Abstract. We consider microcontroller-programming with a declarative language based on the logic-programming language Datalog. Many microcontrollers have very limited memory, e.g. the Atmel ATmega328P that is used in the standard Arduino board today has 2 KBytes RAM. Our prototype implementation translates a Datalog dialect to C-code for the Arduino IDE. However, in order to prove the correctness of the implementation, it is necessary to ensure that the memory is sufficient for the derived facts. In this paper, we propose a class of constraints called “generalized exclusion constraints”, and show how they can be used for this task. Moreover, they are needed to exclude conflicting commands to the hardware, e.g. that different values are output on the same pin in the same state. This class of constraints also generalizes keys and functional dependencies, therefore our results also help to prove that such constraints hold for derived predicates.

1 Introduction

A microcontroller is a small computer on a single chip. For instance, the Atmel 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, such as household appliances. Cars usually contain dozens of microcontrollers. Nearly 30 billion units are sold per year. In contrast, about 260 million PCs are sold per year.

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. This includes a large library of interface functions for many different input and output devices that can be connected to the board. In 2013, David Cuartielles (co-founder of Arduino) stated in an FAQ that 700.000 official Arduino boards were sold, but they estimated that there is at least one clone board per official board.

The software for microcontrollers is often developed in Assembler or versions of C. There basically is no operating system (there might be a boot-loader to simplify programming). Debugging can be difficult.
We believe that declarative programming can be an interesting option even for such small devices. In [10], 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.
- Declarative programs are not bound to a specific execution model or hardware technology, therefore they can make use of quite different hardware.
- 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.

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 [6], cloud computing [7], and semantic web applications [2].

In the current paper, we present a simplified variant Microlog of our language. There is a prototype implementation (of the original Microlog language) that compiles into C code for the Arduino IDE. However, currently, we cannot guarantee that the very restricted memory of the microcontroller is sufficient for storing all necessary derived facts. While restricted memory is in principle a problem for many programs, the danger of insufficient memory is quite real on this small hardware. Furthermore, in embedded systems, we cannot simply print an error message and hope that some administrator solves the problem. When the board is in real use, it should not stop working. It must be provably correct.

While there are different approaches to this task, in this paper we use a type of constraints that we call “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 [3,8]. They require that projections of two relations are disjoint: $\pi_{A_{i_1},\ldots,A_{i_m}}(R) \cap \pi_{B_{j_1},\ldots,B_{j_m}}(S) = \emptyset$. Such constraints are also a special case of the generalized exclusion constraints we study here.

In Section 2, we define a rule-based language for programming microcontrollers and its translation into pure Datalog. The semantics is easier than our previous proposal [10], and the usability seems basically the same. 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 [4].
2 A Datalog-Variant for Microcontroller Systems

2.1 Standard Datalog

Let us first quickly repeat 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 \( A \leftarrow B_1 \land \cdots \land B_n \), where the head literal \( A \) and the body literals \( B_i \) are atomic formulas \( p(t_1, \ldots, t_m) \) with a predicate \( p \) and terms \( t_1, \ldots, t_m \). 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 \( p(c_1, \ldots, c_m) \) 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.

- \( \text{succ}(T, S) \): 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 \( \text{succ}(T, S) \) is true iff \( S = T + 1 \land T \geq 0 \).
- \( t_1 < t_2 \), 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 \( \geq 0 \).
- \( t_1 + t_2 \geq t_3 \) (\( t_3 \) is at most \( t_2 \) milliseconds after \( t_1 \)).

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\textsuperscript{0} \cite{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 $p(T, t_1, \ldots, t_n)$ with a fixed special variable $T$ that cannot be used directly in the program. For a normal rule $A \leftarrow B_1 \land \cdots \land B_m$, 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 $\hat{A} \leftarrow \hat{B}_1 \land \cdots \land \hat{B}_m$.

