Microlog Abstract Execution and State Enumeration

Mario Wenzel[1]

Abstract: In this paper we describe a method of abstract execution for Microlog programs by attaching conditions to derived facts. This method is used to enumerate the possible state space of a Microlog program. The set of states is reduced by considering return values from the environment as abstract memory positions, possibly collapsing an untenable number of states to just a few.

Microlog is a deductive database language with a strong logic foundation based on Datalog with calls to external functions that may be used to control sensors and actors. Derived facts and results of function calls are fed back into the system, creating a thought-act-cycle that allows for programming of intelligent agents.

Keywords: Datalog; Logic Programming; Microlog; Finite State Machines; Termination.

1 Introduction

We usually prefer declarative logic programming (LP) over imperative programming because the LP languages have mathematically precise semantics based on logic, which makes programs easier to verify and programming arguably easier to teach. Even primary school children can handle deduction-based systems (like Prolog) but struggle with the specifics of backtracking [Kow82]. As microcontrollers have become very cheap (Arduino, for example), they have found their way to hobbyists' workshops and school and university courses. When programming for microcontrollers there are usually not enough resources to properly separate concerns using best-practice frameworks, embedded DSLs, etc., so that we neither find special LP languages for microcontrollers, nor resource-friendly implementations of LP languages embedded into C or C++, which are the de-facto standard microcontroller programming languages. As memory management and debugging on microcontrollers are difficult problems, non-professionals struggle to create complex programs.

The Microlog language is a Datalog variant with explicit call-semantics to cause side-effects and collect input interactively. The result (i. e. minimal model) of one deduction phase is fed back into the system as the extensional database, giving rise to an intentionally non-terminating[2] interactive system. Microlog can be used to express goal-based agents (Level 3 of 5 on Russell and Norvig's taxonomy of intelligent agents) [Wen21; RN10].

[1] Martin-Luther-Universität Halle-Wittenberg, Institut für Informatik,
Von-Seckendorff-Platz 1, D-06099 Halle (Saale), Germany
mario.wenzel@informatik.uni-halle.de

[2] Microcontrollers, but also servers, usually run and react to external input and actions until they are turned off.

In this paper we present a formal framework to execute Microlog programs in an abstract fashion. We precompute possible “states”, which are sets of Datalog-facts that are true at a point in time. This does not work for all Microlog programs, because one can write programs for which the number of facts keeps growing over time. This generalizes our previous work [WB20] to include negation and therefore allows for more interesting programs. We also present a heuristic to detect non-termination in this procedure.

We shortly recapitulate the Microlog language, introduce conditional facts and extend the usual Datalog operations with them. This allows us to define a model of abstract execution. Finally we show a necessary criterion for non-termination of our procedure, which is generally undecidable.

2 Microlog Recap

The definition of Microlog presented here differs from the one in previous papers as we skip explicit timestamping of facts based on the Dedalus$_0$ language [Alv+10]. The syntactic conversion between both variants is a simple transformation and the resulting program semantics are isomorphic.

Microlog is an extension of Datalog with stratified negation. A Microlog program is a finite set of stratifiable rules of the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$, where the head literal A is a positive atomic formula and the body literals B_i each are either a positive atomic formulas $p(t_1, \ldots, t_m)$, negated atomic formulas $\neg p(t_1, \ldots, t_m)$, or quantifier-free formulas in some chosen theory of first-order logic[3]. Terms t are variables, constants of the theory, or library constants, like `HIGH` or `LOW` for the Arduino digital input levels, which are prefixed with #. Rules must be range-restricted, i.e., all variables that appear anywhere in the rule must appear also in a body literal with a positive atomic formula. This ensures that all variables are bound to a value when the rule is applied, and the quantifier-free formulas can be evaluated.

In order to model information flow through time, we introduce some special predicates and syntactic rules:

- Predicates prefixed with next_ are only allowed in rule-heads. Any next_p-fact deduced becomes a p-fact in the next iteration of the deduction. This allows information to flow through time. We call rules with such a predicate in the head “inductive rules”.
- For each callable n-ary function f with arguments $a_1, \ldots, a_n$ and an return value r there exists
  - An $(n+1)$-ary predicate ret_f$(a_1, \ldots, a_n, r)$ which is only allowed in rule-bodies.

[3] e. g., Presburger Arithmetic, though the idea is, that one could drop in any theory supported by an attached SMT solver.

  – An n-ary predicate call_f$(a_1,\ldots,a_n)$ which is only allowed in rule-heads. To have a consistent argument list in the concrete syntax we write it as call_f$(a_1,\ldots,a_n,?)$ with a ? for the output positions that are not assigned a value. Placeholders for invented values have also been used in ILOG [HY90].

  Each call_-fact deduced, causes the corresponding function to be called with those arguments. Though the set-semantics ensures that duplicate calls are eliminated. i.e., even if there are different ways to deduce the fact, only one call is done. The arguments and the return value are available for the next iteration of the deduction as a ret_-fact. A rule with a call_-predicate as its head is called a "call rule". All call rules are inductive rules, as they also transport information into the future.
- All other rules are called "deductive rules".

We define the program semantics in terms of a fixed environment $\mathscr{E}$ that contains the results of all calls to ever be done a priori, as a set of ret_-facts with an additional timestamp argument, to model impure functions that return different results on subsequent calls (and have other side-effects in an actual program run). This can be used for a deterministic and reproducible simulation environment and makes for a clean definition. An actual microcontroller would just do the appropriate call to the external function with all its side-effects on demand. For an environment $\mathscr{E}$ we define an operation seed$_{\mathscr{E}}$ that converts the next_-facts to their unprefixed versions, and obtains return values for ? in the call_-facts to convert them to ret_-facts. The first argument to the seed-function is the timestamp for which to look up the function return values.

$$\begin{aligned}\text{seed}_{\mathscr{E}}(\mathscr{T},\mathscr{S}) = &\{\text{p}(t_1,\ldots,t_n) \mid \text{next\_p}(t_1,\ldots,t_n) \in \mathscr{S}\} \cup \\ &\{\text{ret\_f}(a_1,\ldots,a_n,r) \mid \text{call\_f}(a_1,\ldots,a_n) \in \mathscr{S} \\ &\wedge \text{ret\_f}(\mathscr{T},a_1,\ldots,a_n,r) \in \mathscr{E}\}\end{aligned}$$

This call-convention is a combination of action atoms [Bas+10] with its function calls in the rule head, and the earlier proposal of external atoms [CI05] that allow for pure function calls in the rule body. Special symbols in the rule head as place holders for invented values of a logic program have been used in [HY90] to denote fresh identifiers.

