Integrity Constraints for Microcontroller Programming in Datalog

Stefan Brass and Mario Wenzel

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

Abstract. We consider microcontroller-programming with a declarative language based on the logic-programming language Datalog. Our prototype implementation translates a Datalog dialect to C-code for the Arduino IDE. In order to prove the correctness, one must ensure that the very limited memory of the microcontroller is sufficient for the derived facts. In this paper, we propose a class of constraints called "generalized exclusion constraints" that can be used for this task. Moreover, they are needed to exclude conflicting commands to the hardware, e.g. different output values on a pin in the same state. This class of constraints also generalizes keys and functional dependencies, therefore our results also help to prove such constraints for derived predicates.

1 Introduction

A microcontroller is a small computer on a single chip. For instance, the Amtel ATMega328P contains an 8-bit CPU, 32 KByte Flash Memory for the program, 2 KByte static RAM, 1 KByte EEPROM for persistent data, 23 general purpose I/O pins, timers, analog/digital-converters, pulse-width modulators, and support for serial interfaces. It costs less than 2 dollars and consumes little energy. Microcontrollers are used in many electronic devices.

For hobbyists, schools, and the simple development of prototypes, the Arduino platform is quite often used. It basically consists of a few variants of boards with the microcontroller, fitting hardware extension boards ("shields"), and an IDE with a programming language based on C.

The software for microcontrollers is often developed in Assembler or C. We believe that declarative programming can be an interesting option even for such small devices. In [8,9], we proposed a language "Microlog" for programming Microcontrollers like on the Arduino. The language is based on Datalog, which is a simple and very pure subset of the logic programming language Prolog. More specifically, we were inspired by the language Dedalus [1]. Declarative languages have many advantages:

– Declarative programs are usually shorter than an equivalent program in a procedural language. This enhances the productivity of the programmers.
– There can be no problems with uninitialized variables or memory leaks.

- The language is relatively simple, therefore it can be used also by non-experts (e.g., Arduino boards are a nice device to be used in school).
- The language has a mathematically precise semantics based on logic, which makes programs easier to verify.
- The simple semantics also permits powerful optimization, e.g. in [9], we translate a subclass of programs to a finite automaton extended with a fixed set of variables (i.e. we use "parameterized states").
- Many programs become easier to understand and more flexible by a data-driven architecture. E.g., the configuration data for a home-automation system used as an example in [9] is basically a small database.

One reason for the current revival of Datalog is that it is used also for applications that are not database applications, such as static analysis of program code [5], cloud computing, and semantic web applications.

While classic Datalog is not turing-complete, we use it on an infinite sequence of states (similar to the language Dedalus [1]). If one ignores technical restrictions such as the restricted memory and the finite range of integers we could get from the clock, it would be possible to simulate a turing machine. Therefore, using Datalog to specify the input-output-behaviour of a microcontroller is not restricted to toy applications.

We have a prototype implementation that compiles our Datalog-based language "Microlog" into C code for the Arduino IDE. In order to be sure that the program will never stop working, one has to prove that the memory is sufficient for storing all derived facts for the current and the next state. While restricted memory is in principle a problem for many programs, the danger of insufficient memory is quite real on this small hardware. The solution in our paper [9] is fully automatic, but works only for a restricted set of programs. In this paper, we follow a different path based on integrity constraints. They have to be specified manually, but if the set is sufficiently complete, the method presented here may be able to prove that the constraints hold in all states.

We call class of constraints studied in this paper "generalized exclusion constraints". Each instance expresses that two facts cannot occur together in a model. This includes key constraints: There, different facts with the same key value cannot both be true in the same state. Whereas keys are local to one relation, "generalized exclusion constraints" can be specified also for facts from different relations. Exclusion constraints appear already in [2,6]. They require that projections of two relations are disjoint: $\pi_{A_{i_1},\ldots,A_{i_n}}(R) \cap \pi_{B_{j_1},\ldots,B_{j_m}}(S) = \emptyset$. Of course, such constraints are also a special case of the constraints studied here.

In Section 2, we define a rule-based language for programming microcontrollers and its translation into pure Datalog. In Section 3, we define the type of constraints that is investigated in this paper, and show how they can be used for the task at hand. In Section 4, we give the tools to prove that the constraints are indeed satisfied for a given program. While we focus in this paper on microcontroller programming, the technique is applicable to any Datalog program. Therefore, it is an interesting alternative to our previous work on computing functional dependencies for derived predicates [3].

2 A Datalog-Variant for Microcontroller Systems

2.1 Standard Datalog

Let us first quickly repeat the definition of standard Datalog. The Datalog dialect for Arduino microcontroller systems will be translated to a standard Datalog program in order to define its semantics. Also the generalized exclusion constraints will be defined for standard Datalog, which makes them applicable for all applications of Datalog, not only in microcontroller programming.

A Datalog program is a finite set of rules of the form $A \leftarrow B_1 \wedge \cdots \wedge B_n$, where the head literal A and the body literals B_i are atomic formulas $p(t_1, \dots, t_m)$ with a predicate p and terms $t_1, \dots, t_m$. Terms are constants or variables. Rules must be range-restricted, i.e. all variables appearing in the head A must also appear in at least one body literal B_i. This ensures that all variables are bound to a value when the rule is applied. The requirement will be slightly modified for rules with built-in predicates, see below. A fact is a rule with an empty body, i.e. it has the form $p(c_1, \dots, c_m)$ with constants c_i.

