Integrity Constraints for Microcontroller Programming in Datalog

Stefan Brass and Mario Wenzel

University of Halle, Germany

ADBIS 2021 (Aug. 26, 2021)

Contents

It all Started with my Interest in Fireworks . . .

Professional Fireworks are electrically ignited

The Core of a Firing System is a Microcontroller

Microcontrollers

- A Microcontroller is a small computer on a single chip.
- E.g., the Amtel ATmega328P ($\sim$ 2–3 €) contains
  - 8-bit CPU (AVR instruction set, 32 registers, $\leq$ 20 Mhz)
  - 32 KByte flash memory for the program
  - 2 KByte static RAM
  - 1 KByte EEPROM for persistant data
  - 23 general purpose I/O pins

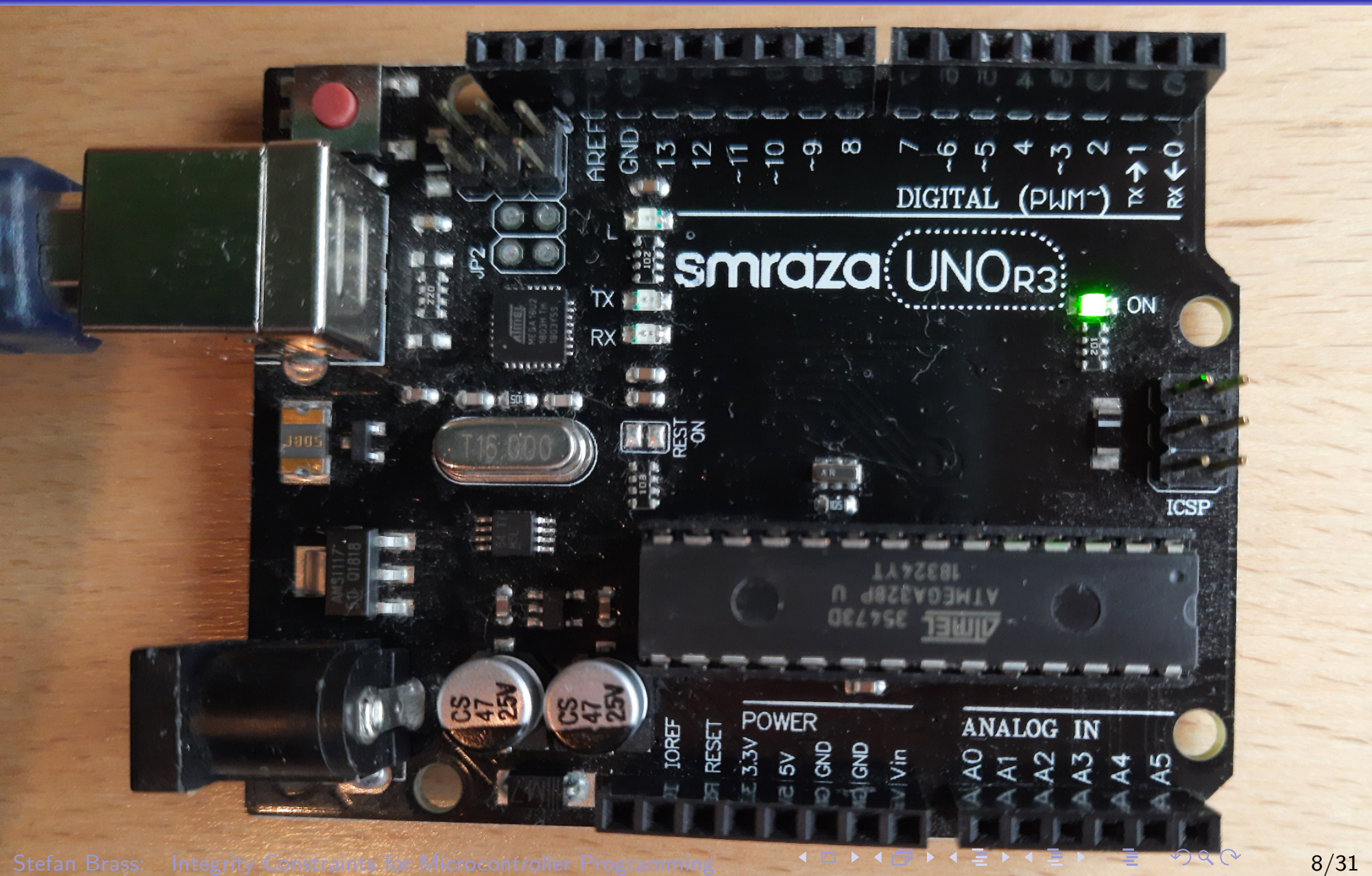
    Minus 1, if one needs a reset pin, and 2 for an external osciallator.

