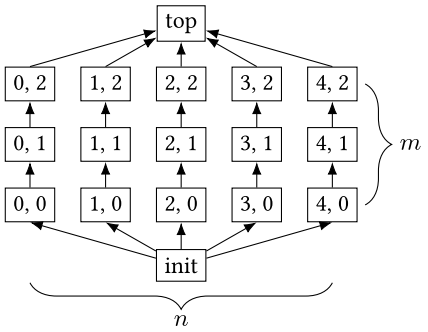
Sequential: Every action a_i depends on a fluent that is made true by a_{i-1}. The resulting plans are of length N for N actions.

Mixed: This benchmark aims to portray a more realistic test set, with tree-structured and sequential subsets of random length. Some of the branches of a tree-structured subset conflict based on their modifications. Every action a_i has a probability of $\frac{1}{5}$ producing a tree branch or a conflicting branch respectively, by depending on $a_{\lfloor \frac{i}{2} \rfloor}$ instead of a_{i-1}. With a probability of $\frac{3}{5}$, an action a_i depends on its direct predecessor a_{i-1}, producing a sequential sub-structure. In sequential subsets, actions invalidate their precondition. As the benchmark is randomized, repeated runs of this benchmark may not necessarily yield the same results.

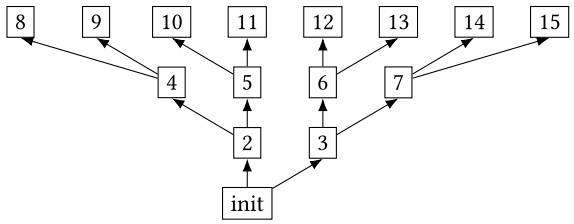
Parallel Chains: Another realistic usage profile consists of running sequential test procedures in parallel. The benchmark is constructed by generating n *Sequential* instances of length m, which depend on a common init action and are jointly required by a final top action.

Fig. 4 shows that the size of the ground program w.r.t. number of rules and atoms increases with plan length. For the *Tree-Structured* and *Mixed* benchmarks, the search horizon stays low. Consequently, only a low number of instances of the transition subprogram have to be added to make the planning problem satisfiable. In contrast, the program size of the *Sequential* benchmark increases quickly. We observe a significant increase in the number of rules and atoms in the case that we increase the search horizon in exponential steps according to the solving heuristic (see Section A).

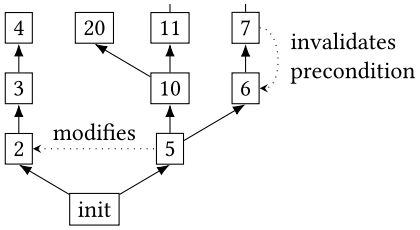
Fig. 5 presents the time spent in showing satisfiability and unsatisfiability and the total planning time for our benchmark families. Our upper bound on the horizon is 1000, consequently *Sequential* benchmarks are only conducted up to this size. Starting at



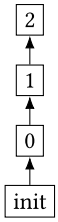
(a) parallel chains family



(b) tree-structured family



(c) mixed family



(d) sequential family

Fig. 3. Dependency structure of the considered benchmark families.