In order to work with time, we need some built-in predicates for integers. Whereas normal predicates are defined by rules (or facts) as above, built-in predicates have a fixed semantics that is built into the system. They can only appear in rule bodies and have additional requirements for the range-restriction so that the rule body is evaluable at least in the sequence from left to right.

- $\mathsf{succ}(\mathsf{T}, \mathsf{S})$: This returns the next point in time (state) S for a given state T. Therefore, the variable T must appear in a literal to the left of this literal in the rule body so that it is bound when this literal is executed. We use $\mathbb{N}_0$ as logical time, and $\mathsf{succ}(\mathsf{T}, \mathsf{S})$ is true iff $\mathsf{S} = \mathsf{T} + 1 \ \wedge \ \mathsf{T} \geq 0$.
- $t_1 < t_2$, and the same with the other comparison operators $=, \neq, \leq, >, \geq$. If the terms t_1 or t_2 are variables, the variable must appear already in a body literal to the left (and therefore be bound to a value).
- $t_1 + t_2 < t_3$ which ensures that t_3 is more than t_2 time units (milliseconds) after t_1. Again variables must be bound to the left. The delay t_2 must be ≥ 0.
- $t_1 + t_2 \geq t_3$ (t_3 is not more than t_2 milliseconds after t_1, possibly before t_1).

2.2 Datalog with States

The program on a microcontroller must act in time. It basically runs forever (until the power is switched off), but the time-dependent inputs lead to some state change, and outputs depend on the state and also change over time. We do not assume knowledge about the outside world, but it is of course possible that the outputs influence future inputs. So it is quite clear that a programming language for microcontrollers must be able to define a sequence of states.

We borrow from Dedalus$_0$ [1] the idea to add a time (or state) argument to every predicate. Note that this is logical time, the numbers have no specific meaning except being a linear order.

Every predicate that looks like having n arguments really has $n+1$ arguments with an additional "zeroth" time argument in front. For a literal A of the form $p(t_1, \ldots, t_n)$ let $\hat{A}$ be $p(\mathsf{T}, t_1, \ldots, t_n)$ with a fixed special variable T that cannot be used directly in the program. For a normal rule $A \leftarrow B_1 \wedge \cdots \wedge B_m$, all time arguments are this same variable T, i.e. the rule describes a deduction within a state. Thus, the rule is an abbreviation for the standard Datalog rule $\hat{A} \leftarrow \hat{B}_1 \wedge \cdots \wedge \hat{B}_m$.

In order to define the next state, we also permit rules with the special mark "@next" in the head literal:

$$p(t_1, \ldots, t_n)@\mathsf{next} \leftarrow B_1 \wedge \cdots \wedge B_m.$$

This rule is internally replaced by:

$$p(\mathsf{S}, t_1, \ldots, t_n) \leftarrow \hat{B}_1 \wedge \cdots \wedge \hat{B}_m \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S}).$$

Note that @next can only be applied in the head, i.e. we can transfer information only forward in time. All conditions can only refer to the current point in time.

Facts can be marked with @start, in which case the constant 1 is inserted for the time argument, i.e. $p(c_1, \ldots, c_n)@\mathsf{start}$ is replaced by $p(1, c_1, \ldots, c_n)$. Since sometimes setup settings must be done before the main program can start, we also permit @init which uses the time constant 0. This pre-state is also necessary because the results of calls are only available in the next state. For instance, we will need the real-time as returned by `millis()` in every state. However, to be available in the start state 1, the function must be called in state 0.

Facts without this mark hold in all states (they are time-independent). However, since all predicates have a time argument, and we want rules to be range-restriced, we define a predicate always as

$$\begin{array}{ll} \mathsf{always}@\mathsf{init}. & \%\ \mathsf{always}(0). \\ \mathsf{always}@\mathsf{next} \leftarrow \mathsf{always}. & \%\ \mathsf{always}(\mathsf{S}) \leftarrow \mathsf{always}(\mathsf{T}) \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S}). \end{array}$$

A fact $p(c_1, \ldots, c_m)$ is replaced by $p(\mathsf{T}, c_1, \ldots, c_n) \leftarrow \mathsf{always}(\mathsf{T})$. (A possible optimization would be to compute time-independent predicates and remove the time argument from them.)

The minimal model of such a program is usually infinite (at least with `always` or similar predicates), therefore the iteration of the T_P-operator to compute derived facts does not stop. However, this is no real problem, since we actually compute derived facts state by state. We forbid direct access to the succ-relation and to the special variables T and S. Therefore, within a state, only finitely many facts are derivable. After we reached a fixpoint, we apply the rules with @next in the head to compute facts for the next state. When that is done, we can forget the facts in the old state, and switch to the new state. Within that state, we can again apply the normal rules to compute all facts true in that state.

2.3 Interface Predicates

Of course, a Datalog program for a Microcontroller must interface with the libraries for querying input devices and performing actions on output devices. A few examples of interface functions are shown in Fig. 1.

```
#define HIGH   0x1
#define LOW    0x0
#define INPUT  0x00
#define OUTPUT 0x01
void pinMode(uint8_t pin, uint8_t mode);
void digitalWrite(uint8_t pin, uint8_t val);
int  digitalRead(uint8_t pin);
unsigned long millis(void);
```

Fig. 1. Extract from `Arduino.h` Header Files

For each function f that can be called, there is a special predicate $\mathsf{call\_f}$ with a reserved prefix "$\mathsf{call\_}$". The predicate has the same arguments as the function to be called and of course the standard time argument. E.g. derived facts about the predicate $\mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val})$ lead to the corresponding calls of the interface faction `digitalWrite` in state T. The implementation ensures that duplicate calls are eliminated, i.e. even if there are different ways to deduce the fact, only one call is done.