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

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

This rule is internally replaced by:

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

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. $p(c_1, \ldots, c_m) \text{@start}$ 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-restricted, we define a predicate always as

$$\text{always}\text{@init.} \quad \% \text{always}(0).$$

$$\text{always}\text{@next} \leftarrow \text{always.} \quad \% \text{always}(S) \leftarrow \text{always}(T) \land \text{succ}(T, S).$$

A fact $p(c_1, \ldots, c_m)$ is replaced by $p(T, c_1, \ldots, c_n) \leftarrow \text{always}(T)$. (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.
For each function \( f \) that can be called, there is a special predicate \( \text{call}_f \) with a reserved prefix “\text{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 \( \text{call}_{\text{digitalWrite}}(T, \text{Port}, \text{Val}) \) lead to the corresponding calls of the interface function \( \text{digitalWrite} \) 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 \( \text{ret}_f \) that contains all parameters of the call and a parameter for the return value. For instance, for the function \( \text{digitalRead} \), there are two predicates:

\[- \text{call}_{\text{digitalRead}}(T, \text{Port}), \text{and}
\- \text{ret}_{\text{digitalRead}}(S, \text{Port}, \text{Val}).\]

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 \( \text{call}_f \) predicate can be used only in rule heads. We use the syntax \( f(t_1, \ldots, t_n)@\text{call} \), which is translated to \( \text{call}_f(t_{i_1}, \ldots, t_{i_k}) \), where \( i_1 < i_2 < \cdots < i_k \) are all arguments that are not the special marker \( ? \). For instance, a rule that calls \( \text{digitalRead} \) is written as

\[\text{digitalRead}(\text{Port}, ?)@\text{call} \leftarrow \ldots\]

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

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

Finally, we need also constants from the interface definition. If our Datalog program contains e.g. \#\text{HIGH}, this corresponds to the constant \text{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:

\[ p(t_1, \ldots, t_n)@after(Delay) \leftarrow A_1, \ldots, A_m. \]

This is internally translated to the following rules:

\[
\begin{align*}
\text{delayed}_p(t_1, \ldots, t_n, \text{From}, \text{Delay}@next & \leftarrow \\
& A_1 \land \cdots \land A_m \land \\
& \text{millis}@\text{ret}(\text{From}).
\end{align*}
\]

\[
\begin{align*}
\text{delayed}_p(X_1, \ldots, X_n, \text{From}, \text{Delay}@next & \leftarrow \\
& \text{delayed}_p(X_1, \ldots, X_n, \text{From}, \text{Delay}) \land \\
& \text{millis}@\text{ret}(\text{Now}) \land \\
& \text{From} + \text{Delay} < \text{Now}.
\end{align*}
\]

\[
\begin{align*}
p(X_1, \ldots, X_n}@next & \leftarrow \\
& \text{delayed}_p(X_1, \ldots, X_n, \text{From}, \text{Delay}) \land \\
& \text{millis}@\text{ret}(\text{Now}) \land \\
& \text{From} + \text{Delay} \geq \text{Now}.
\end{align*}
\]

\[ \text{millis}(?)@\text{call}. \]

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.

\[
\begin{align*}
\text{pinMode}(13, \#\text{OUTPUT}@setup. \\
\text{turn_on}@\text{start.} \\
\text{turn_off}@after(1000) & \leftarrow \text{turn_on.} \\
\text{turn_on}@after(1000) & \leftarrow \text{turn_off.} \\
\text{digitalWrite}(13, \#\text{HIGH}@\text{call} & \leftarrow \text{turn_on.} \\
\text{digitalWrite}(13, \#\text{LOW}@\text{call} & \leftarrow \text{turn_off.}
\end{align*}
\]