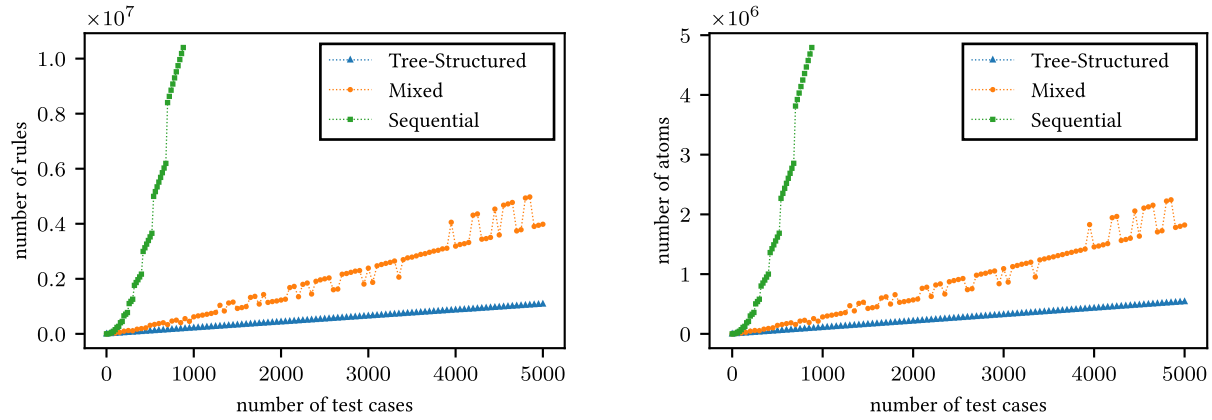


Fig. 4. Ground program size w.r.t. the number of actions.

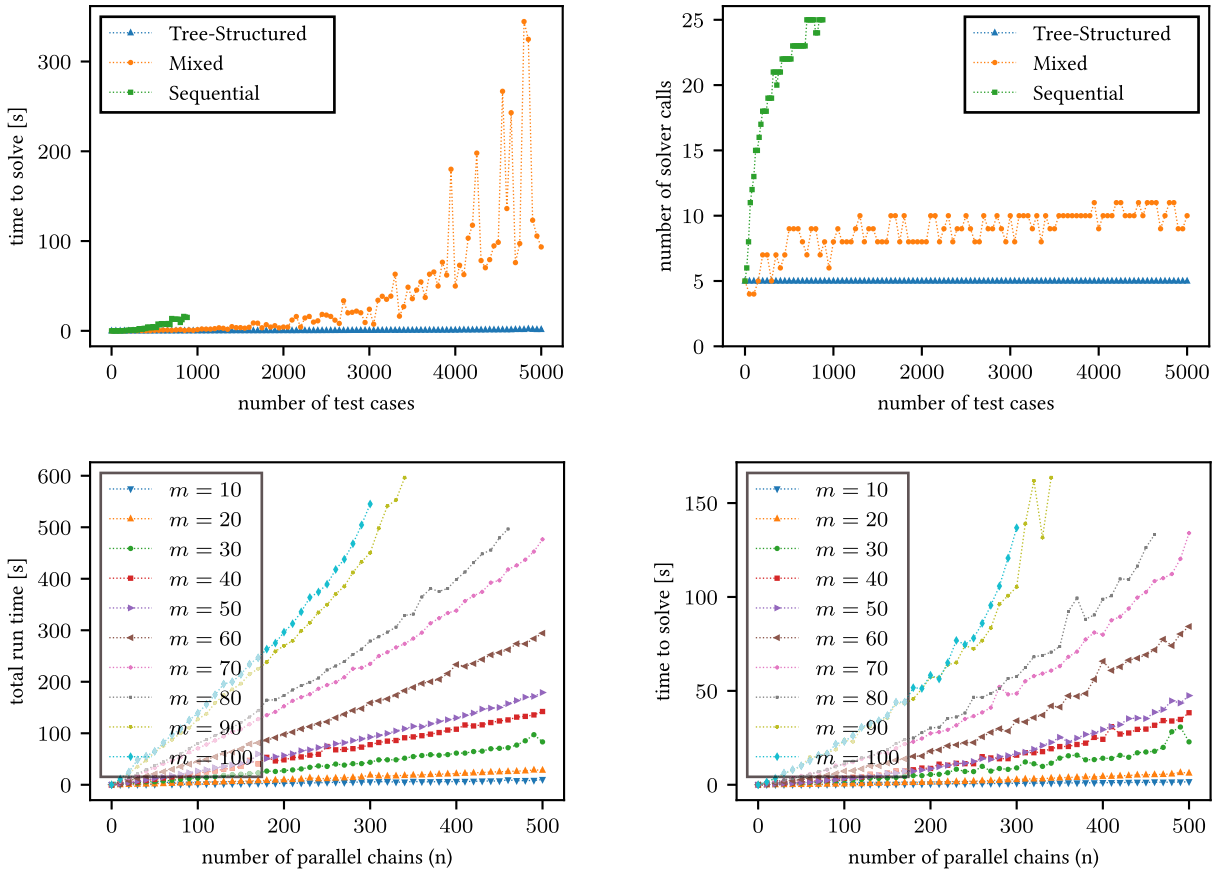


Fig. 5. Total run time and time to satisfiability for all benchmark families.

approximately 2000 actions, *Mixed* instances take significantly longer to solve than *Tree-Structured* instances. In contrast to the highly parallel *Tree-Structured* benchmark with short plan lengths, we observe an increase in the number of interactive solver calls with *Sequential* and *Mixed*, as the number of actions increases. This is a result of our search heuristic (see Section A), which incrementally approaches the larger search horizon needed for longer plans.