The sequence of calls is undefined. If a specific sequence is required, one must use multiple states. Conflicts between functions (where a different order of calls has different effects) can be specified by means of our exclusion constraints.

If an interface function f returns a value, there is a second predicate $\mathsf{ret\_f}$ that contains all parameters of the call and a parameter for the return value. For instance, for the function `digitalRead`, there are two predicates:

- $\mathsf{call\_digitalRead}(\mathsf{T}, \mathsf{Port})$, and
- $\mathsf{ret\_digitalRead}(\mathsf{S}, \mathsf{Port}, \mathsf{Val})$.

If the call is done in one state, the result value is available in the next state. This ensures, e.g., that the occurrence of a call cannot depend on its own result.

Since calls of interface functions usually have side effects and cannot be taken back, it is important to clearly define which calls are actually done. In contrast, the evaluation sequence of literals in a rule body can be chosen by the optimizer. Therefore the special $\mathsf{call\_f}$ predicate can be used only in rule heads. We use the syntax $\mathsf{f}(t_1, \ldots, t_n)@\mathsf{call}$, which is translated to $\mathsf{call\_f}(t_{i_1}, \ldots, t_{i_k})$, where $i_1 < i_2 < \cdots < i_k$ are all arguments that are not the special marker ?. For instance, a rule that calls `digitalRead` is written as

$$\mathsf{digitalRead}(\mathsf{Port}, ?)@\mathsf{call} \leftarrow \ldots$$

It seems more consistent if the call and the result look like the same predicate with the same number of arguments. Correspondingly, $\mathsf{f}(t_1, \ldots, t_n)@\mathsf{ret}$ is replaced by $\mathsf{ret\_f}(t_1, \ldots, t_n)$. It can only appear in rule bodies.

For calls that should occur in the initialization state, the suffixes @call@init could be used together, but this does not look nice. We use @setup in this case.

Finally, we need also constants from the interface definition. If our Datalog program contains e.g. #`HIGH`, this corresponds to the constant `HIGH` in the generated C-code.

2.4 Real Time

So far, we have just a sequence of states. How much time it really takes from one state to the next depends on the necessary deductions in the state and the time needed for the interface function calls. Many control programs need real time. This can be achieved with the interface function `millis()` that returns the number of milliseconds since the program was started.

For common patterns of using real time information, we should define abbreviations. For instance, delaying a call to a predicate for a certain number of milliseconds can be written as follows:

$$p(t_1,\dots,t_n)@\mathsf{after}(\mathsf{Delay}) \leftarrow A_1,\dots,A_m.$$

This is internally translated to the following rules:

$$\begin{array}{l}
\mathsf{delayed_}p(t_1,\dots,t_n,\mathsf{From},\mathsf{Delay})@\mathsf{next} \leftarrow \\
\qquad A_1 \wedge \dots \wedge A_m \wedge \mathsf{millis}@\mathsf{ret}(\mathsf{From}). \\
\mathsf{delayed_}p(X_1,\dots,X_n,\mathsf{From},\mathsf{Delay})@\mathsf{next} \leftarrow \\
\qquad \mathsf{delayed_}p(X_1,\dots,X_n,\mathsf{From},\mathsf{Delay}) \wedge \\
\qquad \mathsf{millis}@\mathsf{ret}(\mathsf{Now}) \wedge \mathsf{From}+\mathsf{Delay} < \mathsf{Now}. \\
p(X_1,\dots,X_n)@\mathsf{next} \leftarrow \\
\qquad \mathsf{delayed_}p(X_1,\dots,X_n,\mathsf{From},\mathsf{Delay}) \wedge \\
\qquad \mathsf{millis}@\mathsf{ret}(\mathsf{Now}) \wedge \mathsf{From}+\mathsf{Delay} \geq \mathsf{Now}. \\
\mathsf{millis}(?)@\mathsf{call}.
\end{array}$$

The function `millis()` is called in every state so that there is always the current time available. Since we do not exactly know how long the processing for one state takes, we cannot be sure that we really get every milliseconds value. Therefore, the comparisons are done with $\leq$ and $>$ instead of $=$ and $\neq$.

Example 1. Most Arduino boards have an LED already connected to Port 13. With the following program we can let this LED blink with 1000 ms on, then 1000 ms off, and so on. The similar program `BinkWithoutDelay` from the Arduino tutorial has 16 lines of code.

$$\begin{array}{ll}
\mathsf{pinMode}(13,\#\mathtt{OUTPUT})@\mathsf{setup}. & \\
\mathsf{turn_on}@\mathsf{start}. & \\
\mathsf{turn_off}@\mathsf{after}(1000) & \leftarrow \mathsf{turn_on}. \\
\mathsf{turn_on}@\mathsf{after}(1000) & \leftarrow \mathsf{turn_off}. \\
\mathsf{digitalWrite}(13,\#\mathtt{HIGH})@\mathsf{call} & \leftarrow \mathsf{turn_on}. \\
\mathsf{digitalWrite}(13,\#\mathtt{LOW})@\mathsf{call} & \leftarrow \mathsf{turn_off}.
\end{array}$$

The internal Datalog version (with all abbreviations expanded) of the program is shown in Fig. 2. □

3 Generalized Exclusion Constraints

Obviously, it should be excluded that `digitalWrite` is called in the same state and for the same port with two different values. Since no specific sequence is

(1) $\mathsf{call\_pinMode}(0, 13, \#\mathtt{OUTPUT})$.

(2) $\mathsf{turn\_on}(1)$.

