TECHNISCHE UNIVERSITÄT
KAISERSLAUTERN

Symbolic Execution of Synchronous Quartz Programs

MasterThesis

von

Daniel Theis

February 14, 2022

Technische Universität Kaiserslautern,
Department of Computer Science,
67663 Kaiserslautern,
Germany

Examiner: Prof. Dr. Klaus Schneider
M. Eng. Xiao Wang

Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Symbolic Execution of Synchronous Quartz Programs" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

Kaiserslautern, den 14.2.2022

Daniel Theis

Acknowledgements

First I want to thank Prof. Dr. Klaus Schneider for the possibility to write this master thesis as an introduction to further promotional studies, as well as for his extraordinary guidance and advice. I also want to thank my mentor M. Eng. Xiao Wang for supervising this thesis together with the main examiner Prof. Dr. Klaus. Schneider. Last but not least I want to thank my parents and my sisters for their emotional support through the whole time of this master thesis as well as my complete studies.

Abstract

In synchronous programs, all actions that to not explicitly take time happen instantaneously and need to synchronize with each other. This makes the verification of such programs more complicated, as dependencies between different assignments that should happen at the same time need to be considered. By doing a symbolic execution, these dependencies can be integrated into causality conditions that must hold regardless of the input values. However as soon as those programs use numeric values beside Boolean signaling values, the number of possible input values is enlarged significantly.
In this thesis, a tool is introduced that implements such a symbolic execution in the language Quartz, which can verify causality step-wise while also taking care of dependencies between numeric variables. For this, a given program is divided into its possible macrosteps with their defined guarded actions, which is represented by an extended finite state machine. By using two newly introduced operators, an expression capturing all dependencies is created for each variable in each state, which can then be transformed to causality conditions. Finally, the SMT-solver Z3 is used to verify these conditions for arbitrary input values.

Zusammenfassung

In synchronen Programmen werden alle Aktionen die nicht explizit Zeit benötigen instantan ausgeführt und müssen sich miteinander synchronisieren. Hierdurch wird die Verifikation von solchen Programmen schwieriger, da Abhängigkeiten zwischen verschiedenen Zuweisungen, die gleichzeitig passieren sollen, beachtet werden müssen. Durch symbolisches Ausführen können diese Abhängigkeiten in Kausalitätsbedingungen integriert werden, welche unabhängig von den Eingabevariablen gelten müssen. Sobald diese Programme jedoch numerische Werte neben Boolschen Signalwerten verwenden, erhöht sich die Anzahl an möglichen Eingabewerten signifikant.
In dieser Arbeit wird ein Tool vorgestellt, welches eine solche symbolische Ausführung in der Sprache Quartz implementiert, die Kausalität schrittweise überprüfen kann und währenddessen auch Abhängigkeiten zwischen numerischen Variablen beachtet. Hierfür wird ein gegebenes Programm in seine möglichen Makro-Schritte mit definierten geschützten Aktionen zerlegt, was durch einen erweiterten endlichen Zustandsautomaten repräsentiert wird. Durch zwei neu eingeführte Operatoren wird für jede Variable in jedem Zustand ein Ausdruck erstellt, welcher alle Abhängigkeiten beinhaltet und daraufhin in Kausalitätsbedingungen umgewandelt werden kann. Schlussendlich wird der SMT-Solver Z3 verwendet um diese Bedingungen für beliebige Eingabewerte zu verifizieren.

Contents

List of Figures

1. Introduction

For the development of any software or program, testing and verification of the code has always been a very crucial step. While higher-level compilers, beside checking the code for correct syntax, are already able to spot unused code sequences or never reachable cases of conditional, they unfortunately can not reason about the semantic correctness of a program, which is about the question if a given program "does" what it "should do" in regard to its specification. The definition of what a program should do is however also relatively obscure as it depends on the given context. While functional programs usually should terminate at some point, providing a result of a certain computation, reactive systems often should never terminate or only when given a certain input (for example an `off` command) and otherwise continuously react to the environment. The sudden termination of an EKG (electrocardiogram) for example could have lethal consequences, thus ensuring that this does not happen is a fundamentally important step during verification.

With synchronous languages playing an important role for reactive systems due to its synchrony requirements and zero-time reactions, the need for verification also applies to them. However the behavior of zero-time actions and reactions introduces additional problems which do not appear in asynchronous languages. While the exact set of problems depends on the concrete implementation of the zero-time behavior, a certain class of problems can be defined as causality problems which are often induced by causality cycles (i.e. variable assignments or signal emissions that depend on each other). In asynchronous programs, one of those actions would be strictly executed before the next, i.e. in a given order, if atomicity is assumed for the actions. In synchronous programs however, if both dependent actions should happen at the same time. This dependency must somehow be resolved. Causal behavior is achieved, when there is a possible solution (such as a variable assignment satisfying all dependencies) and this solution is also unique, otherwise there would be a non-deterministic choice of one of the multiple solutions. As such dependency cycles can be arbitrary complex, constructive causality further extends this definition to causal solutions that can be deduced without guessing any variable value, which makes it possible to compute the solution in a reasonable time (i.e. not brute-forcing all possible solutions).

The idea of the tool which is introduced in this thesis, is to perform a symbolic execution on programs in the synchronous language Quartz, which verifies causality for all time-steps and further provides input values to reach a certain execution point of the program for further verification. While symbolic causality analysis have been done with different approaches for various synchronous languages [SBT96; HM95; SBS04], they often were reduced to

only signals or Boolean variable dependencies while not arguing about numeric values. Having only Boolean expressions with Boolean values makes it possible to use SAT-solvers to verify causality conditions in a symbolical way. However if numeric operations and predicates are contained in such an expression, these must be encoded to their Boolean representation first, which enlarges the number of variables and thus may render these approaches unfeasible. In contrast to that, the presented the tool in this thesis makes use of the SMT-solver Z3 to directly work with numeric variables and expressions.

The thesis is divided into six Chapters, with the first being this introduction. The second Chapter 2 gives the general preliminaries for synchronous languages and symbolic execution as well as introducing the language Quartz, while Chapter 3 formally introduces the used (symbolic) types and two new operators. Chapter 4 then finally explains the tool and its implementation in more detail, with Chapter 5 describing the transition of the computed causality conditions to Z3. Finally, Chapter 6 summarizes the thesis together with an overview of possible future work.

2. Preliminaries

The general idea of verification is nothing new in terms of software testing and is thus an important step during quality control to assure that a given program is working correctly. However, the definition of what "correct" means may be different, depending on the given context. While a program simply implementing a given mathematical function is deemed correct if it returns the expected output when given a certain input, another program in a real-time scenario must react in a given short time frame to be correct, as a "too slow" reaction is considered incorrect behavior. For synchronous languages the correctness terminology is even more profound as it extends to causality, which means that in each step of the computation, each variable has a unique determined value which can also be computed deterministically. The following chapter provides the preliminaries for this verification problem.

2.1. Synchronous Language Paradigm

As the name suggests, the main idea of synchronous languages is, that actions should happen synchronously, i.e. at the same time. This stands in contrast to classic programming languages, in which each action happens one after the other in a predetermined order. However, in contrast to concurrent programs which synchronize for each action, the restriction for synchrony in synchronous programs is even stricter. The so-called synchrony hypothesis (after [BG92]) states, that each action and reaction should take no time, i.e. it is assumed that each action is in fact instantaneous. The external environment, in which actions are executed, remains invariant and is equal for all actions, however, changes done by actions have direct effect on the other actions and vice-versa. Furthermore, a statement that provides a temporal context exactly defines the time it takes and in reverse, only takes time if it is explicitly stated. This abstracts synchronous languages from a real world view, where each computation takes time, even if it is only a nanosecond, and instead behaves more as a mathematical model. However, there is a strong resemblance to combinational circuits, which are also assumed to produce an output instantaneously given a certain input, in contrast to sequential circuits.

This reason together with the synchrony hypothesis in general makes synchronous languages ideal for reactive systems [Hal98], as reactions in such systems should also happen instantaneously (at least in theory) and additionally be deterministic [Ber89]. There exist various synchronous languages with different programming styles behind; the most common known are ESTEREL [BG92] which is an imperative languages, LUSTRE [PHP87; Hal+91] and SIGNAL [Le +86] which are data-flow driven and last but not least STATE-

CHARTS [Har87] which uses a hierarchical representation of automata. In the following, ESTEREL is given as an example, as its imperative style is adapted in Quartz [Sch09] and thus acts as a basis for the problem descriptions. Additionally, differences in the zero-time semantics between ESTEREL and Quartz can be explained at a later point.