In case of the *Parallel Chains* benchmark, runtimes increase with both higher chain length m and more parallel chains n. However, as indicated by the previously discussed benchmark families, our solution is best suited for highly parallel instances: For the same total number of actions, instances with high n and small m require significantly less time to solve than longer, less parallel plans with high m.

Considering differences in run time for a constant n and increasing m, we notice a non-uniform distribution. This is a result of the incremental search heuristic: For instance, the difference in runtime of $m = 90$ to $m = 100$ is smaller than the difference of $m = 90$ to $m = 80$. This irregularity occurs because for increasing the horizon from 80 to 90, an additional grounding and solving step is necessary, whereas 90 and 100 fall in the same horizon step.

Presumably through memory limitations of our benchmark hardware resulting in swapping, jitter is introduced for larger problem sizes.

By comparing total runtime with time spent for solving, we can observe a difference in almost an order of magnitude. This suggests that a large portion of the total run time is spent in preprocessing and grounding.

Due to the reuse of a single ground program in the entire test execution, we avoid repeated preprocessing and keep the internal solver state. This speeds up subsequent replanning as shown in Fig. 6. The *Sequential* benchmark exhibits a steep growth of ground program size with the number of actions, as the latter is equal to the minimal search horizon. In the first phase of the horizon search, solver calls spend large parts of the execution time in preprocessing and grounding rather than solving. The peak at call 22 marks the finishing goal of the initial plan search. From call 23, the search for a new plan starts after an initial action has failed. Though solving now takes longer due to the overall larger ground program size, the total planning time is reduced, as less preprocessing and grounding time is needed.

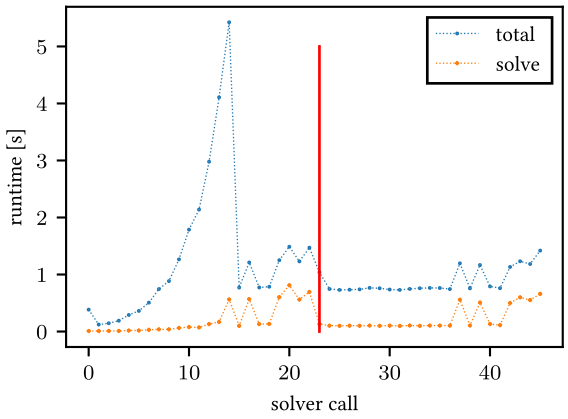


Fig. 6. Time spent per solver call for planning and replanning of a *sequential* instance with 500 actions.

VI. Conclusion

Smoke testing is an important method to increase the reliability of hardware-depending systems. Due to concurrent access to the same physical resource and the infeasibility of the use of virtualization, smoke testing requires some form of planning. In this paper, we propose to decompose test cases in terms of atomic actions consisting of preconditions and effects and developed a declarative framework that allows to automatically generate complete and incomplete plans based on ASP. By modeling tests as state transitions of a system-under-test, our method is capable of generating shortest, parallel test plans, while offering the flexibility to incorporate additional goals, constraints and knowledge [13].

Broader use cases can be modeled as variants of smoke testing, with some adjustments to the planning goals and constraints. For instance, *regression testing* can be seen as scaled-up smoke testing, which we have shown is viable for thousands of tests or more, especially if planning time is not critical.

Furthermore, functionality can easily be tested piecewise in isolation to inspect reasons for test failures. By only requiring the execution of a specific action as our goal and minimize the total number of planned actions, we obtain a minimal trace of actions to produce a particular SUT behavior.