(3) $\mathsf{delayed\_turn\_off}(\mathsf{S}, \mathsf{From}, 1000) \leftarrow$
$\mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{ret\_millis}(\mathsf{T}, \mathsf{From}) \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(4) $\mathsf{delayed\_turn\_off}(\mathsf{S}, \mathsf{From}, \mathsf{Delay}) \leftarrow$
$\mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}) \wedge$
$\mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}) \wedge \mathsf{From} + \mathsf{Delay} < \mathsf{Now} \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(5) $\mathsf{turn\_off}(\mathsf{S}) \leftarrow$
$\mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}) \wedge$
$\mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}) \wedge \mathsf{From} + \mathsf{Delay} \geq \mathsf{Now} \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(6) $\mathsf{delayed\_turn\_on}(\mathsf{S}, \mathsf{From}, 1000) \leftarrow$
$\mathsf{turn\_off}(\mathsf{T}) \wedge \mathsf{ret\_millis}(\mathsf{T}, \mathsf{From}) \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(7) $\mathsf{delayed\_turn\_on}(\mathsf{S}, \mathsf{From}, \mathsf{Delay}) \leftarrow$
$\mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}) \wedge$
$\mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}) \wedge \mathsf{From} + \mathsf{Delay} < \mathsf{Now} \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(8) $\mathsf{turn\_on}(\mathsf{S}) \leftarrow$
$\mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}) \wedge$
$\mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}) \wedge \mathsf{From} + \mathsf{Delay} \geq \mathsf{Now} \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(9) $\mathsf{call\_digitalWrite}(\mathsf{T}, 13, \#\mathtt{HIGH}) \leftarrow$
$\mathsf{turn\_on}(\mathsf{T})$.

(10) $\mathsf{call\_digitalWrite}(\mathsf{T}, 13, \#\mathtt{LOW}) \leftarrow$
$\mathsf{turn\_off}(\mathsf{T})$.

(11) $\mathsf{always}(0)$.

(12) $\mathsf{always}(\mathsf{S}) \leftarrow \mathsf{always}(\mathsf{T}) \wedge \mathsf{succ}(\mathsf{T}, \mathsf{S})$.

(13) $\mathsf{call\_millis}(\mathsf{T}) \leftarrow \mathsf{always}(\mathsf{T})$.

Fig. 2. Blink Program from Example 1 with all appreviations expanded

defined for the calls, it is not clear whether the output will remain high or low (the last call overwrites the value set by the previous call). What is needed here is a key constraint. In this section, we consider only standard Datalog. Therefore, we must look at the translated/internal version of the example. There, the predicate is $\mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val})$, and we need that the first two arguments are a key for all derivable facts. In logic programming and deductive databases, constraints are often written as rule with an empty head (meaning "false"). Thus, a constraint rule like the following should never be applicable:

$$\leftarrow \mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val}_1) \wedge \mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val}_2) \wedge \mathsf{Val}_1 \neq \mathsf{Val}_2.$$

If we look at the program, we see that a violation of this key could only happen if turn_on and turn_off would both be true in the same state. Thus, we need also this constraint:

$$\leftarrow \mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{turn\_off}(\mathsf{T}).$$

The common pattern is that there are conflicts between two literals, such that the existence of a fact that matches one literal excludes all instances of the other literal. This leads to the following definition:

Definition 1 (Generalized Exclusion Constraint). *A "Generalized Exclusion Constraint" (GEC) is a formula of the form*

$$\leftarrow p(t_1, \ldots, t_n) \wedge q(u_1, \ldots, u_m) \wedge \varphi$$

and φ *is either* true *or a disjunction of inequalities* $t_{i_\nu} \neq u_{j_\nu}$ *for* $\nu = 1, \ldots, k$.

The implicit head of the rule is false, so the constraint is satisfied in a Herbrand interpretation $\mathcal{I}$ iff there is no ground substitution θ for the two body literals such that $p(t_1, \ldots, t_n)\theta \in \mathcal{I}$ and $q(u_1, \ldots, u_m)\theta \in \mathcal{I}$ and φ is true or there is $\nu \in \{1, \ldots, k\}$ with $t_{i_\nu}\theta \neq u_{j_\nu}\theta$.

Example 2. The "generalized exclusion constraints" are really a generalization of the exclusion constraints of [2,6]: For instance, consider relations $\mathsf{r}(A, B)$ and $\mathsf{s}(A, B, C)$ and the exclusion constraint $\pi_A(\mathsf{r}) \cap \pi_A(\mathsf{s}) = \emptyset$. In our formalism, this would be expressed as $\leftarrow \mathsf{r}(A, \_) \wedge \mathsf{s}(A, \_, \_) \wedge \mathsf{true}$.

As in Prolog, every occurence of "_" denotes a new variable (a placeholder for unused arguments). It is a violation of the constraint if the same value A appears as first argument of r and as first argument of S. □

In the following, when we say simply "exclusion constraint" or even "constraint", we mean "generalized exclusion constraint". We also allow to drop "$\wedge\, \mathsf{true}$" in the constraint formula.