2.1.1. Example: ESTEREL

Although the original paper [BG92] defines a small difference between so called "basic" and "plain" ESTEREL, this exemplary presentation will not distinguish between the two variants, as the plain variant mostly adds syntactic sugar to ease the use of programming, while maintaining a formally correct definition through the basic ESTEREL statements. Modern versions of ESTEREL have improved the programmatic design and added/refined statements, types and the semantic in general, which thus may differ from the explanations given here [Ber00]. That said, basic ESTEREL has only three primitive types, which are `integer`, `boolean` and `triv` (which is a unique constant with the same name). Further abstract types like `DOUBLE` or `TIME` can be declared by the user, which can be used for variables in later use (although they have no defined behavior other than being differentiable from other types). For communication purposes, signals can be used, which also have a defined type. The difference between variables and signals is the fact that variables are always present in their declared scope, while signals can be present or absent during each reaction, which is useful for event-driven sensors. With *stat* being a statement, *exp* being an expression and *type* being a primitive or user-declared type, the list of basic statements is given in Table 2.1.

Table 2.1.: *Basic ESTEREL statements [BG92]*

`nothing`	dummy statement
`halt`	halting statement
`X :=` *exp*	assignment statement
`call P` (variable-list) (expression-list)	external procedure call
`emit S`(*exp*)	signal emission
$stat_1$; $stat_2$	sequence
`loop` *stat* `end`	infinite loop
`if` *exp* `then` $stat_1$ `else` $stat_2$ `end`	conditional
`present S then` $stat_1$ `else` $stat_2$ `end`	test for signal presence
`do` *stat* `watching S`	watchdog
$stat_1 \parallel stat_2$	parallel statement
`trap T in` *stat* `end`	trap definition
`exit T`	exit from trap
`var X :` *type* `in` *stat* `end`	local variable declaration
`signal S` (`combine` *type* `with` *comb*) `in` *stat* `end`	local signal declaration

In ESTEREL, statements are executed in the given program order, however

each statements takes zero time in regard to the environment. Thus sequential variable assignments like

```
X := 0;  X := X + 3
```

are totally correct with X finally being 3 while still happening in zero time. In contrast to that, the three statement

```
emit(S1); emit(S2)    emit(S2); emit(S1)    emit(S1)||emit(S2)
```

are all equal with both signals `S1` and `S2` being present at the same (zero) time with the trivial value `triv`. As the `trap T` statement aborts its body statement as soon as `exit T` is executed, the following example [BG92] demonstrates the timing in regard to simultaneous execution:

```
trap T1 in
    trap T2 in
        X := 0
        ||
        Y := 0; exit T2
        ||
        Z := 0; exit T1; Z := 1
    end;
    U := 0
end
```

At first, both `trap` statements are entered simultaneously. Each of the parallel branches executes its statements in order but at the same time as the other branches. As `exit T1` comes before `Z := 1` in its branch sequence, its execution leads to the instant abortion of the trap statement such that `Z := 1` is not executed. At the same time, in the second branch `trap T2` is exited due to `exit T2`. However in the case of two simultaneous abortions, the outer abortion has priority (thus also aborting the inner `trap` statement) which leads to the instantaneous termination of the whole statement. Because of this reason, `U := 0` is also not executed as it comes in sequence after `trap T2` which was aborted due to the outer `trap`. The final result is thus `X`, `Y` and `Z` being 0.

If the same signal is emitted multiple times, the function `comb` in the signals declaration defines how the occurrence of multiple signal values is handled. If no explicit function is given, multiple `triv` value just act as a single `triv` value (i.e. the signal is present) while integer values add up, thus in the following code [BG92]:

```
emit S(2);
Y := ?S
||
emit S(1);
present S then
    X := ?S
end
```

both signal emissions are executed, which makes `S` present and thus both variables `X` and `Y` receive the same value $2 + 1 = 3$. This example also terminates instantaneously with each statement being executed.

2.1.2. Causality

Due to the given behavior of ESTEREL there exist various programs which are not causal, i.e. either have no solution or multiple solutions exist, of which none can be determined uniquely. Examples [BS91] for this are:

```
signal S in
    present S then nothing else emit S end
end
```

Code 1: *ESTERELs implementation of* $x = \neg x$

and

```
signal S1, S2 in
    present S1 then nothing else emit S2 end
    ||
    present S2 then nothing else emit S1 end
end
```

Code 2: *ESTERELs implementation of* $x = \neg y, \quad y = \neg x$

of which the first has no solution (`S` is emitted if and only if `S` is not present), while the second has two solutions where either `S1` is present and `S2` absent or vice-versa. However none of both solutions can be uniquely determined, which makes the second example also non-deterministic. Furthermore, there are ways to achieve infinite loops, which at the same time should terminate instantly, thus also leading to non-causal behavior. An explicit example [BG92] for this just uses the `loop` statement:

```
loop
    X := X + 1
end
```

while an implicit loop [BS91] can be created through:

```
emit S(?S + 1)
```

These examples give the basic idea of how causality can be understood in synchronous programs. However the class of non-causal programs can be different for various languages, which also depend on the concrete semantics of the individual language. While the last two examples show causality problems which are special to ESTEREL given by its timing semantics, the first two examples depict general cyclic causality problems given by their Boolean representation formulas. For these, the aforementioned similarity to combinational circuits can be seen in Figures 2.1 and 2.2, as the respective circuits are directly given by the formulas.

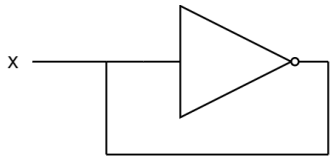

Figure 2.1.: *Circuit for* $x = \neg x$

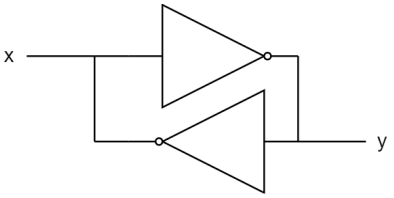

Figure 2.2.: *Circuit for* $x = \neg y, \quad y = \neg x$

2.2. Quartz

Quartz [Sch09] (which is implemented in the Averest framework[1]) is an imperative synchronous language which is based on modern versions of ESTEREL. However the semantics in regard to the synchrony hypothesis are significantly different. In Quartz, time is divided into macro-steps which act as logical time units and are arranged as a set of multiple micro-steps which effectively are the zero-time statements. In difference to ESTEREL, during each macro-step, every immediate statement has to be in fact executed at the same time. That means that the order of the statements inside a macro-step technically does not matter (while still maintaining control-flow of conditionals). This makes Quartz more practical for hardware design reasons, as sequential elements of circuits are modeled by a sequence of macro-steps while each macro-step describes a combinational circuit internally.

Quartz supports the atomic data types `bool` (Booleans), `btv{n}` (bitvectors), `nat{n}` (natural numbers/unsigned integers), `int{n}` (signed integers) and `real` (real numbers/floating point numbers) as well as the composite data types `array(`α`, n)` (array with `n` elements of type α) and the tuple type $\alpha_1 * \cdots * \alpha_n$ consisting of multiple elements of possibly different types. Bitvectors, signed and unsigned integers can be optionally bounded by either the number `n` of bits for bitvectors or the greatest absolute value `n - 1` for signed and unsigned integers. Different to ESTEREL, signals do not exist as separate entity; instead, Boolean variables can be used for signaling events. For this, variables can have different storage types, which are either `mem` for memorized variables or `event` for event/signal variables which reset their value each macro-step.

The core statements of Quartz are shown in Table 2.2.

[1] `http://www.averest.org`

Table 2.2.: *Basic Quartz statements [SB17]*

`nothing`	empty statement
`l:pause`	start/end of macro step
x `=` τ	immediate assignment
`next(`x`) =` τ	delayed assignment
`if(`σ`)` S_1 `else` S_2	conditional
$S_1; S_2$	sequence
`do` S `while(`σ`)`	iteration
$S_1 \| S_2$	synchronous concurrency
`[weak] [immediate] abort` S `when(`σ`)`	preemption: abortion
`[weak] [immediate] suspend` S `when(`σ`)`	preemption: suspension
$\{\alpha\ x; S\}$	local variable x of type α
$inst$: $name(\tau_1, \ldots, \tau_n)$	call of module *name*

Further statements exists, which however can be defined by means of these core statements and thus can be considered as syntactic sugar. Examples for this are `emit(x)` which is simply defined as `x = 1` for a Boolean variable `x` as reminiscence to ESTEREL or `l:await(`σ`)` which is defined as `do l:pause; while(!`σ`)`.[2]

The most important statement for synchrony is the `pause` statement (which was also added to ESTEREL in a later version), as it ends the current macro-step and marks the start of the next macro-step. `pause` statements can optionally have a label `l` which otherwise is generically given by the compiler. With macro-steps being logical units of time, the delayed assignment, which assigns the value in the next macro-step (i.e. after the next `pause` statement), is the only possible way to assign a variable's value by means of itself. That means `x = x + 1` is never a valid statement, as the final value of `x` (which is assigned on the left side) must be equal to the expression on the right side which involves this final value `x`. The correct way to write such an assignment, for example for a loop iteration variable, would thus be `next(x) = x + 1`.