We obtain a state for our system using the deductive rules. Through inductive rules some data, be it from existing facts or calls to external functions, is fed back into the system as the initial input for the next state deduction.

Definition 1 (Program Semantics)**.** The semantics of the Microlog program P is the mapping from input facts $\mathscr{E}$ (the environment), a set of ret_-facts corresponding to derived call_-facts, to a sequence of minimal models (or states) $\langle \mathscr{S}_0, \mathscr{S}_1, \ldots, \rangle$. Let $T_P(S)$ be the stratum-aware fixed-point consequence operator for the program P on the extensional database S. Then $\mathscr{S}_0 = T_P(\emptyset)$ and $\mathscr{S}_n = T_P(\text{seed}_{\mathscr{E}}(n, \mathscr{S}_{n-1}))$.

Definition 2 (Program Behaviour)**.** A program behaviour is the actions performed given an environment $\mathscr{E}$. This is the sequence of call facts from the program semantics: $\langle \mathscr{C}_0, \mathscr{C}_1, \ldots, \rangle$ where $\mathscr{C}_n = \{\text{call\_f}(a_1, \ldots, a_n) \in \mathscr{S}_n\}$. This is everything that is observable about an actual program run. We cannot examine the internal state of a program run. This also means that two programs, independent of their internal state or concrete program code, are indiscernible iff they behave the same for every environment.

3 Abstract Execution

Naturally, the observable behaviour of a Microlog program depends on an environment (Definition 2). In this section we will look at the semantics of a Microlog program by "factoring out" the environment. We want to consider the semantics of the program without any knowledge of the environment by attaching conditions to facts. This model can later be used again with a concrete environment to generate a sequence of states.

Consider the program

(i) $\text{call\_in}(?)$.
(ii) $\text{zero} \leftarrow \text{ret\_in}(0)$.

We can see that $\text{call\_in}(?)$ will be in every state. With $\text{ret\_in}(n,0) \in \mathscr{E}$ the fact zero will be in the state $\mathscr{S}_n$ $(n > 0)$. It is obvious that any other concrete value for the ? will lead to zero not being in that particular state. In this case, we can conclude the following equivalence (which looks – not just coincidentally – very much like rule (ii) of the example program):

$$\text{zero} \in \mathscr{S}_n \iff \text{ret\_in}(n,0) \in \mathscr{E}$$

Stepping back further, this equivalence stems from the conditions under which the value for ? unifies with the value 0 from the rule. Somewhat informally, we can conclude:

$$\text{zero} \in \mathscr{S} \iff ? = 0$$

We will use the formalism of "conditional facts" to model facts with an attached condition. "Conditional facts" were used by Brass and Dix for characterizing and computing negation semantics [BD94].

3.1 Conditional Facts

There are a number of input values (substitutions for ?) that are are unknown at "compile time". We use special variables to model them. These variables correspond to memory locations that are used for storing return values of the function calls.

Definition 3 (Parameter Variable)**.** Let $\mathsf{V}_1, \mathsf{V}_2, \ldots$ be a sequence of pairwise distinct variables that do not occur in the given Datalog program. We call these "parameter variables".

Definition 4 (Parameterized Fact)**.** A parameterized fact is a formula of the form $p(t_1,\dots,t_m)$ where each t_i is a constant or a parameter variable.

Definition 5 (Condition)**.** A condition φ is a consistent (satisfiable) quantifier-free formula.[4]

Definition 6 (Conditional Fact)**.** A conditional fact is a formula of the form $p(t_1,\dots,t_m) \leftarrow \varphi$ where φ is a condition and $p(t_1,\dots,t_m)$ is a parameterized fact.

If the condition φ is a tautology, as a shorthand notation we do not write the implication arrow and condition down. If all conditions are tautologies and no fact contains a parameter variable, the notation and semantics coincide with normal Datalog. If a fact or state has a tautology $\top$ as its condition, we call it "unconditional".

Definition 7 (Parameterized State)**.** A parameterized state is a finite set of conditional facts.

Parameter variables have a global meaning in the state: If two parameterized facts in a state both contain some V_i, they will have the same value. This is a difference to normal variables in rules, which have only local scope and are limited to a single rule.

Definition 8 (Conditional State)**.** A conditional state $\mathscr{S} \leftarrow \varphi$ is a finite set of parameterized facts $\mathscr{S}$ and a condition φ.

Example 1 (Conditional States Through Parameterized State)**.** Consider the following parameterized state: $\{p \leftarrow \mathsf{V}_1 < 5, q \leftarrow \mathsf{V}_1 > 10, r(\mathsf{V}_2)\}$
As the conditions for p and q disagree, for any specific V_1 they can not appear together in a conditional state. $r(\mathsf{V}_2)$ is unconditional and is therefore in every conditional state.
Through case analysis, we get the following four conditional states, of which one is inconsistent and can never be obtained through an assignment of the parameter variables:

- $\{p,q,r(\mathsf{V}_2)\} \leftarrow \mathsf{V}_1 < 5 \wedge \mathsf{V}_1 > 10$ ↯
- $\{q,r(\mathsf{V}_2)\} \leftarrow \mathsf{V}_1 \geq 5 \wedge \mathsf{V}_1 > 10$
- $\{p,r(\mathsf{V}_2)\} \leftarrow \mathsf{V}_1 < 5 \wedge \mathsf{V}_1 \leq 10$
- $\{r(\mathsf{V}_2)\} \leftarrow \mathsf{V}_1 \geq 5 \wedge \mathsf{V}_1 \leq 10$

We define the operation cd as a mapping from a set of conditional facts $\mathscr{C}$ to a set of conditional states with every possible combination of conditions and their negations. Naively, a fact exists in the state iff its condition is in the condition for the state in non-negated form. Note that we are only asking about the existence of a model at every point and are not interested in specific variable assignments for the parameter variables:

$$\begin{aligned}
\mathrm{cd}(\mathscr{C}) = \{\mathscr{C}' \leftarrow \varphi' \mid \varphi' = (\textstyle\bigwedge \varphi_+ \wedge \textstyle\bigwedge \varphi_-) \wedge \exists M : M \models \varphi' \\
\wedge \varphi_+ \in 2^{\varphi} \wedge \varphi_- = \{\neg c \mid c \in \varphi \setminus \varphi_+\} \\
\wedge \varphi = \{c \mid f \leftarrow c \in \mathscr{C}\} \\
\wedge \mathscr{C}' = \{f \mid f \leftarrow c \in \mathscr{C} \wedge c \in \varphi_+\}\}
\end{aligned}$$

Our oracle or SMT solver deciding $\exists M : M \models \varphi'$ could overapproximate or always return true. This is okay, as at runtime we are deciding which specific state is actually the correct

[4] In the chosen theory.

state, given a concrete variable assignment. This allows for generally undecidable theories to be used as well. The only "danger" is, that we are generating actually unreachable states.