The internal Datalog version (with all abbreviations expanded) of the program is shown in Fig. 2. \(\square\)
(1) call_pinMode(0, 13, #OUTPUT).
(2) turn_on(1).
(3) delayed_turn_off(S, From, 1000) ←
    turn_on(T) ∧ ret_millis(T, From) ∧ succ(T, S).
(4) delayed_turn_off(S, From, Delay) ←
    delayed_turn_off(T, From, Delay) ∧
    ret_millis(T, Now) ∧ From + Delay < Now ∧ succ(T, S).
(5) turn_off(S) ←
    delayed_turn_off(T, From, Delay) ∧
    ret_millis(T, Now) ∧ From + Delay ≥ Now ∧ succ(T, S).
(6) delayed_turn_on(S, From, 1000) ←
    turn_off(T) ∧ ret_millis(T, From) ∧ succ(T, S).
(7) delayed_turn_on(S, From, Delay) ←
    delayed_turn_on(T, From, Delay) ∧
    ret_millis(T, Now) ∧ From + Delay < Now ∧ succ(T, S).
(8) turn_on(S) ←
    delayed_turn_on(T, From, Delay) ∧
    ret_millis(T, Now) ∧ From + Delay ≥ Now ∧ succ(T, S).
(9) call_digitalWrite(T, 13, #HIGH) ←
    turn_on(T).
(10) call_digitalWrite(T, 13, #LOW) ←
    turn_off(T).
(11) always(0).
(12) always(S) ← always(T) ∧ succ(T, S).
(13) call_millis(T) ← always(T).

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

3 Generalized Exclusion Constraints

Obviously, it should be excluded that call_digitalWrite is called in the same state
and for the same port with two different values. Since no specific sequence is
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 call_digitalWrite(T, Port, 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:

← call_digitalWrite(T, Port, Val1) ∧ call_digitalWrite(T, Port, Val2) ∧ Val1 ≠ Val2.

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:

\[ \leftarrow \text{turn_on}(T) \land \text{turn_off}(T). \]

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

\[ \leftarrow p(t_1, \ldots, t_n) \land q(u_1, \ldots, u_m) \land \varphi \]

and \( \varphi \) is either true or a disjunction of disequalities \( t_i \neq u_j \) for \( \nu = 1, \ldots, k \).

The implicit head of the rule is false, so the constraint is satisfied in a Her-
brand interpretation \( I \) iff there is no ground substitution \( \theta \) for the two body
literals such that \( p(t_1, \ldots, t_n) \theta \in I \) and \( q(u_1, \ldots, u_m) \theta \in I \) and \( \varphi \) is true or
there is \( \nu \in \{1, \ldots, k\} \) with \( t_i \theta \neq u_j \theta \).

**Example 2.** The “generalized exclusion constraints” are really a generalization
of the exclusion constraints of [3,8]: For instance, consider relations
\( r(A,B) \) and \( s(A,B,C) \) and the exclusion constraint \( \pi_A(r) \cap \pi_A(s) = \emptyset \). In our formalism,
this would be expressed as \( \leftarrow r(A,_,_) \land s(A,_,_) \).

As in Prolog, every occurrence 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 “con-
straint”, we mean “generalized exclusion constraint”.

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

**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:

\[ \leftarrow \text{delayed_turn_on}(T, \text{From}_1, \text{Delay}_1) \land \text{delayed_turn_on}(T, \text{From}_2, \text{Delay}_2) \land \left( \text{From}_1 \neq \text{From}_2 \lor \text{Delay}_1 \neq \text{Delay}_2 \right). \]

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, we must
know that function calls have unique results, i.e. the functional property holds.
In the example, only the `millis()` function returns a result:

\[ \leftarrow \text{ret_millis}(T, \text{Now}_1) \land \text{ret_millis}(T, \text{Now}_2) \land \text{Now}_1 \neq \text{Now}_2. \]
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