Immediate assignments can be interpreted as equations, while multiple immediate assignments form an equation system, which for causality must have a uniquely determinable solution. This also holds if multiple assignments are done to the same variable, as all those assignment equations must hold at the same time. Thus

```
x = n; x = abs(n);
```

with `abs(n)` being the absolute value function, is not causal for negative numbers `n`. If no assignment is made to a variable during a macro-step and no delayed assignment has been made in the macro-step before, the value of the variable is defined by the so-called "reaction to absence" which is given by the variable's storage type. For memorized variables this is the value that was assigned most recently (or the type's default value if no assignment has been executed since the declaration) while for event variables the value always

[2] `https://es.cs.uni-kl.de/tools/TeachingTools/QuartzRef.pdf`

resets to the default value in this case. Together with these explanations and the implicitly given semantics of operators and data-flow-statements, various semantic problems can be analyzed in the following Section.

2.3. Semantic Problems

Due to the synchrony hypothesis, various problems can appear in syntactically correct synchronous programs as a result of the implemented semantic. With the semantic regarding zero-time assignments being different in Quartz and ESTEREL, some of the problems are only present in one of the languages while others are present in both. A special class of those problems are causality cycles which in principle can appear in every synchronous language, with the trivial example being $x = \neg x$ as Boolean function. As the existence of deterministic solutions may depend on the given inputs, it is not always possible for compilers to statically detect unsolvable cycles. For this reason, they need to be taken care of in more detail.

2.3.1. Infinite Loops

Similar to ESTEREL, also Quartz can have the problem of infinite loops if no `pause` statement is executed inside the loop. However as any assignment inside the loop would thus be executed infinitely often at the same time, each of the assignments (to the same variable) would need to assign the same value for the program to be correct. As this value also needs to be the same in the first execution of the assignment, all other instant iterations of the loop would not change this result, thus making the loop redundant. Additionally, all possible ways to exit the loop (if their are any) are bounded to some condition, which because of the synchrony either holds "the whole time" and thus instantly exits the loop or does never hold, which makes the loop infinite again. For this reason, it makes sense to always require a `pause` statement in each loop or another derived statement which includes a `pause` in its definition (like `await` for example). This problem can be spotted statically by the compiler. It has to be noted that in ESTEREL a conditional exit of an otherwise instantaneous loop is indeed possible as assignments still happen one after each other, which can make the condition true at some point.

2.3.2. Schizophrenia

Schizophrenia is another a problem appearing because of loops, but in this case the problem lies in the reincarnation of a locally declared variable inside the loop when reentering the loop in the same macro-step. An example [Sch09] for this is:

```
module Schizophrenic(event !x0,!x1,!x2,!x3) {
    loop {
        event x;
        if(x) emit(x1); else emit(x0);
        pause;
        emit(x);
        if(x) emit(x3); else emit(x2);
    }
}
```

In this example, `x` is a locally declared event variable which is only emitted in one statement after the `pause`. Thus also `x3` is emitted. However in the same instantaneous step, the loop is left and reentered and further executed until the next `pause` (which is the same statement as before). As the scope of variable `x` is exactly the loop, in the moment when the loop is reentered, the "old" value is not valid anymore as it is outside its scope. Nonetheless, a new local variable with the same name is declared, i.e. reincarnated. The "new" value of the reincarnated variable is by default 0, thus `x0` is emitted as `x` does not hold in this scope. This repeats in each loop iteration which results in both `x0` and `x3` being emitted in each macro-step (except the first in which only `x0` is emitted). If however `x` was declared outside the loop, then `x1` and `x3` would be emitted in each step after the initial emission of `x0`. While this is not exactly a semantic problem as this behavior is well-defined, it opens up a common pitfall when programming in synchronous languages. In ESTEREL for example, this problem exists too [TS04].

2.3.3. Causality

With causality cycles being briefly introduced in Section 2.1.2, the two main examples Code 1 and Code 2 can be directly transferred to Quartz, in which these look like this:

```
module Negate(event !o) {
    if(o) nothing; else emit(o);
}
```

Code 3: *Quartz implementation of* $x = \neg x$

and

```
module Xor(event !o1,!o2) {
    if(o1) nothing; else emit(o2);
    if(o2) nothing; else emit(o1);
}
```

Code 4: *Quartz implementation of* $x = \neg y, \quad y = \neg x$

As Quartz has no signal entity and instead uses Boolean event variables, these programs can be simplified by using Boolean operators:

```
module NegateAlt(event !o) {
    if(!o) emit(o);
}
```

Code 5: *Quartz alternative implementation of* $x = \neg x$

and

```
module XorAlt(event !o1,!o2) {
    if(!o1) emit(o2);
    if(!o2) emit(o1);
}
```

Code 6: *Quartz alternative implementation of* $x = \neg y, \quad y = \neg x$

These alternative implementations finally make the connection to (cyclic) circuits clearly visible, as they are directly representing their Boolean formulas. In contrast to ESTEREL, also any assignment like `x = x + 1` is effectively a causality cycle due to a dependence of the variable `x` to itself. If one would try to solve this cycle by iteratively calculating the result on the right side and applying it on the left side, this would result in a kind of feedback-loop. In electric circuits this would happen analogously, given a slight delay of signal propagation in reality, however physical limits would limit the amplification at some point (which usually is not the wanted behavior anyhow). Two more educated examples [Sch09] are the two programs given in Code 7 and Code 8 (the numbering already refers to the examples given in the Appendix A).

```
module P02(event !o1,!o2) {
    emit(o2);
    if(o1) {
        if(o2) pause;
        emit(o1);
    }
}
```

Code 7: *Quartz implementation of* $x_0 = x_0 \wedge \neg y_0, \quad y_0 = 1, \quad x_1 = x_0 \wedge y_0$

```
module P11(event !o1,!o2) {
    if(o1) {
        emit(o2);
        if(o2) pause;
        emit(o1);
    }
}
```

Code 8: *Quartz implementation of* $x_0 = x_0 \wedge \neg y_0, \quad y_0 = x_0, \quad x_1 = x_0 \wedge y_0$

These programs are nearly identical except the `emit(o2)` statement, which is either inside or outside the conditional in the programs. Due to the given Boolean formulas (the indices indicate the timestep of the according variable), it can be logically deduced that x must be 0 in both timesteps for both programs while y is 1 for program `P02` and 0 for program `P11` in the first step (and 0 in the second). However depending on the concrete definition of

causality, both programs can be either causally correct, incorrect or even `P02` correct and `P11` incorrect. The reason for these possible differences is due to constructiveness of the unique solution.

For `P02` it can be reasoned that as `o1` is emitted in any case and thus there is no possible path left in which `o1` can be emitted during the first step (as the inner conditional is fulfilled). Thus `o1` must be 0 by default which leads to the outer conditional not being fulfilled, making the program causally correct in a constructive sense. In `P11` however, as all emissions happen inside the outer conditional, it is not possible to deduce any value for either `o1` or `o2` until a speculative assumption is made, whether the conditional holds or not. For this, the value of `o1` would need to be guessed initially, which makes the unique solution not constructive.

A more formal reasoning for constructiveness is done by a fixpoint-iteration starting with unknown values, which is explained in Section 3.4. The basis for such fixpoint-iterations is the use of symbolic values (similar to the variables in the Boolean functions) which is explained in the next Section.

2.4. Symbolic Execution

Meaningful programs should in reality almost always be deterministic, except when non-determinism is explicitly required and handled. While programs like Code 5 have no valid solution and thus simply can not be run correctly, other programs like Code 6 have multiple solutions and thus, when implemented in hardware circuits for example, choose one of these stable solutions nondeterministically which is caused by the propagation delay of electric currents, which can not be calculated effectively. Both cases can thus be seen as causally incorrect. As sometimes non-causal behavior only appears for certain inputs, the idea of symbolic execution is to use these input variables and calculate all internal and output variables as functions of the input variables.

Causality can then be defined as certain conditions that must be fulfilled by these variables, which include determinism and uniqueness (for each set of input variables). Simple causality could be achieved by transforming the variable expressions to Boolean formulas and using a SAT-solver to verify that the resulting set of Boolean formulas has a unique solution. However as seen in the examples of Code 7 and Code 8, while both have unique solutions which can be computed with the Boolean formulas, the solution of the second program is not constructive.

2.4.1. SOS Rules

The SOS (short for Structural Operational Semantics) rules, which are similar to deduction proof rules, are a way to define the semantics of a given language formally. As Quartz is a synchronous language, not all variables are directly known while evaluating the program in regard to these rules (even without symbolic execution). To evaluate the semantics in regard to this problem, the SOS rules are divided into two categories, namely the SOS reaction rules

which try to complete the environment of the current macro-step (i.e. one reaction), which can then be used for the SOS transition rules to verify the consistency of the environment and calculate delayed assignments as well as the residual statement for the next macro-step [Sch09]. In this context, the environment $\mathcal{E}$ is a function mapping all variables to their values. If a value is unknown, this is denoted by the value $\perp$. In the original paper, the definition for $\mathcal{E}$ is more complicated as it also takes care of the schizophrenia problem which was discussed in Section 2.3, together with another function $\hbar$ which takes care of the incarnation index. For simplification reasons, this is omitted here. An environment is called incomplete if there exist variables which have the value $\perp$, however this does not necessarily mean that a transition can not be computed. This is due to lazy evaluation rules like $\perp \vee 1 = 1$. As the list of SOS rules is rather large, even for only the core statements of Quartz, only some examples are given here; the full list is documented in [Sch09].