Moreover, through the use of parallel plans, we can find issues which may not occur in sequential or isolated test runs. Such issues, typically referred to as *race conditions* or *race hazards*, arise from conflicting concurrent use of resources and lead to non-deterministic test outcomes. When such dependencies of two tests on the same resource are insufficiently specified but some property of the execution environment is implicitly assumed, we call this a *hidden assumption*. Finding *race conditions* and *hidden assumptions* can be time-consuming, but can be automated to some degree by our solution: Running a large number of structurally diverse plans [14] can provide insight on sequences of actions which cause the system to fail, by recording failing plans and looking for similarities.

Experiments show that generating *short* and *highly parallel* plans can be efficiently done using ASP. We believe that domains like networking tests or tests of distributed systems allow for such plans.

Inherently sequential test procedures prove adverse to our solution and may be better addressed by existing planning tools. These occur in domains where all actions operate on a common resource, forcing mutual exclusion, like testing workflows in a graphical user interface.

The AI planning problem is used in different contexts in the area of testing: [15] uses PDDL to find well-known security issues in web applications, [16] considers test case generation, [17] uses contingent planning in the area of penetration testing, [18] proposes to test chatbots using planning, [19] considers test case generation for systems, and [20] considers hierarchical GUI test case generation using planning methods. In the domain of software testing, to the best of our knowledge, test execution planning has not been modeled in the sense of an AI planning problem. However, work has been done on obtaining *test specification*, like Behavior-Driven Development (BDD) [21], [22] or Model-Driven Testing. At a first glance, actions may seem reminiscent of *given-when-then-style* scenarios in Behavior-Driven Development (BDD) [21], [22]. However, while BDD is concerned with how tests are specified in natural language, this work focuses on fast test execution. Moreover, while BDD scenarios may cover multiple layers of abstraction, actions should remain relatively low-level. Conceptually, a BDD test is more akin to an abstract view of a *subset of a plan* than to an action. However, synergies could emerge when using the BDD process in conjunction with planning-based test execution.

Solving planning problems based on the propositional satisfiability solvers instead of specialized planners has been explored since the 1990s [10], [23], exploiting the flexibility of general-purpose SAT solvers. Similarly, Answer Set Programming has successfully been used to implement classical [24] as well as real-world planning such as tasks robotics [25], [26]. Accord-ing to [27], ASP-based planners perform especially well for short plans with complex dependencies.

We compare our approach to modeling uncertainty as in conformant planning [2], [11] in Section C, and also point out the related idea of oversubscription planning [3] when it comes to achieving as many actions as possible. Though, in contrast to oversubscription approaches as e.g. in [3], we do not impose cost-estimates on actions such that a global constraint on the total cost is satisfied (e.g. for resources such as time, or power consumption), but merely achieve the maximum number of actions. However, such an extension could be achieved by extending actions with temporal information, such as an estimated duration, where time then can be encoded as a limited resource. For instance, temporal constraints such as a maximal run time or required parallelism could be added. Going further, the notion of a shortest plan could now be redefined as a temporal (cf. [28] for an overview). However, how this affects planning performance and which temporal extensions prove to be of use in practice, remains an open question.

In the future, we plan to investigate the test specification debugging problem, i.e. the questions, why a test specification does not admit a complete test plan and the possibilities to generate substantially different test plans.

Acknowledgment

We are grateful for the valuable feedback and additional references from the anonymous reviewers, which helped greatly to improve this work.

References

[1] G. D. Everett, R. McLeod Jr, *Software testing: testing across the entire software development life cycle*. Piscataway, NJ, Hoboken, N.J: John Wiley & Sons, 2007.

[2] H. Palacios, H. Geffner, "Compiling uncertainty away in conformant planning problems with bounded width," *Journal of Artificial Intelligence Research*, vol. 35, pp. 623–675, 2009, doi: 10.1613/jair.2708.

[3] D. E. Smith, "Choosing Objectives in Over-Subscription Planning," in *Proceedings of the Fourteenth International Conference on Automated Planning and Scheduling (ICAPS 2004)*, June 3-7 2004, Whistler, British Columbia, Canada, 2004, pp. 393–401, AAAI.