3.2 Abstract Rule Application

In this section we will examine rule application using conditional facts in order to construct a set of reachable states for our Microlog program.

Definition 9 (Unification Condition)**.** Let $\theta = \{X_1 \mapsto Y_1, \dots, X_n \mapsto Y_n\}$ be a most general unifier of two literals that does not map parameter variables to variables of the rule (since the direction of variable-to-variable bindings is arbitrary, this is always possible). Then the unification condition φ_θ is $\varphi_\theta = \bigwedge\{X_i = X_i\theta \mid (X_i \mapsto Y_i) \in \theta, X_i \text{ is a parameter variable}\}$. This is exactly the condition under which this unification succeeds.

Let $\theta_{\mathscr{E}}$ be a mapping from parameter values to actual values from the environment (or constants from a definition) during an actual run and φ_θ the unification condition for two literals A and B, then substituting the parameter values in the condition with actual values from the environment makes the substitution condition true iff unification still succeeds under this refinement: $(A\theta = B\theta) \rightarrow ((A\theta\theta_{\mathscr{E}} = B\theta\theta_{\mathscr{E}}) \iff \varphi_\theta\theta_{\mathscr{E}})$

Definition 10 (Rule Application to Conditional Facts)**.**
Let $A \leftarrow B_1 \wedge \cdots \wedge B_m \wedge C_1 \wedge \cdots \wedge C_n \wedge \neg D_1 \wedge \cdots \wedge \neg D_o$ be a rule, where

- B_i, $i = 1, \dots, m$, are ordinary positive literals,
- C_i, $i = 1, \dots, n$, are literals with a built-in predicate (i. e., formulas of the chosen theory),
- $\neg D_i$, $i = 1, \dots, o$, are ordinary negative literals.

Let $B'_i \leftarrow \varphi_i$, $i = 1, \dots, m$, be conditional facts and θ be a most general unifier for $(B_1, \dots, B_m)$ and $(B'_1, \dots, B'_m)$ that does not map parameter variables to variables of the rule. $(B'_1, \dots, B'_m)$ are the facts used in one rule application.

Let $B''_i \leftarrow \varphi'_i$, $i = 1, \dots, j$, be all conditional facts. Let $\theta_{j,o}$ be all $1, \dots, j$, $1, \dots, o$ most general unifiers for all $\{D_1\theta, \dots, D_o\theta\}$ and all $\{B''_1, \dots, B''_j\}$ that do not map parameters to variables of the rule.

Let
$$\begin{aligned}\Phi := \varphi_\theta \wedge & \\ & \bigwedge(\{\varphi_i \mid i = 1, \dots, m\} \cup \{C_i\theta \mid i = 1, \dots, n\}) \wedge \\ & \neg \bigvee\{\varphi'_i \wedge \varphi_{\theta_{i,i'}} \mid \text{for each } \theta_{i,i'} \text{ with } i = 1, \dots, j,\ i' = 1, \dots, o\}\end{aligned}$$

Φ is a conjunction of

1. the unification condition for unification of the positive body literals with respective known conditional facts,
2. the conditions of all conditional facts used in the rule application,

3. the additional formulas of the rule (using the substitution from the unification), and
4. a negated disjunction that represents all the possible conditions that are sufficient for a fact to exist, that would make the rule application fail due to a negated body literal.

If Φ is consistent, then the rule application yields $A\theta \leftarrow \varphi$, where φ is equivalent to Φ. Else, the rule application is not possible.

Example 2 (Rule Application)**.** Consider the rule $p(X) \leftarrow q(X) \wedge r(X) \wedge \neg s(X) \wedge X > 5$ with the conditional facts $\{q(\mathsf{V}_1) \leftarrow \mathsf{V}_1 < 7, q(2), r(\mathsf{V}_2), r(10), s(\mathsf{V}_3)\}$. Up to direction of the variable bindings there are three possible rule instances:

- $\theta_1 = \{X \mapsto \mathsf{V}_1, \mathsf{V}_2 \mapsto \mathsf{V}_1\}$ with the substitution condition $\varphi_{\theta_1} = (\mathsf{V}_2 = \mathsf{V}_1)$ and the substitution $\theta_{1'} = \{\mathsf{V}_1 \mapsto \mathsf{V}_3\}$ for the negation.
The obtained conditional fact is $p(\mathsf{V}_1) \leftarrow \mathsf{V}_1 > 5 \wedge \mathsf{V}_2 = \mathsf{V}_1 \wedge \mathsf{V}_1 < 7 \wedge \mathsf{V}_1 \neq \mathsf{V}_3$
- $\theta_2 = \{X \mapsto 10, \mathsf{V}_1 \mapsto 10\}$ with the substitution condition $\varphi_{\theta_2} = (\mathsf{V}_1 = 10)$ and $\varphi_{\theta_{2'}} = (10 = \mathsf{V}_3)$ for the negation. The conditional fact is $p(10) \leftarrow \mathsf{V}_1 = 10 \wedge 10 \neq \mathsf{V}_3 \wedge 10 < 7 \lightning$ but as the condition is not consistent, the rule application fails.
- $\theta_3 = \{X \mapsto 2, \mathsf{V}_2 \mapsto 2\}$ with the substitution condition $\varphi_{\theta_3} = (\mathsf{V}_2 = 2)$ and $\varphi_{\theta_{3'}} = (2 = \mathsf{V}_3)$ for the negation. The obtained conditional fact is $p(2) \leftarrow \mathsf{V}_2 = 2 \wedge 2 \neq \mathsf{V}_3 \wedge 2 > 5 \lightning$, which is inconsistent and the rule application fails.