Example 3. We already illustrated with digitalWrite above that our constraints can express keys. We can also express any functional dependency. For instance, consider $\mathsf{r}(A, B, C)$ and the FD $B \longrightarrow C$. This is the same as the generalized exclusion constraint $\leftarrow \mathsf{r}(\_, B, C_1) \wedge \mathsf{r}(\_, B, C_2) \wedge C_1 \neq C_2$. □

Example 4. For the original task, to check that memory is sufficient to represent all facts in a single state, we need in particular the following constraint:

$$\leftarrow \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_1, \mathsf{Delay}_1) \wedge \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_2, \mathsf{Delay}_2) \wedge (\mathsf{From}_1 \neq \mathsf{From}_2 \vee \mathsf{Delay}_1 \neq \mathsf{Delay}_2).$$

This is actually a key constraint and means each state contains at most one delayed_turn_on-fact. Of course, we need the same for delayed_turn_off. With that, the potentially unbounded set of facts in a state already becomes quite small. The implicit state argument is no problem, because we compute only facts for the current state and for the next state. Also arguments filled with constants in the program cannot lead to multiple facts in the state. Furthermore, function calls have unique results, i.e. the functional property holds. E.g. constraints like the following for the `millis()` function can be automatically generated:

$$\leftarrow \mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}_1) \wedge \mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}_2) \wedge \mathsf{Now}_1 \neq \mathsf{Now}_2.$$

With these constraints, we already know that a state for the Blink program can contain at most one fact of each predicate. This certainly fits in memory.

The full set of constraints for the Blink program from Example 1 is shown in Fig. 3. Five of the constraints are keys, but (C) to (H) state that no two of the

(A) $\leftarrow \mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val}_1) \wedge \mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val}_2) \wedge \mathsf{Val}_1 \neq \mathsf{Val}_2.$

(B) $\leftarrow \mathsf{call\_pinMode}(\mathsf{T}, \mathsf{Port}, \mathsf{Mode}_1) \wedge \mathsf{call\_pinMode}(\mathsf{T}, \mathsf{Port}, \mathsf{Mode}_2) \wedge \mathsf{Mode}_1 \neq \mathsf{Mode}_2.$

(C) $\leftarrow \mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{turn\_off}(\mathsf{T}).$

(D) $\leftarrow \mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}).$

(E) $\leftarrow \mathsf{turn\_off}(\mathsf{T}) \wedge \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}).$

(F) $\leftarrow \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_1, \mathsf{Delay}_1) \wedge \mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}_2, \mathsf{Delay}_2).$

(G) $\leftarrow \mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}).$

(H) $\leftarrow \mathsf{turn\_off}(\mathsf{T}) \wedge \mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}, \mathsf{Delay}).$

(I) $\leftarrow \mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}_1, \mathsf{Delay}_1) \wedge \mathsf{delayed\_turn\_off}(\mathsf{T}, \mathsf{From}_2, \mathsf{Delay}_2) \wedge (\mathsf{From}_1 \neq \mathsf{From}_2 \vee \mathsf{Delay}_1 \neq \mathsf{Delay}_2).$

(J) $\leftarrow \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_1, \mathsf{Delay}_1) \wedge \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_2, \mathsf{Delay}_2) \wedge (\mathsf{From}_1 \neq \mathsf{From}_2 \vee \mathsf{Delay}_1 \neq \mathsf{Delay}_2).$

(K) $\leftarrow \mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}_1) \wedge \mathsf{ret\_millis}(\mathsf{T}, \mathsf{Now}_2) \wedge \mathsf{Now}_1 \neq \mathsf{Now}_2.$

Fig. 3. Constraints for the Blink Program

predicates turn_on, turn_off, delayed_turn_on, delayed_turn_off occur in the same state. There should be an abbreviation for such a constraint set: “For every T, at most one instance of turn_on(T), turn_off(T), delayed_turn_on(T, From, Delay), delayed_turn_off(T, From, Delay) is true.” This includes also the keys (I) and (J). The keys (A) and (B) could come from a library, and keys of type (K) should be automatic for all ret_f predicates. □

4 Refuting Violation Conditions

4.1 Violation Conditions

A “violation condition” describes the situation where two rule applications lead to facts that violate a constraint. Our task will be to show that all violation conditions themselves violate a constraint or are otherwise inconsistent or impossible to occur. Basically, we get from a constraint rule to a violation condition if we do an SLD resolution step (corresponding to unfolding) on each literal:

Definition 2 (Violation Condition). *Let a Datalog program P and a generalized exclusion constraint $\leftarrow A_1 \wedge A_2 \wedge \varphi$ be given. Let*

– $A'_1 \leftarrow B_1 \wedge \cdots \wedge B_m$ *($m \geq 0$) be a variant with fresh variables of a rule in P,*
– $A'_2 \leftarrow C_1 \wedge \cdots \wedge C_n$ *($n \geq 0$) be a variant with fresh variables of a rule in P (it might be the same or a different rule), such that*
– (A_1, A_2) *is unifiable with* (A'_1, A'_2)*. Let θ be a most general unifier.*

Then the violation condition is:

$$(B_1 \wedge \cdots \wedge B_m \wedge C_1 \wedge \cdots \wedge C_n \wedge \varphi)\theta.$$

The "fresh variables" requirement means that the variables are renamed so that the constraint and the two rules have pairwise disjoint variables.

The disjunction $\varphi\theta$ can be simplified by removing inequalities $t_i \neq u_i$ that are certainly false, because t_i and u_i are the same variable or the same constant. If the disjunction becomes empty in this way, it is false, and we do not have to consider the violation condition futher. If t_i and u_i are distinct constants for some i, the inequality and thus the whole disjunction can be simplified to true.