SOS Reaction Rules

A (simplified) SOS reaction rule is defined as:

$$\langle \mathcal{E}, S \rangle \looparrowright \langle \mathcal{D}_{\text{must}}, \mathcal{D}_{\text{can}}, t_{\text{must}}, t_{\text{can}} \rangle$$

where, given the current environment $\mathcal{E}$ and a statement S, the reaction are the sets of actions $\mathcal{D}$ that must or can be executed and the Boolean values t that state if the statement must or can be instantaneous. Logically $\mathcal{D}_{\text{must}} \subseteq \mathcal{D}_{\text{can}}$ and $t_{\text{must}} \rightarrow t_{\text{can}}$ must hold. Some examples for those rules are:

$$\langle \mathcal{E}, \texttt{x = } \tau \rangle \looparrowright \langle \{(x = \tau)\}, \{(x = \tau)\}, 1, 1 \rangle$$

$$\langle \mathcal{E}, \texttt{next(x) = } \tau \rangle \looparrowright \langle \{\}, \{\}, 1, 1 \rangle$$

$$\langle \mathcal{E}, \texttt{l:pause} \rangle \looparrowright \langle \{\}, \{\}, 0, 0 \rangle$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = 1 \quad \langle \mathcal{E}, S_1 \rangle \looparrowright \langle \mathcal{D}^1_{\text{must}}, \mathcal{D}^1_{\text{can}}, t^1_{\text{must}}, t^1_{\text{can}} \rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{) } S_1 \texttt{ else } S_2 \rangle \looparrowright \langle \mathcal{D}^1_{\text{must}}, \mathcal{D}^1_{\text{can}}, t^1_{\text{must}}, t^1_{\text{can}} \rangle}$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = \perp \quad \langle \mathcal{E}, S_1 \rangle \looparrowright \langle \mathcal{D}^1_{\text{must}}, \mathcal{D}^1_{\text{can}}, t^1_{\text{must}}, t^1_{\text{can}} \rangle \quad \langle \mathcal{E}, S_2 \rangle \looparrowright \langle \mathcal{D}^2_{\text{must}}, \mathcal{D}^2_{\text{can}}, t^2_{\text{must}}, t^2_{\text{can}} \rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{) } S_1 \texttt{ else } S_2 \rangle \looparrowright \langle \mathcal{D}^1_{\text{must}} \cap \mathcal{D}^2_{\text{must}}, \mathcal{D}^1_{\text{can}} \cup \mathcal{D}^2_{\text{can}}, t^1_{\text{must}} \wedge t^2_{\text{must}}, t^1_{\text{can}} \vee t^2_{\text{can}} \rangle}$$

It has to be noted that delayed assignments have no effect for the current macro-step, thus no direct reaction is provided. The reaction rules can be used together with a fixpoint-analysis to complete the environment if possible, which starts with all input variables known, as well as all delayed assignments from the previous macro-step, while all other local and output variables are not known initially and thus have the value $\perp$. Details can be found in [Sch09].

SOS Transition Rules

As soon as the environment is complete or as complete as possible after the fixpoint-iteration, the SOS transition rules can be used to verify the environment, i.e. if no rule fits (for example if the condition of a conditional is still unknown), the current macro-step is not causal and thus the evaluation can not continue. If the environment passes this verification, the delayed assignments as well as the residual statement for the next macro-step are computed in parallel, which can then be used for the next reaction. A (simplified) SOS transition rule is formulated as

$$\langle \mathcal{E}, S \rangle \twoheadrightarrow \langle S', \mathcal{D}, t \rangle$$

where S is the initial statement, S' the residual statement for the next macro-step, $\mathcal{D}$ a set of pairs (x, v) where v is a value of the same type as x and finally t a Boolean value denoting if the execution of the current SOS rule is instantaneous. $\mathcal{D}$ specifically denotes the value pairs of all delayed assignments such that they can be applied before the next macro-step. Some examples for SOS transition rules are:

$$\frac{[\![x]\!]_{\mathcal{E}} = [\![\tau]\!]_{\mathcal{E}}}{\langle \mathcal{E}, \texttt{x = } \tau \rangle \twoheadrightarrow \langle \texttt{nothing}, \{\}, 1 \rangle}$$

$$\langle \mathcal{E}, \texttt{next(x) = } \tau \rangle \twoheadrightarrow \langle \texttt{nothing}, \{(x, [\![\tau]\!]_{\mathcal{E}})\}, 1 \rangle$$

$$\langle \mathcal{E}, \texttt{l:pause} \rangle \twoheadrightarrow \langle \texttt{nothing}, \{\}, 0 \rangle$$

$$\frac{[\![\sigma]\!]_{\mathcal{E}} = 1 \qquad \langle \mathcal{E}, S_1 \rangle \twoheadrightarrow \langle S_1', \mathcal{D}_1, t_1 \rangle}{\langle \mathcal{E}, \texttt{if(}\sigma\texttt{) } S_1 \texttt{ else } S_2 \rangle \twoheadrightarrow \langle S_1', \mathcal{D}_1, t_1 \rangle}$$

In this case there is no rule for the case that σ evaluates to $\bot$, thus in this case the evaluation would stop, as the correct transition can not be computed because the condition is not known. As with the SOS reaction rules, a complete list can be found in [Sch09].

2.4.2. Transitions to Extended Finite State Machines

As the SOS rules are complicated to use with symbolical values directly, due to there being no way for them to decide whether a symbolic expression $x \leq y$ for symbolic input variables x and y is true, the SOS rules are instead indirectly used to transform a given program to an extended finite state machine (EFSM) with guarded actions. The time is then represented directly by the states of the EFSM, as each state represents exactly one possible macro-step, while the transitions between the states are guarded by transition conditions which model the control-flow between different macro-steps. Each state is represented by a `pause` statement from which the macro-steps starts (with the initial state being an exception), thus the transition conditions are exactly the conditions that need to hold to reach the successor `pause` statement when starting at an initial `pause` statement. The same reasoning can be used to create guarded actions for all actions (i.e. [delayed] assignments, asserts, ...).

That means the conditions that must be fulfilled to reach the relevant action starting from the `pause` statement of the current state, form the guard's Boolean expression of the guarded action. A simple example for this is the program in Code 9, which gets transformed to the EFSM given in Figure 2.3.

```
module EFSMExample(int ?a,?b,x) {
    if(a == 3)
        x = 4;
    else
        x = a + 5;
    l1:pause;
    if(b <= 4)
        l2:pause;
    x = b - 3;
}
```

Code 9: *Exemplary Quartz program to demonstrate the EFSM transformation*

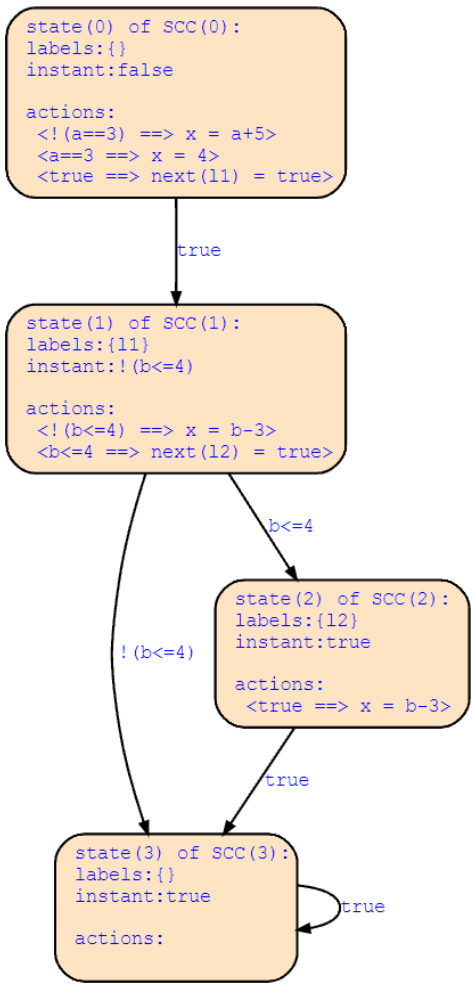

Figure 2.3.: *EFSM of Code 9 with sink state*

It can be seen there are four states of which one is the initial state and thus has no label and one is an additional sink state which signals the termination of the program. For the first conditional, each of both branches has exactly one action, thus one guarded action is created in the initial state for each branch. In either case, the `pause` statement labeled `l1` is reached and thus an implicit guarded action for a delayed assigned to the label-variable is additionally created (where the guard is always `true`). The same holds for the transition condition which goes from the initial state to the state labeled `l1`, which is the start of the next macro-step. The second conditional either leads to another `pause` statement labeled `l2` or directly terminates the program after an additional assignment. Thus this assignment has the same guard as the transition condition to the sink state. In case the conditional branch is taken, the transition leads to a third state which is labeled `l2`. Starting from this state, the program now definitely terminates together with the last assignment, thus making the guard of the guarded action as well as the transition condition `true` . This instantaneous termination is also signified by the `instant` parameter of the EFSM, which for the second state however is guarded by a condition.