    Many pins have additional special functions besides digital input/output.
  - 3 timers (1$*$16 Bit, 2$*$8 Bit) with pulse-width modulators
  - 6 analog/digital converters (10 Bit resolution)
  - support for serial interfaces (UART/USART, SPI, I^2C)

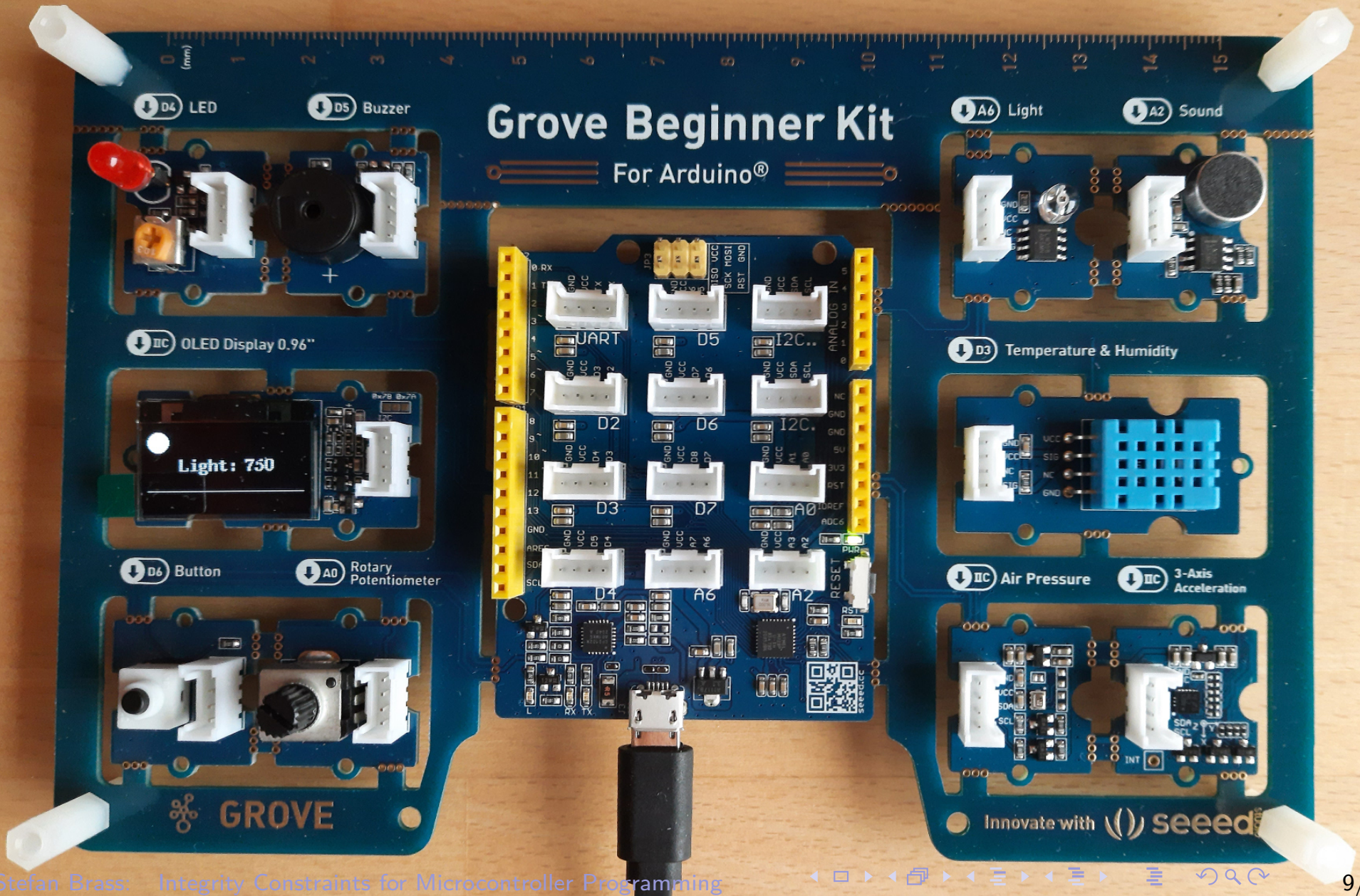
Arduino

- Arduino is a platform for using microcontrollers.
  It consists of
  - a family of boards with such microcontrollers,
    E.g. the Arduino UNO R3 contains the ATmega328P (at 16 MHz).
  - a preinstalled boot loader on the microcontroller,
    This permits to store programs from a connected PC via a USB interface.
  - an IDE for developing software for the Arduino in C,
    IDE, boot loader and hardware design are open source.
  - hardware extension "shields" that easily connect to the main board (e.g., for a display),
  - many software libraries,
  - an active community.
- An Arduino UNO compatible board costs about 10 €.