Example 5. Consider constraint (A), the key constraint for call_digitalWrite:

$$\leftarrow \mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val}_1) \wedge \mathsf{call\_digitalWrite}(\mathsf{T}, \mathsf{Port}, \mathsf{Val}_2) \wedge \mathsf{Val}_1 \neq \mathsf{Val}_2.$$

The two rules with matching head literals are rules (9) and (10):

$$\begin{array}{ll} \mathsf{call\_digitalWrite}(\mathsf{T}, 13, \#\mathtt{HIGH}) & \leftarrow \mathsf{turn\_on}(\mathsf{T}). \\ \mathsf{call\_digitalWrite}(\mathsf{T}, 13, \#\mathtt{LOW}) & \leftarrow \mathsf{turn\_off}(\mathsf{T}). \end{array}$$

We rename the variables of the rules so that the constraint and the two rules have pairwise disjoint variables (we start with index 3, since 1 and 2 appear in the constraint):

$$\begin{array}{ll} \mathsf{call\_digitalWrite}(\mathsf{T}_3, 13, \#\mathtt{HIGH}) & \leftarrow \mathsf{turn\_on}(\mathsf{T}_3). \\ \mathsf{call\_digitalWrite}(\mathsf{T}_4, 13, \#\mathtt{LOW}) & \leftarrow \mathsf{turn\_off}(\mathsf{T}_4). \end{array}$$

Now we do the unification of the head literals with the literals from the constraint. A possible most general unifier (MGU) is

$$\{\mathsf{T}_3/\mathsf{T}, \mathsf{T}_4/\mathsf{T}, \mathsf{Port}/13, \mathsf{Val}_1/\#\mathtt{HIGH}, \mathsf{Val}_2/\#\mathtt{LOW}\}.$$

MGUs are unique modulo a variable renaming. Now the violation condition is

$$\mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{turn\_off}(\mathsf{T}) \wedge \#\mathtt{HIGH} \neq \#\mathtt{LOW}.$$

Since $\#\mathtt{HIGH} \neq \#\mathtt{LOW}$ is true, the violation condition can be simplified to

$$\mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{turn\_off}(\mathsf{T}).$$

This is what we would expect: It should never happen that turn_on and turn_off are true in the same state.

It would also be possible to match the two literals of the constraint with different variants (with renamed variables) of the same rule, but in this example, that would give conditions like $\#\mathtt{HIGH} \neq \#\mathtt{HIGH}$, which are false. Such obviously inconsistent violation conditions do not have to be considered. □

Violation conditions express the conditions under which the result of a derivation step violates an exclusion constraint:

Theorem 1. $\mathsf{T}_P(\mathcal{I})$ *violates an exclusion constraint* $\leftarrow A_1 \wedge A_2 \wedge \varphi$ *if and only if there is a violation condition for* P *and* $\leftarrow A_1 \wedge A_2 \wedge \varphi$ *which is true in* $\mathcal{I}$.

The T_P operator, well known in logic programming, yields all facts that can be derived by a single application of the rules in P, given the facts that are true in the input interpretation. One starts with the empty set of facts $\mathcal{I}_0$ which certainly satisfies all exclusion constraints. Then one iteratively applies the T_P operator, i.e. $\mathcal{I}_{i+1} := \mathsf{T}_P(\mathcal{I}_i)$, to get the minimal model $\mathcal{I}_\omega := \bigcup_{i \in \mathbb{N}} \mathcal{I}_i$, which is the intended interpretation of P.

Theorem 2. *Let* P *be a Datalog program,* $\mathcal{C}$ *be a set of generalized exclusion constraints, and* $\mathcal{H}$ *be some set of Herbrand interpretations that includes at least all interpretations that occur in the iterative computation of the minimal model. If all violation conditions for* P *and constraints from* $\mathcal{C}$ *are false in all* $\mathcal{I} \in \mathcal{H}$ *that satisfy* $\mathcal{C}$*, then the minimal Herbrand model* $\mathcal{I}_\omega$ *of* P *satisfies* $\mathcal{C}$.

Thus, we have to show that the violation conditions are unsatisfiable assuming the constraints. However, it turns out that this does not work well in the initialization state 0 and the start state 1. Therefore, the theorem permits to throw in additional knowledge formalized as some set of Herbrand interpretations $\mathcal{H}$ that is a superset of the relevant interpretations. In the example, we need that ret_f-predicates cannot occur in state 0: This is obvious, because there is no previous state that might contain a call. One could also precompute all predicates that might occur in state 0 and 1 and use this knowledge to restrict $\mathcal{H}$.

4.2 Proving Violation Conditions Inconsistent

The consistency check for the violation conditions is done by transforming the task to a formula that can be checked by a constraint solver for linear arithmetic constraints [7,4]. Since we assume that all constraints were satisfied before the derivation step that is described by the violation condition, we can exclude any match of two literals A_1 and A_2 from the violation condition with a constraint:

Definition 3 (Match Condition). *Let two literals* A_1 *and* A_2 *and an exclusion constraint* $\leftarrow C_1 \wedge C_2 \wedge \gamma$ *be given. Let* $\leftarrow C_1' \wedge C_2' \wedge \gamma'$ *be a variant of the constraint with fresh variables (not occurring in* A_1 *and* A_2*). If* (A_1, A_2) *are unifiable with* (C_1', C_2') *there is a match condition for* (A_1, A_2) *and this constraint, computed as follows:*

- *Let* θ *be a most general unifier without variable-to-variable bindings from variables of* (A_1, A_2) *to variables of* (C_1', C_2') *(since the direction of variable-to-variable bindings is arbitrary, this is always possible).*
- *Let* A_1 *be* $p(t_1, \ldots, t_n)$ *and* A_2 *be* $q(u_1, \ldots, u_m)$.
- *Then the match condition is*

$$t_1 = t_1\theta \wedge \cdots \wedge t_n = t_n\theta \wedge u_1 = u_1\theta \wedge \cdots \wedge u_m = u_m\theta \wedge \gamma'\theta.$$

The requirement on the direction of variable-to-variable bindings ensures that the match condition contains only variables that also occur in A_1 or A_2.