If a program contains a loop, the extended finite state machine similarly also has a loop or a greater cycle if multiple `pause` statement are contained in the loop. Other statements like `abort` have slightly more complex transformations, nonetheless all Quartz program can be transformed to an EFSM by applying the relevant transformation rules which however are not further discussed here. With each state of the EFSM representing a macro-step, the EFSM acts as a basis for the symbolic analysis tool which is explained in Chapter 4.

2.4.3. Three/Four-Valued Logic

In Subsection 2.4.1 the use of the value $\perp$ is introduced, which stands for a currently unknown value. Together with the two original Boolean values 0 and 1, this forms the three-valued logic, which makes use of the lazy-evaluation rules to compute expressions, even if some variables in the expressions are still unknown. This logically only works because the final values of the currently unknown variables do not change the result of an expression, even if these variables' values become known at a later point. This simplification of evaluation however does not change the fact that in a causally correct program all variables should become known at some point for each macro-step.

On the other side, using $\perp$ as a symbolical value that does not interfere with any concrete value, makes it possible to interpret guarded actions with immediate assignments as guarded assignments. That means any guarded, immediate assignment

$$\langle \varphi \implies x = \tau \rangle$$

with guard φ can be interpreted as the assignment

$$x = \langle \varphi \implies \tau \rangle$$

where $\langle \varphi \implies \tau \rangle$ evaluates to τ if $\varphi = 1$ and otherwise evaluates to $\perp$. If condition φ would not hold, the assignment would thus effectively be $x = \perp$.

Table 2.3.: *Truth tables of $\neg$, $\wedge$ and $\vee$ in four-valued logic*

x	$\neg x$
$\bot$	$\bot$
0	1
1	0
$\top$	$\top$

$\wedge$	$\bot$	0	1	$\top$
$\bot$	$\bot$	0	$\bot$	$\top$
0	0	0	0	$\top$
1	$\bot$	0	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

$\vee$	$\bot$	0	1	$\top$
$\bot$	$\bot$	$\bot$	1	$\top$
0	$\bot$	0	1	$\top$
1	1	1	1	$\top$
$\top$	$\top$	$\top$	$\top$	$\top$

As a symbolic placeholder, this assignment can then just be discarded if a concrete value is assigned to x by another statement. This interpretation is used later in Section 3.2.2 to define the Trigger-Operator which in turn is then used in Section 4.3 to exactly do this transformation from guarded actions to guarded assignments. The reasoning that $\bot$ should not interfere with the concrete values is also defined more formally in Section 3.2 by extending the values to a complete lattice, in which the supremum is used as an interference measure.

Having interference in mind, another symbolic value $\top$ can be introduced to denote the final value of a variable which has been assigned multiple different (concrete) values. Including $\top$ into the range of possible values, the four-valued logic is formed. However as $\top$ represents the fact that a variable has no uniquely determined value, any further interaction with this value does not fix this error and thus this value persists. This is especially important for the definition of the Boolean operators which include the two symbolic values. The truth tables for the logical negation, conjunction and disjunction are given in Table 2.3.

The same expansion can be made for non-Boolean types, however in these cases this does not necessarily result in the simplification of any operations, as these types may not have any lazy-evaluation rules at all. Nonetheless, the use of the two additional symbolic values still enables the reduction of guarded actions to guarded assignments.

2.4.4. Equation Systems

As already stated in Section 2.2, all immediate assignments can be interpreted as single equation in an equation system that must be fulfilled. However, also all delayed statements from the previous macro-step have to be added to complete the equation system, as their values must not interfere with the current assignments too. With the left side of all equations containing only one variable, the right sides can be grouped together for each variable on the left side, forming a set of values or expression of values that must be equal.

In Subsection 2.4.2, the EFSM together with guarded actions were introduced. These guarded actions can be transformed to guarded assignment by using four-valued logic. If this grouping is now done for the guarded assignment equations, the right side of the grouped equation consists of a set of guarded values, which all need to be equal or $\bot$ for the assigned value to be unique (either a concrete value or $\bot$ if none of the guards hold). Using the

supremum on this set is the final operation that must be performed to only have one equation for every variable (which still contains cyclic dependencies to other variables or itself). If one of those equations eventually evaluates to $\bot$ or $\top$ at any time, a causality issue is found, thus making the currently examined macro-step causally incorrect.

The resulting equation system are then put into an SMT-solver, which, given the input variables as freely tweakable parameters can then decide if there is a constructive solution for this equation system or not.

Many causal problems can already be described by means of only Boolean event variables which can only be emitted (i.e. not explicitly set to 0). In this case, each equation for such a variable can be written as disjunction of all the different conditions that lead to an `emit` statement (which is similar to guarded actions). This can be either done separately for each macro-state or combined in a single equation by using label variables to annotate the macro-states in which the according conditions lead to an `emit` statement through a conjunction. For the program in Code 8, with `st` being the initial label and `l` the `pause` label, the equation system would then look like this [Sch20]:

$$o_1 = \texttt{st} \wedge o_1 \wedge \neg o_2 \ \vee \ \texttt{l}$$
$$o_2 = \texttt{st} \wedge o_1$$

For these special cases, constructive causality can be directly analyzed on these equations by using a fixpoint-iteration starting at $o_1 = o_2 = \bot$. Only one label may be active at a time, thus $\texttt{st} \wedge \texttt{l}$ must be 0 in any case. If the fixpoint still contains uncertainties (i.e. $\bot$ or $\top$ elements), the program is not constructively causal. This is implicitly used to reason if programs should be considered constructive for the Examples A.

2.5. Satisfiability Modulo Theories

When working with Boolean variables, the equations from the equation systems can be transformed into Boolean expressions, as the check for equality can be implemented by the logical equivalence operator. This effectively results in a set of Boolean expressions which need to be checked for solvability for each macro-step, which can be done by any SAT-solver. However by using this approach, more programs would be deemed causal than expected if the generation of any solution should be constructive (see the example of Code 7 and Code 8). Thus for the constructive causality, a fixpoint analysis must be used at some point to guarantee that any solution, if it uniquely exists, can also deterministically be found without nondeterministic guessing of variable's values.

Other than that, as soon as numeric variables are used, a sole Boolean verification using SAT-solvers will not work anymore (although in theory numeric values could be reduced to their Boolean encoding which however is not useful when working with this high-level representation). For this reason, solvers of the satisfiability modulo theories (SMT) type will be used, which extend the

solvability problem to mathematical formulas. A simple example regarding this question would be to determine if the formula $x \cdot x = 9$ is solvable for any integral number x (which it is for $x \in \{-3, 3\}$). On the other side, the formula $x \cdot x = 8$ has no integral solution but real ones (i.e. $x = \pm\sqrt{8}$). For this thesis, the SMT-solver Z3 will be used.

2.5.1. Z3

Z3[3] is a theorem prover from Microsoft Research which is freely available under the MIT license. Although Z3 supports the common SMT-LIB(2) standard [BFT17], it also provides bindings for various programming languages, which includes the .NET framework and can thus directly be used with F#. While the general formulas that need to be verified are explained in Chapter 4, the concrete translation from these formulas to be used in Z3 are explained in Chapter 5.1.

[3] `https://github.com/Z3Prover/z3`

3. Formal Definitions

In synchronous languages each action in the same macro-step is executed at the same time, which especially includes immediate assignments. Beside the common problems stated in Section 2.3, this can also mean that one variable is assigned multiple values in different statements inside one single macro-step. While doing a causality analysis it must be ensured that for each variable either only one value is assigned at a time (for example due to an `if-else`-statement) or that the multiple assignments evaluate to the same value. On the other hand it must also be ensured that each variable has a defined value in each macro-step, especially if there is neither an immediate assignment in the current macro-step nor a delayed assignment in the previous macro-step. This chapter provides the formal definitions which are used to implement the symbolic execution.