[4] G. Brewka, T. Eiter, M. Truszczynski, "Answer set program-ming at a glance," *Communications of the ACM*, vol. 54, no. 12, pp. 92–103, 2011, doi: 10.1145/2043174.2043195.

[5] I. Niemelä, "Logic programs with stable model semantics as a constraint programming paradigm," *Annals of Mathematics and Artificial Intelligence*, vol. 25, no. 3-4, pp. 241–273, 1999, doi: 10.1023/A:1018930122475.

[6] M. Gelfond, V. Lifschitz, "Classical negation in logic programs and disjunctive databases," *New Generation Computing*, vol. 9, no. 3/4, pp. 365–386, 1991, doi: 10.1007/BF03037169.

[7] T. Janhunen, I. Niemelä, "The answer set programming paradigm," *AI Magazine*, vol. 37, no. 3, pp. 13–24, 2016.

[8] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, "Answer Set Solving in Practice," *Synthesis Lectures on Artificial Intelligence and Machine Learning*, vol. 6, pp. 1–238, 2012, doi: 10.2200/S00457ED1V01Y201211AIM019.

[9] M. Gebser, R. Kaminski, B. Kaufmann, T. Schaub, "Clingo = ASP + control: Preliminary report," *CoRR*, vol. abs/1405.3694, 2014.

[10] J. Rintanen, K. Heljanko, I. Niemelä, "Planning as satisfia-bility: parallel plans and algorithms for plan search," *Artificial Intelligence*, vol. 170, no. 12-13, pp. 1031–1080, 2006, doi: 10.1016/j.artint.2006.08.002.

[11] A. Cimatti, M. Roveri, P. Bertoli, "Conformant planning via symbolic model checking and heuristic search," *Artificial Intelligence*, vol. 159, no. 1-2, pp. 127–206, 2004, doi: 10.1016/j.artint.2004.05.003.

[12] M. Gebser, R. Kaminski, B. Kaufmann, M. Ostrowski, T. Schaub, S. Thiele, "Engineering an incremental ASP solver," in *Logic Programming, 24th International Conference, ICLP 2008*, Udine, Italy, December 9-13 2008, Proceedings, vol. 5366 of Lecture Notes in Computer Science, 2008, pp. 190–205, Springer.

[13] T. C. Son, C. Baral, T. H. Nam, S. A. McIlraith, "Domain-dependent knowledge in answer set planning," *ACM Transactions on Computational Logic*, vol. 7, no. 4, pp. 613–657, 2006, doi: 10.1145/1183278.1183279.

[14] T. Eiter, E. Erdem, H. Erdogan, M. Fink, "Finding similar/-diverse solutions in answer set programming," *Theory and Practice of Logic Programming*, vol. 13, no. 3, pp. 303–359, 2013, doi: 10.1017/S1471068411000548.

[15] F. Wotawa, J. Bozic, "Plan it! automated security testing based on planning," in *Proceedings of Testing Software and Systems - International Conference, ICTSS 2014, Madrid, Spain, September 23-25, 2014., vol. 8763 of Lecture Notes in Computer Science*, 2014, pp. 48–62, Springer.

[16] A. E. Howe, A. von Mayrhauser, R. T. Mraz, "Test case generation as an AI planning problem," *Automated Software Engineering*, vol. 4, no. 1, pp. 77–106, 1997, doi: 10.1023/A:1008607721339.

[17] D. Shmaryahu, G. Shani, J. Hoffmann, M. Steinmetz, "Simulated penetration testing as contingent planning," in *Proceedings of the Twenty-Eighth International Conference on Automated Planning and Scheduling, ICAPS 2018, Delft, The Netherlands, June 24-29, 2018*, 2018, pp. 241–249, AAAI Press.

[18] J. Bozic, O. A. Tazl, F. Wotawa, "Chatbot testing using AI planning," in *Proceedings of International Conference On Artificial Intelligence Testing IEEE, AITest 2019, Newark, CA, USA, April 4-9*, 2019, 2019, pp. 37–44, IEEE.

[19] R. T. Mraz, A. E. Howe, A. von Mayrhauser, L. Li, "System testing with an AI planner," in *Proceedings of Sixth International Symposium on Software Reliability Engineering, ISSRE 1995, Toulouse, France, October 24-27, 1995*, 1995, pp. 96–105, IEEE Computer Society.