Again, some parts of the condition can be immediately evaluated. The formula basically corresponds to the unification (plus the formula from the constraint). Most conditions will have the form $X = X$ and can be eliminated. However, if the literals from the constraint contain constants or equal variables, the condition becomes interesting. Note that we cannot simply apply the unification as in Definition 2, because we finally need to negate the condition: We are interested in values for the variables that are possible without violating the exclusion constraint.

Definition 4 (Violation Formula). *Let a violation condition*

$$A_1 \wedge \cdots \wedge A_m \wedge B_1 \wedge \cdots \wedge B_n \wedge \varphi$$

be given, where $A_1, \ldots, A_m$ have user-defined predicates and $B_1, \ldots, B_m$ have built-in predicates. The violation formula for this violation condition is a conjunction ($\bigwedge$) of the following parts:

- φ
- *For each B_i its logical definition. If B_i has the form* $\mathsf{succ}(t_1, t_2)$*, the logical definition is $t_2 = t_1 + 1 \wedge t_1 \geq 0$. For $t_1 + t_2 \geq t_3$ and $t_1 + t_2 < t_3$, we take that and add $t_2 \geq 0$. For other built-in predicates, it is B_i itself.*
- *For all possible match conditions μ of a constraint $\leftarrow C_1 \wedge C_2 \wedge \gamma$ with two literals A_i and A_j, the negation $\neg\mu$.*

Example 6. This example continues Example 5 with the violation condition:

$$\mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{turn\_off}(\mathsf{T}).$$

We use constraint (C): $\leftarrow \mathsf{turn\_on}(\mathsf{T}) \wedge \mathsf{turn\_off}(\mathsf{T})$. Formally, we have to rename the variable in the constraint, e.g. to T_1, and then compute the unifier T_1/T of the constraint literals with the literals in the violation condition. The match condition is $\mathsf{T} = \mathsf{T} \wedge \mathsf{T} = \mathsf{T}$, which can be simplified to true. Since there is no other matching constraint, the violation formula, which requires that the violation condition does not violate the constraint, is $\neg\mathsf{true}$, i.e. false. Therefore, the violation condition cannot be satisfied. Of course, as soon as we know that the violation formula is unsatisfiable, we can stop. Thus, even if there were other matching constraints, we would not have to consider them. □

Example 7. For a more complex case, let us consider Constraint (J) which ensures that there can be only one fact about $\mathsf{delayed\_turn\_on}$ in each state:

$$\begin{aligned}\leftarrow\ & \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_1, \mathsf{Delay}_1) \wedge \mathsf{delayed\_turn\_on}(\mathsf{T}, \mathsf{From}_2, \mathsf{Delay}_2) \wedge \\ & (\mathsf{From}_1 \neq \mathsf{From}_2 \vee \mathsf{Delay}_1 \neq \mathsf{Delay}_2).\end{aligned}$$

In order to generate violation conditions, all possibilities for matching rule heads with the two literals of the constraint must be considered. In this case, facts that might violate the constraint can be derived by applying Rule (6) and Rule (7) (see Fig. 2). For space reasons, we consider only the violation condition that corresponds to the case that both constraint literals are derived with different instances of Rule (6): We rename the variables once to $\mathsf{S}_3, \mathsf{From}_3, \mathsf{T}_3$ and once to $\mathsf{S}_4, \mathsf{From}_4, \mathsf{T}_4$. An MGU is

$$\{\mathsf{S}_3/\mathsf{T}, \mathsf{From}_3/\mathsf{From}_1, \mathsf{Delay}_1/1000, \mathsf{S}_4/\mathsf{T}, \mathsf{From}_4/\mathsf{From}_2, \mathsf{Delay}_2/1000\}.$$

Thus, the resulting violation condition is:

$$\begin{aligned}&\mathsf{turn\_off}(\mathsf{T}_3) \wedge \mathsf{ret\_millis}(\mathsf{T}_3, \mathsf{From}_1) \wedge \mathsf{succ}(\mathsf{T}_3, \mathsf{T}) \wedge \\ &\mathsf{turn\_off}(\mathsf{T}_4) \wedge \mathsf{ret\_millis}(\mathsf{T}_4, \mathsf{From}_2) \wedge \mathsf{succ}(\mathsf{T}_4, \mathsf{T}) \wedge \\ &(\mathsf{From}_1 \neq \mathsf{From}_2 \vee 1000 \neq 1000)\end{aligned}$$

Of course, $1000 \neq 1000$ is false and can be removed. Now we want to compute the violation formula. The easy parts are:

- The formula part of the violation condition: $\mathsf{From}_1 \neq \mathsf{From}_2$.
- The definition of the built-in succ-literals:

  $$\mathsf{T} = \mathsf{T}_3 + 1 \wedge \mathsf{T}_3 \geq 0 \wedge \mathsf{T}_4 = \mathsf{T} + 1 \wedge \mathsf{T}_4 \geq 0.$$

  Note that $\mathsf{T}_3 = \mathsf{T}_4$ can be derived from this.