3.1. Types

For this thesis only Boolean and numeric types are considered in detail, as arrays, tuples and bitvectors are effectively solely collections of these basic types which extend the common type operations onto their respective elements. In regard to this, it has to be stated that the numeric types are internally represented as bitvectors too; but with the difference that for the symbolic analysis these bitvectors are directly interpreted as a certain number with respect to a predefined encoding (for example signed or unsigned radix-2). That is the reason why numeric types are included in the detailed analysis. These four types will be denoted as:

$\mathbb{B} := \{0, 1\}$	for Boolean Type
$\mathbb{N} := \{0, 1, 2, \dots\}$	for Natural Type
$\mathbb{Z} := \{\dots, -1, 0, 1, \dots\}$	for Integer Type
$\mathbb{R}$	for Real Type (according to IEEE 754)

For the symbolic analysis these types have to be extended to all expressions that evaluate to the relevant type when inserting values to the used variables. Thus the following expression type $\mathbb{B}^*$ is recursively defined for the Boolean

type $\mathbb{B}$:

$$\begin{aligned}
& & \mathbb{B} &\subseteq \mathbb{B}^* \\
a \text{ is a Boolean variable} &\Rightarrow & a &\in \mathbb{B}^* \\
x \in \mathbb{B}^* &\Rightarrow & \neg x &\in \mathbb{B}^* \\
x, y \in \mathbb{B}^* &\Rightarrow & (x\ \mathbf{op}_{\mathbb{B}}\ y) &\in \mathbb{B}^* \\
n_1, n_2 \in \mathbb{N}^* &\Rightarrow & (n_1\ \mathbf{pred}\ n_2) &\in \mathbb{B}^* \\
n_1, n_2 \in \mathbb{Z}^* &\Rightarrow & (n_1\ \mathbf{pred}\ n_2) &\in \mathbb{B}^* \\
n_1, n_2 \in \mathbb{R}^* &\Rightarrow & (n_1\ \mathbf{pred}\ n_2) &\in \mathbb{B}^* \\
x, y, z \in \mathbb{B}^* &\Rightarrow & (x\ ?\ y : z) &\in \mathbb{B}^* \quad \text{(if-then-else)}
\end{aligned}$$

where $\mathbf{op}_{\mathbb{B}} := \{\vee, \wedge, \Rightarrow, \Leftrightarrow\}$ are the common Boolean operators and $\mathbf{pred} := \{=, <, \leq\}$ are the predicate symbols for numeric types. For $\mathbb{N}$ the expression type $\mathbb{N}^*$ is defined as:

$$\begin{aligned}
& & \mathbb{N} &\subseteq \mathbb{N}^* \\
a \text{ is a natural variable} &\Rightarrow & a &\in \mathbb{N}^* \\
n_1, n_2 \in \mathbb{N}^* &\Rightarrow & (n_1\ \mathbf{op}_{\mathbb{N}}\ n_2) &\in \mathbb{N}^* \\
x \in \mathbb{B}^*, n_1, n_2 \in \mathbb{N}^* &\Rightarrow & (x\ ?\ n_1 : n_2) &\in \mathbb{N}^* \quad \text{(if-then-else)}
\end{aligned}$$

with $\mathbf{op}_{\mathbb{N}} := \{+, -, \cdot, \div, \%\}$. Analogous definitions hold for $\mathbb{Z}^*$ and $\mathbb{R}^*$. In the following all of those expression types are bundled as $\mathbf{T}^* := \{\mathbb{B}^*, \mathbb{N}^*, \mathbb{Z}^*, \mathbb{R}^*\}$ and the sole value types as $\mathbf{T} := \{\mathbb{B}, \mathbb{N}, \mathbb{Z}, \mathbb{R}\}$. Equality over expression types is to be understood semantically and not syntactically, as $0 \vee 1$, $\neg 0$, $1 \wedge 1$ are representing the same elementary value 1 in $\mathbb{B}^*$. If expressions contain variables, equality may depend on the actual values of these variables. For example $a \cdot a$ is equal to 4 for $a \in \{-2, 2\}$ in $\mathbb{Z}^*$ and not equal to 4 for any other value of a, while $(a + b) \cdot 2$ is always equal to $(2 \cdot a) + (2 \cdot b)$ for arbitrary values $a, b \in \mathbb{Z}$.

3.2. New Operators

The aforementioned problem of multiple variable assignments can be handled by using complete, bounded lattices defined over each type. Each complete lattice $(L', \sqsubseteq)$ with $L' = L \cup \{\bot, \top\}$ contains all the elements of the respective type ($L \in \mathbf{T}$) as well as a bottom element ($\bot$) and a top element ($\top$) which represent an unknown value and a faulty value respectively. The lattice has the order relations:

$$\bot \sqsubseteq x\ \forall x \in L \qquad \text{and} \qquad x \sqsubseteq \top\ \forall x \in L$$

and thus defines $\sup(a, b)$ and $\inf(a, b)$ as:

$$\sup(a, b) = \begin{cases} \bot, & \text{if } a = \bot \wedge b = \bot \\ \top, & \text{if } a = \top \vee b = \top \vee \\ & \quad a, b \notin \{\bot, \top\} \wedge a \neq b \\ a, & \text{else} \end{cases} \qquad \text{for } a, b \in L'$$

$$\inf(a,b) = \begin{cases} \top, & \text{if } a = \top \wedge b = \top \\ \bot, & \text{if } a = \bot \vee b = \bot \vee \\ & \quad a, b \notin \{\bot, \top\} \wedge a \neq b \\ a, & \text{else} \end{cases} \qquad \text{for } a, b \in L'$$

This effectively describes an equality-relation where the supremum of two type values $x, y \in L$ is $\top$ if and only if the values are not equal. For the Boolean- and the Natural-type the lattices are shown in Figure 3.1:

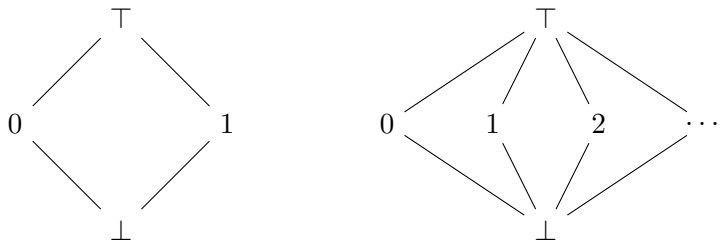

Figure 3.1.: *Boolean Lattice and Natural Number Lattice*

With these lattices the following two operators can be introduced.

3.2.1. Sup-Operator

This operator operates on a multiset of values over L' for a certain type and evaluates to the supremum of its values:

$$\mathrm{Sup}(M) := \begin{cases} \top, & \text{if } \exists x \in M.\ x = \top \vee \\ & \quad \exists x, y \in M.\ x, y \notin \{\bot, \top\} \wedge x \neq y \\ x', & \text{if } \exists x' \in M.\ x' \notin \{\bot, \top\} \wedge \\ & \quad \forall x \in M.\ x \neq \top \wedge \\ & \quad \forall x, y \in M.\ x, y \notin \{\bot, \top\} \Rightarrow x = y \\ \bot, & \text{else} \end{cases}$$

$$\text{with } M = \{x_1, x_2, \dots\}, \quad x_i \in L'$$

The Sup-Operator thus extends the supremum to an arbitrary number of values of the same type.

3.2.2. Trigger-Operator

The Trigger-Operator is defined as:

$$(\varphi \Mapsto x) := \begin{cases} x, & \text{if } \varphi = 1 \\ \top, & \text{if } \varphi = \top \\ \bot, & \text{else} \end{cases} \qquad \text{for } \varphi \in \mathbb{B}'\ x \in L'$$

and thus behaves as a guarded value x, which is only adopted if the Boolean guard φ is true.

3.2.3. Extension to Expression Types

As stated in Section 3.1, symbolic analyses do not only use concrete values but also typed variables and expressions over these variables. Because of that, these two new operators as well as $\bot$ and $\top$ must also be included in each expression type $L^* \in \mathbf{T}^*$, i.e.

$$\begin{aligned} & & \{\bot, \top\} \subseteq L^* \\ M \subseteq L^* & \Rightarrow & \mathrm{Sup}(M) \in L^* \\ \varphi \in \mathbb{B}^*, x \in L^* & \Rightarrow & (\varphi \Rightarrow x) \in L^* \end{aligned}$$

Furthermore, all operators and other expressions operating on the original types must be extended to include the $\bot$ and $\top$ symbols (and expressions over those). The extended semantics for Boolean operators are described in Section 2.4.3, while every numeric operation or predication reduces to $\bot$ or $\top$ if either of its operands is one of those elements, i.e.

$$\begin{aligned} &(x\ \mathbf{op}_{\mathbb{N}}\ \top) = (\top\ \mathbf{op}_{\mathbb{N}}\ x) = (x\ \mathbf{pred}\ \top) = (\top\ \mathbf{pred}\ x) = \top \quad \text{for all } x \in \mathbb{N}^* \\ &(x\ \mathbf{op}_{\mathbb{N}}\ \bot) = (\bot\ \mathbf{op}_{\mathbb{N}}\ x) = (x\ \mathbf{pred}\ \bot) = (\bot\ \mathbf{pred}\ x) = \bot \quad \text{for all } x \in \mathbb{N}^* \text{ if } x \neq \top \end{aligned}$$

Analogously, the same holds for the integer and real type.

The if-then-else-operator has a special role in this regard:

$$(\varphi\ ?\ x : y) := \begin{cases} \top, & \text{if } \varphi = \top \\ \bot, & \text{if } \varphi = \bot \wedge x \neq y \\ x, & \text{if } \varphi = 1 \\ y, & \text{else} \end{cases} \qquad \text{for } \varphi \in \mathbb{B}^*,\ x, y \in L^*$$

which means that the result of the operation is determined directly by the condition φ, except if it is $\bot$, as the result then depends on the equality of the possible cases. If they are equal, it is irrelevant that the condition φ has no concrete value (yet), as the result will be the same nonetheless; if they are not equal though, the final result cannot be determined and thus reduces to $\bot$. It has to be noted that a faulty precondition ($\top$) conveys this status as it indicates an error during the computation. In contrast to that if x is $\top$, this value is only relevant if $\varphi = 1$ as otherwise this part of the operation is actually not computed and thus the indicated error does not happen. A simple example for this is $(\neg(a = 0)\ ?\ 10 \div a\ :\ a)$ where $10 \div a$ is $\top$ for $a = 0$ due to division by zero, but in this case $\neg(a = 0)$ is false and thus $10 \div a$ is not computed.