Arduino Compatible Board



Arduino Clone with Additional Sensors and Display



Contents

Datalog for Microcontroller Programming

- Declarative programs are often shorter than equivalent programs in C (or assembler).

  E.g., our blink example: 6 lines, similar program from the Arduino tutorial: 16.

- Relatively simple language, also for non-experts.

  Arduino boards are a nice device to be used in school.

- Mathematically precise semantics based on logic makes programs easier to verify.

  A verification task is the main subject of this paper.

- Simple semantics permits powerful optimizations.

  We translated a subset of programs to extended finite state machines.

- Data tables (e.g., for configuration data) can be easily written as Datalog facts (data-driven architecture with small DB).

  In another paper, we used a small home automation system as example.

State Sequence

- Microcontroller programs must act in time.

  They specify an input-output behaviour, and the input changes over time.

- Therefore, we use the natural numbers 0, 1, 2, ... as an infinite sequence of states ("logical time").
- Every predicate has an additional state argument (the first).
- Rules are of two types:
  - Rules for deriving facts within a state:
    All state arguments are filled with the same variable T.
  - Rules for deriving facts for the next state:
    The body of the rule contains $succ(T, S)$, the state argument of all other body literals is T (current state), and the state argument of the head literal is S (next state).

  The variables T and S are reserved and cannot appear anywhere else.

Abbreviations (inspired by language Dedalus)

- One must distinguish between
  - the abbreviated syntax used by the programmer ("Microlog"),
  - and the Datalog program after expanding the abbreviations. This defines the semantics of the Microlog program.
- The state argument is not shown in the abbreviated syntax.
- Rules to infer facts about the successor state are marked with `@next` appended to the head literal.

  Facts with `@init` hold in state `0` (for setup), facts with `@start` in state `1`.
- Example (predicate that is true in all states):

  Microlog: $always@init.$
  $always@next \leftarrow always.$

  Datalog: $always(0).$
  $always(S) \leftarrow always(T) \wedge succ(T, S).$

Interface Predicates for Input and Output

- The Arduino libraries offer many C functions such as
  - `void pinMode(uint8_t pin, uint8_t mode)`
  - `void digitalWrite(uint8_t pin, uint8_t val)`
  - `unsigned long millis(void)`
- For each such function f, there are two interface predicates:
  - $call\_f$ (with the same arguments), and
  - ret_f (with one additional argument for the return value).

    Unless the return type is `void`.
- If $call\_f(c_1, \ldots, c_n)$ is derived, $f(c_1, \ldots, c_n)$ is called, and $ret\_f(c_1, \ldots, c_n, c)$ with the return value c is available in the next state.
- In Microlog, $@call$ and $@ret$ are used instead of the prefix.

  And ? is written in the call for the return value (to make the arity equal).

Abbreviation for Real Time Delay

- `millis()` returns the real time since the program was started.
- Abbreviation:

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

- Expanded to:

$$delayed\_p(t_1, \ldots, t_n, From, Delay)@next \leftarrow A_1 \wedge \cdots \wedge A_m \wedge millis@ret(From).$$

$$delayed\_p(X_1, \ldots, X_n, From, Delay)@next \leftarrow delayed\_p(X_1, \ldots, X_n, From, Delay) \wedge millis@ret(Now) \wedge From + Delay < Now.$$

$$p(X_1, \ldots, X_n)@next \leftarrow delayed\_p(X_1, \ldots, X_n, From, Delay) \wedge millis@ret(Now) \wedge From + Delay \geq Now.$$

$$millis(?)@call.$$

Example

- On most Arduino boards, a LED is connected to Pin 13.
- The following program turns the LED on for 1000 ms, then off for 1000 ms, and so on (i.e. the LED blinks):

$$
\begin{aligned}
&\mathit{pinMode}(13, \#\mathit{OUTPUT})@\mathit{setup}.\\
&\mathit{turn\_on}@\mathit{start}.\\
&\mathit{turn\_off}@\mathit{after}(1000) \leftarrow \mathit{turn\_on}.\\
&\mathit{turn\_on}@\mathit{after}(1000) \leftarrow \mathit{turn\_off}.\\
&\mathit{digitalWrite}(13, \#\mathit{HIGH})@\mathit{call} \leftarrow \mathit{turn\_on}.\\
&\mathit{digitalWrite}(13, \#\mathit{LOW})@\mathit{call} \leftarrow \mathit{turn\_off}.
\end{aligned}
$$

- We use the notation $\#C$ to refer to constants defined in the include-files of the library.