Furthermore, we have to add the negation of all possible match conditions for constraints matching two literals in the violation condition (we might stop early as soon as we have detected the inconsistency). In this case, there is only one possible constraint, namely (K). A variant with fresh variables is:

$$\leftarrow \mathsf{ret\_millis}(\mathsf{T}_5, \mathsf{Now}_5) \wedge \mathsf{ret\_millis}(\mathsf{T}_5, \mathsf{Now}_6) \wedge (\mathsf{Now}_5 \neq \mathsf{Now}_6)$$

An MGU with variable-to-variable bindings directed towards the violation condition is $\{\mathsf{T}_5/\mathsf{T}_3, \mathsf{Now}_5/\mathsf{From}_1, \mathsf{T}_4/\mathsf{T}_3, \mathsf{Now}_6/\mathsf{From}_2\}$. This gives the following match condition:

$$\mathsf{T}_3 = \mathsf{T}_3 \wedge \mathsf{From}_1 = \mathsf{From}_1 \wedge \mathsf{T}_4 = \mathsf{T}_3 \wedge \mathsf{From}_2 = \mathsf{From}_2 \wedge \mathsf{From}_1 \neq \mathsf{From}_2.$$

With the trivial equalities removed, this is $\mathsf{T}_4 = \mathsf{T}_3 \wedge \mathsf{From}_1 \neq \mathsf{From}_2$. The negation is added to the violation formula. Thus the total violation formula is:

$$\begin{aligned}&\mathsf{From}_1 \neq \mathsf{From}_2 \wedge \\ &\mathsf{T} = \mathsf{T}_3 + 1 \wedge \mathsf{T}_3 \geq 0 \wedge \mathsf{T}_4 = \mathsf{T} + 1 \wedge \mathsf{T}_4 \geq 0 \wedge \\ &\neg(\mathsf{T}_4 = \mathsf{T}_3 \wedge \mathsf{From}_1 \neq \mathsf{From}_2).\end{aligned}$$

This is easily discovered to be inconsistent. Thus, Constraint (J) cannot be violated if both literals are derived with Rule (6). The other cases can be handled in a similar way. □

Theorem 3. *Let $A_1 \wedge \cdots \wedge A_m \wedge B_1 \wedge \cdots \wedge B_m \wedge \varphi$ be a violation condition and ψ be its violation formula with respect to constraints $\mathcal{C}$. There is a variable assignment $\mathcal{A}$ that makes ψ true in the standard interpretation of arithmetics if and only if there is a Herbrand interpretation $\mathcal{I}$ satisfying $\mathcal{C}$ with the standard interpretation of the built-in predicates such that the violation condition is true in $\mathcal{I}$ for some extension of $\mathcal{A}$ (not all variables of the violation condition might be in ψ).*

5 Conclusions

We are investigating the programming of microcontrollers in Datalog. We have discussed an interesting class of constraints which we called "generalized exclusion constraints". They contain keys, but can specify uniqueness of facts also between different relations. In particular, the constraints can be used to ensure that each state does not contain "too many" facts, e.g. more than what fits in the restricted memory of a microcontroller. But they also can express conflicts between different interface functions that cannot be called in the same state.

This class of constraints is also interesting, because for the most part, they are able to reproduce themselves during deduction. We have introduced the notion of a "violation condition" as a tool for checking this. Violation conditions can be reduced to a "violation formula" that can be checked for consistency by a constraint solver for linear arithmetics. If the violation formula should be consistent, the violation condition can be shown to the user who might then add a constraint to prove that the violation can never occur. A prototype implementation is available at: `https://users.informatik.uni-halle.de/~brass/micrologS/`.

References

1. Alvaro, P., Marczak, W.R., Conway, N., Hellerstein, J.M., Maier, D., Sears, R.: Dedalus: Datalog in time and space. In: de Moor, O., Gottlob, G., Furche, T., Sellers, A.J. (eds.) Datalog Reloaded — First International Workshop, Datalog 2010. LNCS, vol. 6702, pp. 262–281. Springer (2011)
2. Casanova, M.A., Vidal, V.M.P.: Towards a sound view integration methodology. In: Proc. of the 2nd ACM SIGACT-SIGMOD Symposium on Principles of Database Systems (PODS'83). pp. 36–47 (1983)
3. Engels, C., Behrend, A., Brass, S.: A rule-based approach to analyzing database schema objects with Datalog. In: Logic-Based Program Synthesis and Transformation — 27th International Symposium (LOPSTR'17). pp. 20–36. No. 10855 in LNCS, Springer (2018)
4. Imbert, Jean-Louis, C.J., Weeger, M.D.: An algorithm for linear constraint solving: Its incorporation in a Prolog meta-interpreter for clp. The Journal of Logic Programming 16, 235–253 (1993), `https://core.ac.uk/download/pdf/82420821.pdf`
5. Scholz, B., Jordan, H., Subotić, P., Westmann, T.: On fast large-scale program analysis in Datalog. In: Proceedings of the 25th International Conference on Compiler Construction (CC'2016). pp. 196–206. ACM (2016)
6. Thalheim, B.: Dependencies in Relational Databases. Teubner (1991)
7. Van Hentenryck, P., Graf, T.: Standard forms for rational linear arithmetic in constraint logic programming. Annals of Mathematics and Artificial Intelligence 5(2), 303–319 (1992)
8. Wenzel, M., Brass, S.: Declarative programming for microcontrollers — Datalog on Arduino. In: Hofstedt, P., Abreu, S., John, U., Kuchen, H., Seipel, D. (eds.) Declarative Programming and Knowledge Management (DECLARE 2019). LNCS, vol. 12057, pp. 119–138. Springer (2020), `https://arxiv.org/abs/1909.00043`
9. Wenzel, M., Brass, S.: Translation of interactive datalog programs for microcontrollers to finite state machines. In: Fernández, M. (ed.) Logic-Based Program Synthesis and Transformation, 30th International Symposium (LOPSTR 2020). LNCS, vol. 12561, pp. 210–227. Springer (2020)