3.3 Abstract Deduction

We will define the abstract deduction for a program P. using the abstract rule application we have defined above. We define a consequence operator $\check{T}$ that works like the usual stratum-aware fixed-point consequence operator from literature [CGT90] but uses the rule application from Definition 10 instead of the normal immediate consequence operator. This means that for any rule application that yields a fact, the operator also adds to the fact the conditions for 1. unification if a parameter variable is involved, 2. existing conditions of the facts involved in the rule application, 3. formulas of the rule if they involve parameter variables, and 4. negated unification conditions for all matching facts for negative body literals.

Any state $\mathscr{S}_n$ is a parameterized state and is identified by a set of parameterized facts, i. e., a set of seed-facts $\mathscr{S}'_n$, which are the initial facts for a state. The initial state is $\mathscr{S}_0 = \check{T}_P(\emptyset)$. It has no seed facts as there is no preceding state to cause next_/ret_-facts.

Now let any parameterized state $\mathscr{S}_n$ and its seed facts $\mathscr{S}'_n$ be given (for instance, the initial one). Our goal is to compute the possible successor states. Through case distinction we will find the possible instantiations (conditional states) of our state $\mathscr{S}_n$. Let the possible instantiations of $\mathscr{S}_n$ be $\{\mathscr{S}_{n,1} \leftarrow \varphi_1, \ldots, \mathscr{S}_{n,i} \leftarrow \varphi_i\} = \mathrm{cd}(\mathscr{S}_n)$. We get one instantiation for each consistent valuation of the atomic formulas appearing in the conditions in $\mathscr{S}_n$.

We define an operation seed that returns a set containing

- for each next_-fact a fact without the next_-prefix, as above
- for each call_-fact a ret_-fact with a "fresh" parameter variable instead of the return value indicator "?". Instead of looking up the return value or doing the actual call, we introduce a new parameter variable.

Definition 11 (Seed Facts)**.** Let $\mathscr{S}$ be a set of parameterized facts, and let

- $\text{next\_p}_i(t_{i,1},\ldots,t_{i,k_i})$ for $i = 1,\ldots,m$ be all (parameterized) next_p-facts in $\mathscr{S}$, and
- $\text{call\_f}_i(u_{i,1},\ldots,u_{i,l_i})$ for $i = 1,\ldots,n$ be all (parameterized) call_-facts in $\mathscr{S}$.

Then the seed facts $\text{seed}(\mathscr{S})$ for the next state are:

- $p_i(t_{i,1},\ldots,t_{i,k_i})$ for $i = 1,\ldots,m$, and
- $\text{ret\_f}_i(\hat{u}_{i,1},\ldots,\hat{u}_{i,l_i})$ for $i = 1,\ldots,n$, where $\hat{u}_{i,j}$ is $u_{i,j}$, unless $u_{i,j}$ is ?, in which case $\hat{u}_{i,j}$ is the first currently unused parameter variable (not occurring in any next_- or call_-facts in $\mathscr{S}$, and not substituted already for ? in a previous call_f fact in $\mathscr{S}$ or an argument to the left in the same fact).

The computation of the next state starts with the the seed facts from the previous state, which are the (parameterized) call_-facts and the next_-facts.

Definition 12 (Successor State)**.** Given an instantiation $\mathscr{S}_{n,i} \leftarrow \varphi_i$ of a conditional state $\mathscr{S}_n$ and a condition φ_i the seed for successor state under the condition φ_i is $\text{seed}(\mathscr{S}_{n,i}) = \mathscr{S}'_m$ and the successor state is $\mathscr{S}_m = \check{T}_P(\mathscr{S}'_m)$.

We give the following algorithm to enumerate all possible conditional states of a Microlog program.

1. We calculate $\mathscr{S}_0$ using the stratum-aware conditional consequence operator $\check{T}_P$.
2. Through case distinction $\text{cd}(\mathscr{S}_0)$ we obtain possible instantiations $\mathscr{S}_{0,1} \leftarrow \varphi_1,\ldots,\mathscr{S}_{0,i} \leftarrow \varphi_i$.
3. We obtain seed facts for possible successor states by applying the seed-function to the instantiations, yielding new seed-facts: $\text{seed}(\mathscr{S}_{0,j}) = \mathscr{S}'_m$
4. We obtain a new parameterized state by application of the consequence operator: $\mathscr{S}_m = \check{T}_P(\mathscr{S}'_m)$
5. We continue with a case distinction on $\mathscr{S}_m$ and repeat until no new states are obtained.

We create a transition function t that maps a set of seed facts and a condition to another set of seed facts. The full conditional state is always deterministically determined by the consequence operator. The condition from the case distinction is used to distinguish the transition taken, given a assignment for the parameter variables. We give a co-recursive definition for $t(s)$ with s being the seed facts for the (initial) state, i. e., $\mathscr{S}'_0 = \emptyset$.

$$t(s) = \bigcup\{\{(s,\varphi_i) \mapsto \text{seed}(s_i)\} \cup t(\text{seed}(s_i)) \mid s_i \leftarrow \varphi_i \in \text{cd}(\check{T}_P(s))\}$$

Once an already known state seed is discovered again, this recursive call should stop and reference the known state instead of continuing. We obtain a transition function like this:

$$\begin{aligned} t(\mathscr{S}_0) = \{ & (\mathscr{S}'_0, \varphi_1) \mapsto \text{seed}(\mathscr{S}_{0,1}) = \mathscr{S}'_i \\ & \ldots \\ & (\mathscr{S}'_0, \varphi_n) \mapsto \text{seed}(\mathscr{S}_{0,n}) = \mathscr{S}'_j \\ & (\mathscr{S}'_i, \varphi_{\ldots}) \mapsto \ldots \\ & \ldots \\ & (\mathscr{S}'_j, \varphi_{\ldots}) \mapsto \ldots \} \end{aligned}$$

The functions that need to be called upon entering a state are exactly the necessary functions to obtain values for ? when transforming the next_-facts from the previous state to the ret_-facts in the seed of the current state.