[20] A. M. Memon, M. E. Pollack, M. L. Soffa, "Hierarchical GUI test case generation using automated planning," *IEEE Trans-actions of Software Engineering*, vol. 27, no. 2, pp. 144–155, 2001, doi: 10.1109/32.908959.

[21] M. Diepenbeck, U. Kühne, M. Soeken, R. Drechsler, "Be-haviour driven development for tests and verification," in *Proceedings of Tests and Proofs - 8th International Conference, TAP@STAF 2014, York, UK, July 24-25, 2014, vol. 8570 of Lecture Notes in Computer Science*, 2014, pp. 61–77, Springer.

[22] SmartBear Software, "Cucumber," 2019. [Online]. Available: https://cucumber.io/, Accessed: 2020-01-11.

[23] H. A. Kautz, B. Selman, "Planning as satisfiability," in *10th European Conference on Artificial Intelligence, ECAI 92, Vienna, Austria, August 3-7, 1992. Proceedings*, 1992, pp. 359–363, John Wiley and Sons.

[24] M. Gebser, R. Kaufmann, T. Schaub, "Gearing up for effective ASP planning," in *Correct Reasoning - Essays on Logic-Based AI in Honour of Vladimir Lifschitz*, vol. 7265 of Lecture Notes in Computer Science, 2012, pp. 296–310, Springer.

[25] S. Zhang, M. Sridharan, F. S. Bao, "ASP+POMDP: integrating non-monotonic logic programming and probabilistic plan-ning on robots," in *Proceedings of IEEE International Confer-ence on Development and Learning and Epigenetic Robotics, ICDL-EPIROB 2012, San Diego, CA, USA, November 7-9, 2012*, 2012, pp. 1–7, IEEE.

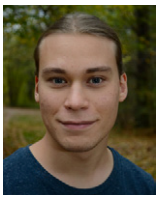
[26] J. J. Portillo, C. L. García-Mata, P. R. Márquez-Gutiérrez, R. Baray-Arana, "Robot platform motion planning using answer set programming," in *Proceedings of the Seventh Latin American Workshop on Non-Monotonic Reasoning, LANMR 2011, Toluca, Estado de México, México, November 7-8, 2011, vol. 804 of CEUR Workshop Proceedings*, 2011, pp. 35–44, CEUR-WS.org.

[27] Y. Jiang, S. Zhang, P. Khandelwal, P. Stone, "Task planning in robotics: an empirical comparison of PDDL- and asp-based systems," *Frontiers of Information Technology & Electronic Engineering*, vol. 20, no. 3, pp. 363–373, 2019, doi: 10.1631/FITEE.1800514.

[28] L. Vila, "A survey on temporal reasoning in artificial intelligence," *AI Communications*, vol. 7, no. 1, pp. 4–28, 1994, doi: 10.3233/AIC-1994-7102.

Tobias Philipp

Tobias Philipp is a full-time senior consul-tant and verification expert in the Development and Verification Team, Defence & Space Division, secunet Security Networks AG, Germany. His research interests include formal verification, logic-based Artificial Intelligence, planning, constraints, and knowledge representation and reason ing. Until 2017, he was a scientifc staff member of the International Center of Computational Logic (ICCL) at Technische Universität Dresden. In 2013, he graduated of the International Master's Program in Computational Logic, Technische Universität Dresden, Germany.



Valentin Roland

alentin Roland is a final year Diploma student at the Faculty of Computer Science, Technische Universität Dresden, Germany. Since 2018, he works as a student assistant in the Development and Verification Team, Defence & Space Division, secunet Security Networks AG. He is interested in formal methods and verification, knowledge representation and reasoning and declarative problem solving. This includes academic research as well as industrial applications thereof.

Lukas Schweizer

Lukas Schweizer is a research assistant in the Computational Logic Group at Technische Universität Dresden, Germany. He is also working as a research engineer at Deepreason.ai, a spin-off from the University of Oxford, developing the next generation of Datalog engines and knowledge graph systems. His interests are knowledge representation and reasoning in general, in particular rule based approaches and answer-set programming.