  These constants are inserted into the generated code. We must assume that different constants used in the same predicate argument are indeed distinct.

Contents

The Task: Is memory sufficient?

- Derivable facts are computed state by state.
- The memory (RAM, e.g. 2 KByte) must contain:
  - All facts of the current state,
  - facts derived for the next state,
  - loop variables and other local variables of the implementation.
- For program verification, it is necessary to show that there will never be more facts than fit in the very limited memory.
- We use a kind of constraints that ensures that certain facts exclude each other, i.e. each state can contain only one out of a larger set of facts.

  For proving that memory is sufficient, we also need to look at argument types (some contain only two different values) and compute an over-approximation of the initial and the start state (at least the predicates that might occur in them).

Generalized Exclusion Constraints

- A "Generalized Exclusion Constraint" (GEC) is a formula of the form

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

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

- As usual in logic programming, the headless "$\leftarrow$" simply means "$\neg$", i.e. the constraint rule may never be applicable.

- The constraint is logically equivalent to

$$t_{i_1} = u_{i_1} \wedge \cdots \wedge t_{i_k} = u_{i_k} \leftarrow p(t_1, \ldots, t_n) \wedge q(u_1, \ldots, u_m).$$

  The head may also be *false* (if $k = 0$).

Relation to Other Constraint Types

- GECs include keys and functional dependencies.
  E.g., we cannot output two different values (high and low) on the same pin in the same state:

$$\leftarrow \quad call\_digitalWrite(T, Pin, Val_1) \wedge call\_digitalWrite(T, Pin, Val_2) \wedge Val_1 \neq Val_2.$$

  While keys obviously help to reduce the number of facts that might be true in a state, excluding conflicting usages of the same pin is also an important verfication task in its own right.

- Of course, known exclusion constraints fit in this framework:

$$\leftarrow \quad turn\_on(T) \wedge turn\_off(T).$$

  Exclusion constraints (introduced by Casanova/Vidal in 1983, and investigated further by Thalheim and Schewe) require that projections of two relations are disjoint: $\pi_{A_{i_1},\ldots,A_{i_n}}(R) \cap \pi_{B_{j_1},\ldots,B_{j_m}}(S) = \emptyset$.

More Related Work

- "Negative Constraints" have the form:
  - $\leftarrow p(t_1, \ldots, t_n).$
  - $\leftarrow p(t_1, \ldots, t_n) \wedge q(u_1, \ldots, u_m).$ (with at least one $t_i = u_j$)

    Such constraints were used by Chabin/Halfeld-Ferrari/Markhoff/Nguyen in their work on validating data from semantic web providers (2018), and by Calí/Gottlob/Lukasiewicz in their work on Datalog$^{\pm}$.
- Approaches also differ in the way the constraints are used:
  - Schewe/Thalheim modify specifications of operations in state oriented systems such that they cannot violate constraints.

    "Towards a theory of consistency enforcement", Acta Informatica, 1999.
  - Chabin et.al. modify queries such that they ignore data that violates the constraints.
  - Our task is to prove that all states occurring during program execution do not violate the constraints.

Contents

Computing Violation Conditions (1)

- We consider all possibilities to derive integrity violations.
- E.g. consider the constraint:

$$\leftarrow \quad call\_digitalWrite(T, Pin, Val_1) \wedge \\ call\_digitalWrite(T, Pin, Val_2) \wedge \\ Val_1 \neq Val_2.$$

- A violation of this constraint means that two matching *call_digitalWrite*-facts are derivable.
- Now we look at all rule pairs that might yield these facts.
- Rules with matching head literals:
  - $call\_digitalWrite(T, 13, \#HIGH) \leftarrow turn\_on(T).$
  - $call\_digitalWrite(T, 13, \#LOW) \leftarrow turn\_off(T).$

Computing Violation Conditions (2)

- We apply unfolding (an SLD resolution step) to the constraint and the rules with matching heads:

$$\leftarrow \quad call\_digitalWrite(\ T,\ Pin,\ \ Val_1\ ) \wedge$$
$$call\_digitalWrite(\ T',\ 13,\ \#HIGH\ ) \leftarrow \quad turn\_on(T').$$

$$call\_digitalWrite(\ T,\ Pin,\ \ Val_2\ ) \wedge$$
$$call\_digitalWrite(\ T'',\ 13,\ \#LOW\ ) \leftarrow \quad turn\_off(T'').$$

$$Val_1 \neq Val_2.$$

- Unifier:
  $\{T'/T,\ T''/T,\ Pin/13,\ Val_1/\#HIGH,\ Val_2/\#LOW\}$
- Result (one of many violation conditions):

$$turn\_on(T)\ \wedge\ turn\_off(T)\ \wedge\ \#HIGH \neq \#LOW.$$

The inequality $\#HIGH \neq \#LOW$ is true and can be removed.