The “Toggle” program switches the state of an LED when a button is pressed, i. e. pushed down and released. The light state should switch on the high-to-low transition of the button state. Checking whether the button is currently pressed is not enough, as this would flicker the LED’s state while the button is held down. It has the following rules:

(i) $\text{call\_digitalRead}(12, ?)$.
(ii) $\text{isPressed} \leftarrow \text{ret\_digitalRead}(12, \texttt{\#HIGH})$.
(iii) $\text{next\_wasPressed} \leftarrow \text{isPressed}$.
(iv) $\text{isReleased} \leftarrow \text{wasPressed} \wedge \neg\,\text{isPressed}$.
(v) $\text{next\_lightOn} \leftarrow \text{isReleased} \wedge \neg\,\text{lightOn}$.
(vi) $\text{next\_lightOn} \leftarrow \neg\,\text{isReleased} \wedge \text{lightOn}$.
(vii) $\text{call\_digitalWrite}(13, \texttt{\#HIGH}) \leftarrow \text{lightOn}$.
(viii) $\text{call\_digitalWrite}(13, \texttt{\#LOW}) \leftarrow \neg\,\text{lightOn}$.

Executing the state enumeration as described (the full example can be found in Appendix A) we obtain the following possible seeds (and therefore states):

$$\begin{aligned} I &\mapsto \{\} \\ II &\mapsto \{\text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#LOW})\} \\ III &\mapsto \{\text{wasPressed}, \text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#LOW})\} \\ IV &\mapsto \{\text{lightOn}, \text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#LOW})\} \\ V &\mapsto \{\text{lightOn}, \text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#HIGH})\} \\ VI &\mapsto \{\text{lightOn}, \text{wasPressed}, \text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#HIGH})\} \\ VII &\mapsto \{\text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#HIGH})\} \end{aligned}$$

The ret_-facts need to be obtained by calling the corresponding function on entry into the state. This is also a “blueprint” for a possible environment. We obtain the transition function between states depending on parameter values in the state obtained by function calls:

$$
\begin{aligned}
(I,\top) &\mapsto II \\
(II,\mathsf{V}_1 = \texttt{\#HIGH}) &\mapsto III & (V,\mathsf{V}_1 = \texttt{\#HIGH}) &\mapsto VI \\
(II,\mathsf{V}_1 \neq \texttt{\#HIGH}) &\mapsto II & (V,\mathsf{V}_1 \neq \texttt{\#HIGH}) &\mapsto V \\
(III,\mathsf{V}_1 = \texttt{\#HIGH}) &\mapsto III & (VI,\mathsf{V}_1 = \texttt{\#HIGH}) &\mapsto VI \\
(III,\mathsf{V}_1 \neq \texttt{\#HIGH}) &\mapsto IV & (VI,\mathsf{V}_1 \neq \texttt{\#HIGH}) &\mapsto VII \\
(IV,\mathsf{V}_1 = \texttt{\#HIGH}) &\mapsto VI & (VII,\mathsf{V}_1 = \texttt{\#HIGH}) &\mapsto III \\
(IV,\mathsf{V}_1 \neq \texttt{\#HIGH}) &\mapsto V & (VII,\mathsf{V}_1 \neq \texttt{\#HIGH}) &\mapsto II
\end{aligned}
$$

It is not surprising that we get 6 states (besides the initial state). We switch the LED on the #HIGH-#LOW-edge of the button input. We need two bits of information to detect that edge. Whether the resulting switch of the LED needs to be from #HIGH to #LOW or from #LOW to #HIGH needs another bit of information (exactly the current state of the LED). With our 6 states we are within the expected size of the state space of no more than 2^3.

A visualisation of this example can be seen in Figure 1. The parameter variable V_1 is available in all states but I, as all these states contain ret_digitalRead$(12,\mathsf{V}_1)$ in their seed facts.

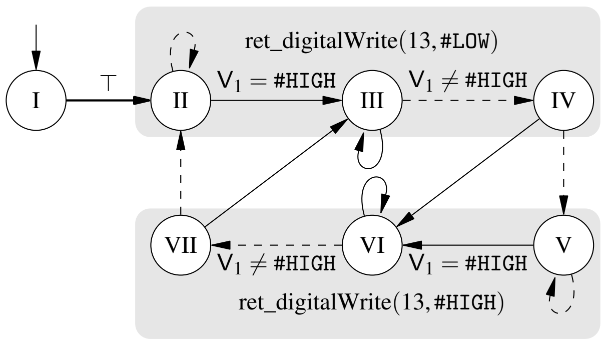

Fig. 1: Visualisation for Abstract Execution of the “Toggle” Program

3.4 Termination of State Enumeration

As there are programs that use an unbounded amount of memory over time, it is clear that the algorithm above can not terminate for those programs. We call the programs using a bounded amount of memory over time “convergent”, and otherwise “divergent”. On the other hand, if the state enumeration algorithm terminates, the number of memory locations the program uses over time is bounded. The same is true for the number of seed facts. A bound for either is a proof for program convergence.

Definition 13 (Microlog Program Convergence)**.** A Microlog program P is convergent iff there exists a number $n \in \mathbb{N}$ such that for all states $\mathscr{S}_m$ and all environments $\mathscr{E}$, the cardinality of the state is no larger than n. $\exists (n \in \mathbb{N})\ \forall (m \in \mathbb{N})\ (|\mathscr{S}_m| \leq n)$

Consider the usual termination argument for Datalog programs: The deduction procedure terminates as there are only a finite number of constants and a finite number of predicates. Once all constants have appeared in all combinations of positions in all predicates, no new facts can be obtained. The same is true if you consider a fixed and finite set of parameter variables in addition to the constants appearing in the program.

Lemma 1. By pigeonhole principle, if the size of the states grows indefinitely over time, new states must be obtained by an ever-increasing number of new parameter variables, as possible facts containing a fixed set of constants and parameter variables will be exhausted.

If a state has $\mathscr{S}_n$ has m distinct parameter variables, we write this as $|\mathscr{S}_n|_\mathsf{V} = m$.

Lemma 2. If there exists an infinite number of states, we have an infinite chain of states where $|\mathscr{S}_m|_\mathsf{V} < |\mathscr{S}_n|_\mathsf{V} < \dots$ and, of course, at least one pair of states (chain of length 2) where $|\mathscr{S}_m|_\mathsf{V} < |\mathscr{S}_n|_\mathsf{V}$.