\[(A) \leftarrow \text{call} \_ \text{digitalWrite}(T, \text{Port}, \text{Val}_1) \land \text{call} \_ \text{digitalWrite}(T, \text{Port}, \text{Val}_2) \land \text{Val}_1 \neq \text{Val}_2.\]
\[(B) \leftarrow \text{call} \_ \text{pinMode}(T, \text{Port}, \text{Mode}_1) \land \text{call} \_ \text{pinMode}(T, \text{Port}, \text{Mode}_2) \land \text{Mode}_1 \neq \text{Mode}_2.\]
\[(C) \leftarrow \text{turn} \_ \text{on}(T) \land \text{turn} \_ \text{off}(T).\]
\[(D) \leftarrow \text{turn} \_ \text{on}(T) \land \text{delayed} \_ \text{turn} \_ \text{off}(T, \text{From}, \text{Delay}).\]
\[(E) \leftarrow \text{turn} \_ \text{off}(T) \land \text{delayed} \_ \text{turn} \_ \text{on}(T, \text{From}, \text{Delay}).\]
\[(F) \leftarrow \text{delayed} \_ \text{turn} \_ \text{on}(T, \text{From}_1, \text{Delay}_1) \land \text{delayed} \_ \text{turn} \_ \text{off}(T, \text{From}_2, \text{Delay}_2).\]
\[(G) \leftarrow \text{turn} \_ \text{on}(T) \land \text{delayed} \_ \text{turn} \_ \text{on}(T, \text{From}, \text{Delay}).\]
\[(H) \leftarrow \text{turn} \_ \text{off}(T) \land \text{delayed} \_ \text{turn} \_ \text{off}(T, \text{From}, \text{Delay}).\]
\[(I) \leftarrow \text{delayed} \_ \text{turn} \_ \text{off}(T, \text{From}_1, \text{Delay}_1) \land \text{delayed} \_ \text{turn} \_ \text{off}(T, \text{From}_2, \text{Delay}_2) \land (\text{From}_1 \neq \text{From}_2 \lor \text{Delay}_1 \neq \text{Delay}_2).\]
\[(J) \leftarrow \text{delayed} \_ \text{turn} \_ \text{on}(T, \text{From}_1, \text{Delay}_1) \land \text{delayed} \_ \text{turn} \_ \text{on}(T, \text{From}_2, \text{Delay}_2) \land (\text{From}_1 \neq \text{From}_2 \lor \text{Delay}_1 \neq \text{Delay}_2).\]
\[(K) \leftarrow \text{ret} \_ \text{millis}(T, \text{Now}_1) \land \text{ret} \_ \text{millis}(T, \text{Now}_2) \land \text{Now}_1 \neq \text{Now}_2.\]

Fig. 3. Constraints for the Blink Program

The predicates \text{turn} \_ \text{on}, \text{turn} \_ \text{off}, \text{delayed} \_ \text{turn} \_ \text{on}, \text{delayed} \_ \text{turn} \_ \text{off} occur in the same state. There should be an abbreviation for such a constraint set: “For every \( T \), at most one instance of \( \text{turn} \_ \text{on}(T) \), \( \text{turn} \_ \text{off}(T) \), \( \text{delayed} \_ \text{turn} \_ \text{on}(T, \text{From}, \text{Delay}) \), \( \text{delayed} \_ \text{turn} \_ \text{off}(T, \text{From}, \text{Delay}) \) 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 \( \text{ret} \_ f \) predicates.

Actually, one could permit more formulas as \( \varphi \) in the constraints. For instance, the monotonicity of \( \text{ret} \_ \text{millis} \) can be expressed as

\[\leftarrow \text{ret} \_ \text{millis}(T_1, \text{Now}_1) \land \text{ret} \_ \text{millis}(T_2, \text{Now}_2) \land \\
\neg((T_1 > T_2 \land \text{Now}_1 < \text{Now}_2) \lor (T_1 < T_2 \land \text{Now}_1 > \text{Now}_2)).\]

What is needed in the following is that (1) every variable in \( \varphi \) occurs also in at least one of the two literals, and (2) conjunctions of \( \varphi \) and the built-in predicates and the match conditions below are decidable.
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 $\leftarrow A_1 \land A_2 \land \varphi$ be given. Let

- $A'_1 \leftarrow B_1 \land \cdots \land B_m$ ($m \geq 0$) be a variant with fresh variables of a rule in $P$,
- $A'_2 \leftarrow C_1 \land \cdots \land C_n$ ($n \geq 0$) 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 $\theta$ be a most general unifier.

Then the violation condition is:

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

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 disequations $t_i \neq u_i$ 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 further. If $t_i$ and $u_i$ are distinct constants for some $i$, the disequation and therefore the whole disjunction can be simplified to true.