3.3. Operator Rules

In Section 4.3, each complete variable assignment will be constructed by a Sup-operation over multiple Trigger-operations. If any further operation uses one of those variables, at least one operand will thus be a Sup-expression. As in theory the multiset M in $\mathrm{Sup}(M)$ may be arbitrary large, working

with a Sup-expression as operand can get quite complicated. For this reason, the following distributive rules can be formulated for the two new operators, which are correct if no inner operand $\mathrm{Sup}(M)$ is $\top$ (for better readability Sup parentheses are omitted):

$$\begin{aligned}
\neg\mathrm{Sup}\{a,b\} &\Leftrightarrow \mathrm{Sup}\{\neg a,\neg b\}\\
\mathrm{Sup}\{a,b\}\ \mathbf{op}_{\mathbb{B}}\ c &\Leftrightarrow \mathrm{Sup}\{a\ \mathbf{op}_{\mathbb{B}}\ c, b\ \mathbf{op}_{\mathbb{B}}\ c\}\\
(x \Rrightarrow \mathrm{Sup}\{a,b\}) &\Leftrightarrow \mathrm{Sup}\{(x \Rrightarrow a),(x \Rrightarrow b)\}\\
(\mathrm{Sup}\{a,b\} \Rrightarrow x) &\Leftrightarrow \mathrm{Sup}\{(a \Rrightarrow x),(b \Rrightarrow x)\}\\
\mathrm{Sup}\{a,b\}\ \mathbf{op}_{\mathbb{N}}\ c &\Leftrightarrow \mathrm{Sup}\{a\ \mathbf{op}_{\mathbb{N}}\ c, b\ \mathbf{op}_{\mathbb{N}}\ c\}\\
(x \Rrightarrow y)\ \mathbf{op}_{\mathbb{N}}\ z &\Leftrightarrow (x \Rrightarrow y\ \mathbf{op}_{\mathbb{N}}\ z)\\
(x \Rrightarrow y)\ \mathbf{op}_{\mathbb{N}}\ (a \Rrightarrow b) &\Leftrightarrow (x \wedge a \Rrightarrow y\ \mathbf{op}_{\mathbb{N}}\ b)
\end{aligned}$$

Additionally, the associative rule

$$\mathrm{Sup}\,\{x_1, x_2, x_3, \dots\} = \mathrm{Sup}\,\{x_1, \mathrm{Sup}\,\{x_2, x_3, \dots\}\}$$

holds, which makes it possible to extend the previous distributive rules to arbitrary large sets M.

Although the restriction that $\mathrm{Sup}(M)$ must not be $\top$ is rather large, the use of those rules is still useful for the symbolic analysis. By calculating the symbolic expressions stepwise as seen in Chapter 4.4, as soon as an expression evaluates to $\top$ for a specific initial variable assignment, there is an error in the program and thus any further computation will have undefined behavior. With this reasoning and under the assumption that no computation is faulty until this point (i.e. no operand expression can evaluate to $\top$), these rules may be applied to simplify the operation expression.

3.4. Handling Cyclic Dependencies

As stated in Section 2.3, immediate assignments in synchronous languages can lead to cyclic dependencies, which have to be considered during the symbolic analysis. One way to deal with this problem is to forbid cyclic dependencies completely and report an error if a cycle is detected at any point in the program. This however leads to the rejection of many causally correct programs, for example consider the program:

```
module Together (event x, y) {
    if(x) emit(y);
    if(y) emit(x);
    emit(x);
}
```

where both event variables x and y depend on each other, forming a dependency cycle which would be rejected with this approach. This program is causally correct however, as the unconditional `emit(x)` leads to the emitting of y, conversely leading to another emitting of x which does not contradict the

previous assignment. Thus the final result is $x = y = 1$ which can be deduced by iterative computation.

3.4.1. Boolean Type

For the Boolean Type it is possible to solve certain cyclic dependencies, as the set of possible values is finite and thus a fixpoint computation can be used. For this, each possible macro-step of the program is transformed to an equation system, such that each (non-input) variable is on the left side of exactly one equation with a Boolean function of all variables on the right side. Each Boolean function defines the cases in which 1 is assigned to relevant variable and therefore the negation of this function defines the cases where 0 is assigned. If the variable is an event variable, the negation also defines the cases in which no assignment is made, as the default value then is also 0.

Starting with the first iteration, all non-input variables are assumed to be unknown at the beginning (i.e. $\bot$). In the next iteration, the value of each variable is applied into each right side function of the equation system. The new values of the variables on the left side are then the (simplified) expressions on the right side. These expression values are then applied to the right side functions in the equation system again and so on, until a fixpoint is reached. Due to finiteness (also of Boolean functions), a fixpoint will in fact be reached and thus determine if the program is causal, which is the case if all variables have in fact a Boolean value which is neither $\bot$ nor $\top$. The reason that Boolean values can be inferred from $\bot$ starting values even if no input variables are present, is due to the fact that Boolean formulas like $x \vee 1$ or $x \wedge 0$ can be computed even if x is $\bot$ as its value is not relevant for the result. However, formulas like $x \vee \neg x$ which would evaluate 1 in classic logic, will still remain $\bot$ in the fixpoint iteration which is correct, as the value is not causally deductible.

As an example, consider the equation system for the module `Together` from above:

$$x = y \vee 1$$
$$y = x$$

The fixpoint-iteration is computed as

step	0	1	2	3
x	$\bot$	$\bot \vee 1 = 1$	1	1
y	$\bot$	$\bot$	1	1

and hence finds a fixpoint after 3 steps. Another example for the non-causal equation system $x = x \vee \neg x$ would be the following program

```
module Anyway (event x) {
    if(x) emit(x);
    if(!x) emit(x);
}
```

for which, although $x = 1$ would be a feasible assignment, there is no reason to initially assume that $x = 1$ in the first place. The assumption $x = 1$ is justified only after the condition that $x = 1$ must hold, whereas the assumption $x = 0$ leads to a contradiction.

3.4.2. Numeric Types

In comparison to the Boolean type, the set of all possible numeric values is theoretically infinite and only practically limited by the used encoding. For this reason, a fixpoint iteration would not work. Besides that, except some edge cases like $x \cdot 0$ which classically evaluates to 0, no operation can be evaluated without both operands having a concrete value. Additionally, as soon as there is a kind of self-dependency like $x = x + 4$ there is no numeric value that can fulfill this equation, however this can not even be directly deducted, as there is no syntactic reason that such a number can not exist, but only a semantic reason. Last but not least, while a Boolean variable may act as a signal which is stable after evaluating multiple dependencies, there is no meaningful interpretation of what a cyclic dependency of numeric variables should causally denote, even if a fixpoint iteration converges to a concrete value when starting with an arbitrary value.

Due to these multiple reasons, any cyclic dependency between numeric variables is interpreted as causally incorrect and thus any program that introduces such a cycle is deemed as not causal too.

4. Tool Description

Based on the formal definitions in the previous chapter, the following chapter provides the details of the tool's implementation in regard to the symbolic execution and verification with an SMT-solver. After explaining the terminus "variable" together with its different meanings in this context (Section 4.1), the construction of the extended finite state machine is explained (Section 4.2). For every state in the EFSM, each variables is assigned an expression that is dependent on the input variables and potentially the previous value of the variable, which can be evaluated if these unknown values are given (Section 4.3). Furthermore, helper-variables for each variable get initialized which are used to simplify the verification procedure and to distinguish immediate and delayed assignments (Section 4.3.2). With these presets, the symbolic analysis procedure using a depth-first-search can finally be explained in Section 4.4, which includes variable substitutions and the subsequent causality checks. It needs to be noted that certain keywords are ambiguous due to them being programmatic concepts that are used in the analyzed language Quartz as well as in the language F# in which the symbolic analysis tool (and also the Averest framework that implements Quartz) is implemented. This explicitly includes the keywords "variable", "type" and "value".

4.1. Variables

In the context of imperative programming languages, it may seem clear that variables are abstract identifiers that represent a given value during the computation of a given program. In typed languages each variable is also bound to a certain type which defines the interpretation of the underlying data as typed value. In synchronous languages this is not necessarily different, but for the symbolic analysis it is necessary to differentiate between different components of a variable.

Each variable is identified by a name, which most commonly is a human-readable string, but can also be more abstract. Averest uses the union-type `QName` which for example also allows derived or incarnated names. Further, each variable must have a declaration, which defines its behavior. For this, Averest uses the record-type `Decl` containing the data type (`Qbool`, `Qnat`, `Qint`, ...), the storage type (`Event`, `Memorized` or `Continuous`) and the way of data flow (`Input`, `Output`, `Label`, ...). Last but not least, each variable has a value which should be of the declared data type.