As usual in termination analysis [BN98], if we can give a measure $>_\alpha$ in which $\mathscr{S}_m >_\alpha \mathscr{S}_n$ and there are no infinite decreasing chains in $>_\alpha$, then this is not a problematic pair, as the operation that leads from $\mathscr{S}_m$ to $\mathscr{S}_n$ can not be repeated indefinitely.

Definition 14 (Problematic Pair of States)**.** Two states $\mathscr{S}_m$ and $\mathscr{S}_n$ form a problematic pair iff $|\mathscr{S}_m|_\mathsf{V} < |\mathscr{S}_n|_\mathsf{V}$ and in the well-founded (partial) order $>_\alpha$ $\mathscr{S}_m >_\alpha \mathscr{S}_n$ does not hold. For such a problematic pair we say $\mathscr{S}_m \prec \mathscr{S}_n$.

Theorem 1. Iff a Microlog program is diverging, there exists at least one infinite chain of reachable states $\mathscr{S}_m, \mathscr{S}_n, \dots$ where $\mathscr{S}_m \prec \mathscr{S}_n \prec \dots$, as that constitutes an infinite chain $|\mathscr{S}_m|_\mathsf{V} < |\mathscr{S}_n|_\mathsf{V} < \dots$.

Theorem 2. If for each pair of states $|\mathscr{S}_m|_\mathsf{V} < |\mathscr{S}_n|_\mathsf{V}$ it can be shown that $\mathscr{S}_m >_\alpha \mathscr{S}_n$, then there exists no pair $\mathscr{S}_m \prec \mathscr{S}_n$ (chain of length 2) and therefore no infinite chain either.

Now we define $>_\alpha$ for $\mathscr{S}_i >_\alpha \mathscr{S}_j$ in such a manner, that if we observe "growth" (difference between sets) from $\mathscr{S}_i$ to $\mathscr{S}_j$, i. e., the potential for an infinite chain, that kind of growth must be infinitely repeatable. If not, they can not form a part of the same chain. $\mathscr{S}_i >_\alpha \mathscr{S}_j$ does hold iff not all of the following hold:

1. The number of facts for each predicate in $\mathscr{S}_j$ is equal or larger than for the same predicate in $\mathscr{S}_i$.
2. The number of facts for at least one predicate in $\mathscr{S}_j$ is strictly larger than for the same predicate in $\mathscr{S}_i$.
3. Given a substitution θ_ε that maps all parameter variables to the same special value ε, $\mathscr{S}_i\theta_\varepsilon = \mathscr{S}_j\theta_\varepsilon$. This means that, even in facts were both program constants and parameter variables occur, growth is only observed in the argument positions with parameter variables and not in combination with a recombination of program constants, which would only possible a finite amount of times.[5]

[5] Corresponds to the measure "facts missing until all possible facts without parameter variables are exhausted".

Example 3 (Problematic Pairs)**.**

- $\{p(\mathsf{V}_1,\mathsf{V}_2), p(\mathsf{V}_1,\mathsf{V}_3)\} \not\prec \{p(\mathsf{V}_1,\mathsf{V}_2), q(\mathsf{V}_3,\mathsf{V}_4), q(\mathsf{V}_5,\mathsf{V}_6)\}$ as p-facts can not be removed indefinitely often (condition 1).
- $\{p(\mathsf{V}_1,\mathsf{V}_1), p(\mathsf{V}_1,\mathsf{V}_2)\} \not\prec \{p(\mathsf{V}_1,\mathsf{V}_2), p(\mathsf{V}_3,\mathsf{V}_4)\}$ as the arguments, up to renaming, can not be recombined arbitrarily often (condition 2).
- $\{p(\mathsf{V}_1)\} \not\prec \{p(\mathsf{V}_1), p(\mathsf{V}_2), q\}$ as facts without parameter variables can not be added indefinitely (condition 3).
- $\{p(1,\mathsf{V}_1)\} \not\prec \{p(1,\mathsf{V}_1), p(2,\mathsf{V}_2)\}$ as facts with different program constants can not be added indefinitely (condition 3).
- Otherwise: $\{p(1,\mathsf{V}_1)\} \prec \{p(1,\mathsf{V}_1), p(1,\mathsf{V}_2)\}$, for example.

If during the state enumeration a pair of states fulfils $\mathscr{S}_i \prec \mathscr{S}_j$, the program admits a chain containing $\mathscr{S}_i, \mathscr{S}_j$. It is possible that $\mathscr{S}_i, \mathscr{S}_j$ are part of an infinite chain. That $\mathscr{S}_i \prec \mathscr{S}_j$ is necessary but not sufficient for the existence of such an infinite chain, as $\mathscr{S}_i, \mathscr{S}_j$ might be part of a chain of finite length instead. As this is undecidable in general, we stop the state enumeration at that point.

4 Conclusion

For space reasons we did not discuss the recovery of the set semantics for the case where parameter variables are not distinct (a relaxation of Definition 3) or where other equalities between parameter variables and other constants of the program or library exists.

We have shown that it is possible to execute Microlog programs in an abstract fashion. States are connected by transitions depending on external input, which is gathered upon entering a state. The state enumeration algorithm is not terminating for some programs and that property is undecidable. We have given a necessary criterion for non-termination which allows us, upon discovery, to stop the enumeration process.

Of course, there must be false positives in that decision procedure. And if the user knows that the procedure eventually terminates as the critical pairs are not on infinite but on finite chains of growing states, they can add additional rules to “break the chain”. We do not have shown that procedure here, but it basically consists of adding distinguishing nullary predicates between such a problematic pair, breaking condition 3. This can be done automatically. Of course, for infinite chains this is not possible and would lead to infinitely many additional nullary predicates.

Once such a state enumeration is complete, we can use finite state machine compilation techniques in order to compile this program.

Our compiler, as well as further convergent and divergent example programs – like a Turing machine template – is available at `https://dbs.informatik.uni-halle.de/microlog/`.