**Example 5.** Consider constraint (A), the key constraint for `call digitalWrite`:

$\leftarrow call\, digitalWrite(T, Port, Val_1) \land call\, digitalWrite(T, Port, Val_2) \land Val_1 \neq Val_2$.

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

- `call_digitalWrite(T, Port, Val_1)`
- `call_digitalWrite(T, Port, Val_2)`

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):

- `call_digitalWrite(T_3, 13, #HIGH)`
- `call_digitalWrite(T_4, 13, #LOW)`

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

$\{T_3/T, T_4/T, Port/13, Val_1/#HIGH, Val_2/#LOW\}$. 


MGUs are unique modulo a variable renaming. Now the violation condition is
\[ \text{turn}_\text{on}(T) \land \text{turn}_\text{off}(T) \land \#\text{HIGH} \neq \#\text{LOW}. \]
Since \( \#\text{HIGH} \neq \#\text{LOW} \) is true, the violation condition can be simplified to
\[ \text{turn}_\text{on}(T) \land \text{turn}_\text{off}(T). \]
This is what we would expect: It should never happen that \( \text{turn}_\text{on} \) and \( \text{turn}_\text{off} \) 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 \( \#\text{HIGH} \neq \#\text{HIGH} \), 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.** \( T_P(I) \) violates an exclusion constraint \( \leftarrow A_1 \land A_2 \land \varphi \) if and only if there is a violation condition for \( P \) and \( \leftarrow A_1 \land A_2 \land \varphi \) which is true in \( I \).

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. By iterating the \( T_P \) operator, we get the minimal model, which is the intended interpretation of \( P \). The iteration starts with the empty set of facts \( I_0 \) which certainly satisfies all exclusion constraints.

Now if under the assumption that \( I_i \) satisfies the exclusion constraints, and possible other knowledge about \( I_i \), we can prove that all violation conditions have no solution in \( I_i \), we can conclude that \( I_{i+1} := T_P(I_i) \) satisfies the constraints. By induction this follows for every finite iteration. The minimal model is the result of transfinite iteration: \( I_\omega := \bigcup_{i \in \mathbb{N}} I_i \). If this would violate \( \leftarrow A_1 \land A_2 \land \varphi \) for a ground substitution \( \theta \), the facts \( A_1 \theta \) and \( A_2 \theta \) would have to be contained already in \( I_i \) and \( I_j \), and therefore \( I_{\max(i,j)} \), violates the constraint, too. This gives us:

**Theorem 2.** Let \( P \) be a Datalog program, \( C \) 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 \( C \) are false in all \( I \in \mathcal{H} \) that satisfy \( C \), then the minimal Herbrand model \( I_\omega \) of \( P \) satisfies \( C \).

Thus, we basically 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 \( \text{ret}_f \)-predicates cannot occur in state 0: This is obvious, because there is no previous state that might contain a call.

We also plan to precompute all predicates that might occur in state 0 and in state 1. Since this is done on the predicate level, it is fast. Then we could consider only interpretations \( \mathcal{H} \) that are compatible with this knowledge.
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 [9,5].

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 $← C_1 ∧ C_2 ∧ γ$ be given. Let $← C'_1 ∧ C'_2 ∧ γ'$ 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 without loss of generality).
- 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

\[
t_1 = t_1θ ∧ \cdots ∧ t_n = t_nθ ∧ u_1 = u_1θ ∧ \cdots ∧ u_m = u_mθ ∧ γ'θ.
\]

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

\[
A_1 ∧ \cdots ∧ A_m ∧ B_1 ∧ \cdots ∧ B_m ∧ ϕ
\]

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 ($∧$) of the following parts:

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

Example 6. This example continues Example 5 with the violation condition:
\[ \text{turn} \_{\text{on}}(T) \land \text{turn} \_{\text{off}}(T). \]