Meaning of Violation Conditions

- Consider the computation of the minimal model of the Datalog program:

$$\begin{aligned} \mathcal{I}_0 &:= \emptyset \\ \mathcal{I}_1 &:= T_P(\mathcal{I}_0) \\ &\vdots \\ \mathcal{I}_j &:= T_P(\mathcal{I}_{j-1}) \\ &\vdots \end{aligned}$$

- If a violation condition is satisfied in $\mathcal{I}_{j-1}$, the corresponding GEC is violated in $\mathcal{I}_j$.

  By applying the two rules, we get facts that make the body of the GEC true.

- And conversely, if a GEC is violated in $\mathcal{I}_j$, one of its violation conditions was satisfied in $\mathcal{I}_{j-1}$.

Refuting Violation Conditions (1)

- Now we must show that all computed violation conditions are false in all interpretations that occur during the computation of the minimal model.
- We do this by applying the GECs.
- We assume that the GECs were satisfied in $\mathcal{I}_{j-1}$.

  All GECs are certainly satisfied in $\mathcal{I}_0 := \emptyset$.
- Based on that, we show that all violation conditions are false in $\mathcal{I}_{j-1}$.

  It might be possible to use other knowledge in addition to show that the violation conditions can never by satisfied. E.g., we might compute all facts that are true in the "setup state" 0 (these do not depend on function results).
- Then it follows that the GECs are satisfied in $\mathcal{I}_j$.

Refuting Violation Conditions (2)

- In the example, the violation condition

  $$turn\_on(T) \wedge turn\_off(T) \wedge \#HIGH \neq \#LOW.$$

  cannot be satisfied because of the GEC

  $$\leftarrow turn\_on(T) \wedge turn\_off(T).$$

- In general, it is a bit more complicated, because the violation condition can have any number of literals.

  It is composed from the two rule bodies and the disjunction of inequalities from the GEC that might be violated. We might also apply several GECs to prove the violation condition unsatisfiable. See "match condition" in the paper.

- But basically, the task of the programmer who wants to use this verification method is to find a set of GECs that can reproduce itself.

  The system can show violation conditions that cannot be refuted yet.

Constraints in the Example

- The functional property of the *ret_f*-predicates could be automatically assumed.

  $\leftarrow ret\_millis(T, Now_1) \wedge ret\_millis(T, Now_2) \wedge Now_1 \neq Now_2.$

- A library could add the constraints that two calls to `digitalWrite` or `pinMode` for the same pin in the same state are excluded.
- There should be an abbreviation for specifying: For each T, at most one instance of the following facts is true:
  - $turn\_on(T)$
  - $turn\_off(T)$
  - $delayed\_turn\_on(T, From, Delay)$
  - $delayed\_turn\_off(T, From, Delay)$

  This actually needs 8 GECs.

Required Memory: In total 8 facts

- For the current state, there can be one fact about each of the predicates:
  - *call_pinMode*
  - *call_digitalWrite*
  - *call_millis*
  - *ret_millis*
  - One of: *turn_on*, *delayed_turn_on*, *turn_off*, *delayed_turn_off*.
  - *always* (system predicate, should not actually be stored)
- For the next state, two facts are derived:
  - One of: *turn_on*, *delayed_turn_on*, *turn_off*, *delayed_turn_off*.
  - *always*

Contents

Conclusions

- The technique is not actually limited to constraints with exactly two literals.
- I am still searching for nice constraint abbreviations.

  I feel a bit ashamed that currently, the constraints must be specified on the Datalog level, not the Microlog level. There also wouldn't have been any space in the paper for a solution ...
- Microlog Compiler (Mario Wenzel): [https://dbs.informatik.uni-halle.de/microlog/]
- Constraint Checker (Stefan Brass): [https://users.informatik.uni-halle.de/~brass/micrologS/]
- There are other interesting types of constraints for this application, e.g. one must call *pinMode*(*Pin*, #*OUTPUT*) before *digitalWrite*(*Pin*, ...).