References

[Alv+10] Peter Alvaro, William R. Marczak, Neil Conway, Joseph M. Hellerstein, David Maier, and Russell Sears. "Dedalus: Datalog in Time and Space". In: *Datalog Reloaded - First International Workshop, Datalog 2010, Oxford, UK, March 16-19, 2010. Revised Selected Papers*. Ed. by Oege de Moor, Georg Gottlob, Tim Furche, and Andrew Jon Sellers. Vol. 6702. Lecture Notes in Computer Science. Springer, 2010, pp. 262–281. DOI: `10.1007/978-3-642-24206-9\_16`.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998. ISBN: 978-0-521-45520-6.

[Bas+10] Selen Basol, Ozan Erdem, Michael Fink, and Giovambattista Ianni. "HEX Programs with Action Atoms". In: *Technical Communications of the 26th International Conference on Logic Programming, ICLP 2010, July 16-19, 2010, Edinburgh, Scotland, UK*. Ed. by Manuel V. Hermenegildo and Torsten Schaub. Vol. 7. LIPIcs. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2010, pp. 24–33. DOI: `10.4230/LIPIcs.ICLP.2010.24`.

[BD94] Stefan Brass and Jürgen Dix. "A general Approach to Bottom-Up Computation of Disjunctive Semantics". In: *Non-Monotonic Extensions of Logic Programming (NMELP'94), ICLP '94 Workshop, Santa Margherita Ligure, Italy, June 17, 1994, Selected Papers*. Ed. by Jürgen Dix, Luís Moniz Pereira, and Teodor C. Przymusinski. Vol. 927. Lecture Notes in Computer Science. Springer, 1994, pp. 127–155. DOI: `10.1007/BFb0030663`.

[CI05] Francesco Calimeri and Giovambattista Ianni. "External Sources of Computation for Answer Set Solvers". In: *Logic Programming and Nonmonotonic Reasoning, 8th International Conference, LPNMR 2005, Diamante, Italy, September 5-8, 2005, Proceedings*. Ed. by Chitta Baral, Gianluigi Greco, Nicola Leone, and Giorgio Terracina. Vol. 3662. Lecture Notes in Computer Science. Springer, 2005, pp. 105–118. DOI: `10.1007/11546207\_9`.

[CGT90] Stefano Ceri, Georg Gottlob, and Letizia Tanca. *Logic Programming and Databases*. Surveys in computer science. Springer, 1990. ISBN: 3-540-51728-6. URL: `http://www.worldcat.org/oclc/20595273`.

[HY90] Richard Hull and Masatoshi Yoshikawa. "ILOG: Declarative Creation and Manipulation of Object Identifiers". In: *16th International Conference on Very Large Data Bases, August 13-16, 1990, Brisbane, Queensland, Australia, Proceedings*. Ed. by Dennis McLeod, Ron Sacks-Davis, and Hans-Jörg Schek. Morgan Kaufmann, 1990, pp. 455–468. URL: `http://www.vldb.org/conf/1990/P455.PDF`.

[Kow82] Robert A. Kowalski. "Logic as a Computer Language for Children". In: *5th European Conference on Artificial Intelligence, ECAI 1982, Paris, 1982, Proceedings*. 1982, pp. 2–10.

[RN10] Stuart J. Russell and Peter Norvig. *Artificial Intelligence - A Modern Approach, Third International Edition*. Pearson Education, 2010. ISBN: 978-0-13-207148-2. URL: `http://vig.pearsoned.com/store/product/1,1207,store-12521%5C_isbn-0136042597,00.html`.

[Wen21] Mario Wenzel. “Expressivity of the Microlog Language”. In: *WLP 2021 - 35th Workshop on (Constraint) Logic Programming*. 2021. URL: `https://maweki.de/files/art_wlp21.pdf`.

[WB20] Mario Wenzel and Stefan Brass. “Translation of Interactive Datalog Programs for Microcontrollers to Finite State Machines”. In: *Logic-Based Program Synthesis and Transformation - 30th International Symposium, LOPSTR 2020, Bologna, Italy, September 7-9, 2020, Proceedings*. Ed. by Maribel Fernández. Vol. 12561. Lecture Notes in Computer Science. Springer, 2020, pp. 210–227. DOI: `10.1007/978-3-030-68446-4\_11`.

A Example Abstract Execution of Toggle Program

(i) $\text{call\_digitalRead}(12, ?)$.
(ii) $\text{isPressed} \leftarrow \text{ret\_digitalRead}(12, \texttt{\#HIGH})$.
(iii) $\text{next\_wasPressed} \leftarrow \text{isPressed}$.
(iv) $\text{isReleased} \leftarrow \text{wasPressed} \wedge \neg\, \text{isPressed}$.
(v) $\text{next\_lightOn} \leftarrow \text{isReleased} \wedge \neg\, \text{lightOn}$.
(vi) $\text{next\_lightOn} \leftarrow \neg\, \text{isReleased} \wedge \text{lightOn}$.
(vii) $\text{call\_digitalWrite}(13, \texttt{\#HIGH}) \leftarrow \text{lightOn}$.
(viii) $\text{call\_digitalWrite}(13, \texttt{\#LOW}) \leftarrow \neg\, \text{lightOn}$.

1. The initial state never has ret_-facts, therefore we start with an empty set of seed facts. We call this state I and we deduce $\text{call\_digitalWrite}(13, \texttt{\#LOW})$ and $\text{call\_digitalRead}(12, ?)$ unconditionally and no other fact. We obtain the inductive facts

   $$\{\text{call\_digitalRead}(12, ?), \text{call\_digitalWrite}(13, \texttt{\#LOW})\}$$

   We extract all call_-facts into ret_-facts (replacing all ? by fresh parameter variables) and next_-facts (none in this case) into their non-prefixed version. This is the seed-transformation.
   The seed for the single subsequent state is $\{\text{ret\_digitalRead}(12, \mathsf{V}_1)$, $\text{ret\_digitalWrite}(13, \texttt{\#LOW})\}$. We call this state II. We obtain the following transition for our transition function:

   $$(I, \top) \mapsto II$$

2. Using the state II seed facts we now we obtain the parameterized state

   $$\begin{aligned}&\{\text{ret\_digitalRead}(12, \mathsf{V}_1), \text{ret\_digitalWrite}(13, \texttt{\#LOW}),\\ &\ \text{call\_digitalRead}(12, ?), \text{isPressed} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH},\\ &\ \text{next\_wasPressed} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH}, \text{call\_digitalWrite}(13, \texttt{\#LOW})\}\end{aligned}$$

   Of course, only the information for the subsequent state is important so that we obtain the inductive facts

   $$\begin{aligned}&\{\text{call\_digitalRead}(12, ?), \text{call\_digitalWrite}(13, \texttt{\#LOW}),\\ &\ \text{next\_wasPressed} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH}\}\end{aligned}$$

   We do a case distinction over the comparison and obtain the two possible subsequent conditional states

   - $\{\text{call\_digitalRead}(12, ?), \text{call\_digitalWrite}(13, \texttt{\#LOW})$, $\text{next\_wasPressed}\} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH}$, which (after seed-transformation) we call state III.