Now consider constraint (C): \[ \leftarrow \text{turn} \_{\text{on}}(T) \land \text{turn} \_{\text{off}}(T). \] 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 \( T = T \land T = T \), which can be simplified to \( \text{true} \). Since there is no other matching constraint, the violation formula, which requires that the violation condition does not violate the constraint, is \( \neg \text{true} \), i.e. \( \text{false} \). With this we have proven that 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. \( \Box \)

Example 7. For a more complex case, let us consider Constraint (J) which ensures that there can be only one fact about \( \text{delayed} \_{\text{turn} \_{\text{on}}} \) in each state:
\[ \leftarrow \text{delayed} \_{\text{turn} \_{\text{on}}}(T, \text{From}_1, \text{Delay}_1) \land \text{delayed} \_{\text{turn} \_{\text{on}}}(T, \text{From}_2, \text{Delay}_2) \land (\text{From}_1 \neq \text{From}_2 \lor \text{Delay}_1 \neq \text{Delay}_2). \]

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):

- \[ \text{delayed} \_{\text{turn} \_{\text{on}}}(S_3, \text{From}_3, 1000) \leftarrow \text{turn} \_{\text{off}}(T_3) \land \text{ret} \_{\text{millis}}(T_3, \text{From}_3) \land \text{succ}(T_3, S_3). \]
- \[ \text{delayed} \_{\text{turn} \_{\text{on}}}(S_4, \text{From}_4, 1000) \leftarrow \text{turn} \_{\text{off}}(T_4) \land \text{ret} \_{\text{millis}}(T_4, \text{From}_4) \land \text{succ}(T_4, S_4). \]

An MGU is
\[ \{S_3/T, \text{From}_3/\text{From}_1, \text{Delay}_1/1000, S_4/T, \text{From}_4/\text{From}_2, \text{Delay}_2/1000\}. \]

Thus, the resulting violation condition is:
\[
\begin{align*}
\text{turn} \_{\text{off}}(T_3) & \land \text{ret} \_{\text{millis}}(T_3, \text{From}_1) \land \text{succ}(T_3, T) \land \\
\text{turn} \_{\text{off}}(T_4) & \land \text{ret} \_{\text{millis}}(T_4, \text{From}_2) \land \text{succ}(T_4, T) \land \\
(\text{From}_1 & \neq \text{From}_2 \lor 1000 \neq 1000)
\end{align*}
\]

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

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

\[ T = T_3 + 1 \land T_3 \geq 0 \land T_4 = T + 1 \land T_4 \geq 0. \]

Note that \( T_3 = T_4 \) 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:

\[ \leftarrow \text{ret\_millis}(T_5, \text{Now}_5) \land \text{ret\_millis}(T_5, \text{Now}_6) \land (\text{Now}_5 \neq \text{Now}_6) \]

An MGU with variable-to-variable bindings directed towards the violation condition is \( \{T_5/T_3, \text{Now}_5/\text{From}_1, T_4/T_3, \text{Now}_6/\text{From}_2\} \). This gives the following match condition:

\[
T_3 = T_3 \land \text{From}_1 = \text{From}_1 \land T_4 = T_3 \land \text{From}_2 = \text{From}_2 \land \text{From}_1 \neq \text{From}_2.
\]

With the trivial equalities removed, this is \( T_4 = T_3 \land \text{From}_1 \neq \text{From}_2 \). The negation is added to the violation formula. Thus the total violation formula is:

\[
\text{From}_1 \neq \text{From}_2 \land \\
T = T_3 + 1 \land T_3 \geq 0 \land T_4 = T + 1 \land T_4 \geq 0 \land \\
\neg(T_4 = T_3 \land \text{From}_1 \neq \text{From}_2).
\]

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. \( \square \)

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

### 5 Conclusions

We are investigating the programming of microcontrollers in Datalog. For our first language proposal and the results of a prototype implementation, see

https://dbs.informatik.uni-halle.de/microlog/

The version in this paper has a simpler semantics (without complicating programming), and will soon be incorporated in the prototype. The above page also links to a first prototype of the constraint check described in this paper.

Programs should be provably correct. 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.

References