- $\{\text{call\_digitalRead}(12,?), \text{call\_digitalWrite}(13,\texttt{\#LOW})\}$
  $\leftarrow \mathsf{V}_1 \neq \texttt{\#HIGH}$, which leads again to state II.

We obtain the following transitions for our transition function:

$$(II, \mathsf{V}_1 = \texttt{\#HIGH}) \mapsto III$$
$$(II, \mathsf{V}_1 \neq \texttt{\#HIGH}) \mapsto II$$

and we continue with state III.

3. From state III, $\{\text{ret\_digitalRead}(12,\mathsf{V}_1), \text{ret\_digitalWrite}(13,\texttt{\#LOW}),$ $\text{wasPressed}\}$, we obtain the following possible sets of inductive facts:

   - $\{\text{call\_digitalRead}(12,?), \text{call\_digitalWrite}(13,\texttt{\#LOW}),$ $\text{next\_wasPressed}\} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH}$, which leads again to state III
   - $\{\text{call\_digitalRead}(12,?), \text{call\_digitalWrite}(13,\texttt{\#LOW}),$ $\text{next\_lightOn}\} \leftarrow \mathsf{V}_1 \neq \texttt{\#HIGH}$, which (after seed-transformation) we call state IV

   We obtain the following transitions for our transition function:

$$(III, \mathsf{V}_1 = \texttt{\#HIGH}) \mapsto III$$
$$(III, \mathsf{V}_1 \neq \texttt{\#HIGH}) \mapsto IV$$

   and we continue with state IV.

4. From state IV, $\{\text{ret\_digitalRead}(12,\mathsf{V}_1), \text{ret\_digitalWrite}(13,\texttt{\#LOW}),$ $\text{lightOn}\}$, we obtain the following inductive facts:

   - $\{\text{call\_digitalRead}(12,?), \text{call\_digitalWrite}(13,\texttt{\#HIGH}),$ $\text{next\_lightOn}, \text{next\_wasPressed}\} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH}$, which (after seed-transformation) we call state VI
   - $\{\text{call\_digitalRead}(12,?), \text{call\_digitalWrite}(13,\texttt{\#HIGH}),$ $\text{next\_lightOn}\} \leftarrow \mathsf{V}_1 \neq \texttt{\#HIGH}$, which (after seed-transformation) we call state V

   We obtain the following transitions for our transition function:

$$(IV, \mathsf{V}_1 = \texttt{\#HIGH}) \mapsto VI$$
$$(IV, \mathsf{V}_1 \neq \texttt{\#HIGH}) \mapsto V$$

   and we continue with state V.

5. From state V, $\{\text{ret\_digitalRead}(12,\mathsf{V}_1), \text{ret\_digitalWrite}(13,\texttt{\#HIGH}),$ $\text{lightOn}\}$, we obtain the following possible sets of inductive facts:

   - $\{\text{call\_digitalRead}(12,?), \text{call\_digitalWrite}(13,\texttt{\#HIGH}),$ $\text{next\_wasPressed}, \text{next\_lightOn},\} \leftarrow \mathsf{V}_1 = \texttt{\#HIGH}$, which leads again to state VI

- $\{$call_digitalRead$(12, ?)$, call_digitalWrite$(13, \texttt{\#HIGH})\}$
  $\leftarrow \mathsf{V}_1 \neq \texttt{\#HIGH}$, which leads again to state V

We obtain the following transitions for our transition function:

$$(V, \mathsf{V}_1 = \texttt{\#HIGH}) \mapsto VI$$
$$(V, \mathsf{V}_1 \neq \texttt{\#HIGH}) \mapsto V$$

6. From state VI, $\{$ret_digitalRead$(12, \mathsf{V}_1)$, ret_digitalWrite$(13, \texttt{\#HIGH})$, wasPressed, lightOn$\}$, we obtain the following possible sets of inductive facts:

- $\{$call_digitalRead$(12, ?)$, call_digitalWrite$(13, \texttt{\#HIGH})$, next_wasPressed, next_lightOn$\}$ $\leftarrow \mathsf{V}_1 = \texttt{\#HIGH}$, which leads again to state VI
- $\{$call_digitalRead$(12, ?)$, call_digitalWrite$(13, \texttt{\#HIGH})\}$
  $\leftarrow \mathsf{V}_1 \neq \texttt{\#HIGH}$, which (after seed-transformation) we call state VII

We obtain the following transitions for our transition function:

$$(V, \mathsf{V}_1 = \texttt{\#HIGH}) \mapsto VI$$
$$(V, \mathsf{V}_1 \neq \texttt{\#HIGH}) \mapsto VII$$

7. From state VII, $\{$ret_digitalRead$(12, \mathsf{V}_1)$, ret_digitalWrite$(13, \texttt{\#HIGH})\}$, we obtain the following possible sets of inductive facts:

- $\{$call_digitalRead$(12, ?)$, call_digitalWrite$(13, \texttt{\#LOW})$, next_wasPressed$\}$ $\leftarrow \mathsf{V}_1 = \texttt{\#HIGH}$, which leads again to state III
- $\{$call_digitalRead$(12, ?)$, call_digitalWrite$(13, \texttt{\#LOW})\}$
  $\leftarrow \mathsf{V}_1 \neq \texttt{\#HIGH}$, which leads again to state II

We obtain the following transitions for our transition function:

$$(VI, \mathsf{V}_1 = \texttt{\#HIGH}) \mapsto III$$
$$(VI, \mathsf{V}_1 \neq \texttt{\#HIGH}) \mapsto II$$

State VII is different from state II since in VII we start with the lights turned on, turning them off, and in state II we start with the light already turned off. Both states differ in their seed-facts but not in their successor states.