In terms of symbolic analysis, this value is actually an expression of the relevant type, according to Section 3.1. Beside the expressions directly provided by Averest, this thesis will also make use of type-expressions using the

Sup-Operator and Trigger-Operator as well as using the symbolic $\bot$ and $\top$ values of the extended lattice to denote unknown/undetermined and faulty values respectively. In order to realize this, new types are introduced.

4.1.1. Type `extendExpr`

The type `extendExpr` is a union-type consisting of `Expr` from Averest, `GrdExpr = BoolExpr * extendExpr` as the Trigger-Operator and `Sup = Set<extendExpr>` as the Sup-Operator. With this, `extendExpr` is therefore the extension of the whole expression set to include the two new operators. As these are not native in Averest, the new operators may only appear on the outside of an extended expression (i.e. an Averest `Expr` can not contain either of those in its inner scope). Because of that, the operator rules introduced in Section 3.3 would need to be used to pull all instances of the new operators to the outside when creating a new extended expression with other extended expressions as operands. However, this problem can be solved alternatively by using the following symbolic type, which, besides interpreting the new operators by Averest expressions directly, also incorporates the $\bot$ and $\top$ symbol from the extended lattice.

4.1.2. Type `Ecausal`

The symbolic type `Ecausal` is a record of `BoolExpr` for its `isBot` and `isTop` identifiers, as well as `Expr` for a `cnVal` identifier, representing an extended expression including $\bot$ and $\top$ as stated before. While the Boolean expressions denote the cases in which the represented (extended) expression is $\bot$ or $\top$ respectively, `cnVal` is the value of the expression which holds if the expression is causal, i.e. neither $\bot$ nor $\top$. If for any operation the `Ecausal`s of its operands are known, the `Ecausal` of the operation can be constructed by applying the operator on the operand's `cnVal`s to get the `cnVal` of the operator, while the Boolean expressions for `isBot` and `isTop` are created by disjunction and conjunction of the cases for which this operation evaluates to $\bot$ or $\top$. As an example, for the Trigger-Operator ($\varphi \Rightarrow x$) (see Section 3.2.2) this looks like this:

$$\begin{aligned}
&\text{isBot} := \varphi.\text{isBot} \ \vee \ \varphi.\text{cnVal} = 1 \wedge x.\text{isBot} \ \vee \ \varphi.\text{cnVal} = 0\\
&\text{isTop} := \varphi.\text{isTop} \ \vee \ \varphi.\text{cnVal} = 1 \wedge x.\text{isTop}\\
&\text{cnVal} := x.\text{cnVal}
\end{aligned}$$

This definition is sufficient to ensure that the Trigger-Operator is causal if neither `isBot` nor `isTop` evaluates to 1. However, if $\varphi.\text{isTop} = 1$, $\varphi.\text{cnVal} = 1$ and $x.\text{isBot} = 1$, then both `isBot` and `isTop` evaluate to 1, because technically $\varphi.\text{cnVal}$ can not be used as φ is $\top$ and thus not causal. To get a totally correct result, each access to a `cnVal` element must be guarded by an additional conjunction with $\varphi.\text{isBot} = 0 \wedge \varphi.\text{isTop} = 0$.

For most operations however (especially the numeric operations), this construction is trivially done with `isTop` simply being the disjunction of its operands `isTop` expressions and `isBot` being the disjunction of the `isBot` expressions. By definition $\top$ is stronger than $\bot$ in the sense that an operation with a $\bot$ operand and a $\top$ operand should evaluate to $\top$ (i.e. faulty values propagate through the whole program). Thus for all numeric operations and predications except if-then-else it holds that:

$$\text{Ecausal}(x \ \mathbf{op}\ y) = \left\{ \begin{array}{l} \text{isBot} := (x.\text{isBot} \ \vee \ y.\text{isBot}) \ \wedge \neg x.\text{isTop} \ \wedge \ \neg y.\text{isTop} \\ \text{isTop} := x.\text{isTop} \ \vee \ y.\text{isTop} \\ \text{cnVal} := x.\text{cnVal} \ \mathbf{op}\ y.\text{cnVal} \end{array} \right\}$$

On a final note, as the Sup-operator may contain multiple operands, it needs to be explained how this construction works for the Sup-operator. If the set inside the operator contains zero or only one element, this construction reduces to $\bot$ or the `Ecausal` of the single element respectively. The `Ecausal` for two elements however is defined as

$$\text{Ecausal}(\text{Sup}\,\{x, y\}) = \left\{ \begin{array}{l} \text{isBot} := x.\text{isBot} \wedge y.\text{isBot} \\ \text{isTop} := (x.\text{isTop} \vee y.\text{isTop}) \ \vee \\ \qquad\qquad (\neg x.\text{isBot} \wedge \neg y.\text{isBot} \ \wedge \ \neg(x = y)) \\ \text{cnVal} := (x.\text{isBot} \ ? \ y.\text{cnVal} : x.\text{cnVal}) \end{array} \right\}$$

thus, if Sup is neither $\bot$ nor $\top$, then both values are equal or exactly one is $\bot$. The if-then-else statement ensures that `cnVal` does in fact carry the value if one of the operands is $\bot$. Last but not least, the `Ecausal` for the Sup-operator with a bigger set can be recursively generated by using the associative rule

$$\text{Sup}\,\{x_1, x_2, x_3, \dots\} = \text{Sup}\,\{x_1, \text{Sup}\,\{x_2, x_3, \dots\}\}$$

together with the construction for two elements.

4.2. Extended Finite State Machine

Given a Quartz program, with help of the TeachingTools (which are also provided by Averest) this program can be parsed to a `QModule` which is a record-type. For symbolic analysis, the important part of this record is `declStmt` which contains multiple lists of all variable declarations used in the Quartz program (input, output, label, ...), as well as the whole statement `stmt` for the given program. By using the `ComputeStateTrans` function of the TeachingTools, the SOS-rules are then applied to this statement, finally returning an Extended Finite State Machine (EFSM), which serves as the basis for the symbolic analysis.

As stated in Section 2.4.2, the resulting EFSM consists of a list of its states called `NodeOfEFSM`. Each of those represent a timestep (macro-step), in which all actions up to the next `pause` statement are executed simultaneously, containing the following information:

sccIndex This is the index of the state, i.e. the identifier of the state in the EFSM.

labels A set of all the label variables that hold in this state (given by their `QName`).

thisStmt The current statement, which is the remaining program that is left from this point.

rsdStmts A set of all possible next states in the EFSM (given by their indices) together with the affiliated. transition condition. This can explicitly be a loop, i.e. the next state is the same state as before.

surfActs A set of guarded actions that are to be executed in this timestep or state. Guards are Boolean expressions over the readable variables whereas actions may be either an instantaneous assignment to a variable, a delayed assignment or an assertion which must be fulfilled.

transfer A map which maps all `QName`s of local variables to their transfer incarnations. These are important to deal with the schizophrenia problem (see 2.3.2), however as the guarded action implicitly already contain this information, this map is not used for the tool.

instantC This Boolean expression states if `thisStmt` terminates instantaneously within this state, depending on the readable variables. If there are no states in which this can become `true`, then the program will not terminate.

For the causality analysis the core information lies in `surfActs`, whereas the reachability analysis will make use of the transition conditions given in `rsdStmts`.

4.2.1. Labels and Current Statement

In each state of the EFSM except the initial state and potentially a sink state after termination, one label holds at any time, which is defined by the `pause` statement leading to this state. These labels can be explicitly given in the Quartz program by annotating the `pause` statements or implicitly by the compiler with a generic name. The statement given in `thisStmt` is the whole remaining statement which is to be executed from this point. If the `pause` statement leading to this state is part of a loop statement (for example `while`), the remaining statement up to the end of this loop is succeeded by the whole loop statement to form the complete remaining statement. This procedure effectively works like a loop unrolling of the current iteration, which is finally the reason why the EFSM can indeed contain cycles, because after completing one loop iteration the remaining statement is the same as before the loop.

4.2.2. Guarded Actions

All variable assignments, immediate as well as delayed, that can be encountered in the current statement before a `pause` statement, are transformed to guarded actions, where the guards are Boolean expressions that need to be true for the actions to take place. The sources of these guards are conditions from loops or other conditional statements (`if`, `abort`, `suspend`, ...). Further, also explicit assertions as well as implicit ones defined by the used operators (the modulo operation $a \% b$ for example defines the implicit assertion $b \neq 0$) are contained in the set `surfActs`. Although other actions beside assignments and assertions exist, they are not relevant for the implemented symbolic analysis. It has to be noted that the order of the different guarded actions does in fact not matter, as all actions in this state of the EFSM, by definition, are executed at the same time nonetheless (delayed assignments do not take effect until the next state/timestep though). For causality it is thus important to assure that no two actions contradict each other for any possible values of the used variables. This is one of the main goals of the symbolic analysis.

4.2.3. Transitions

Each transition between two states of the EFSM is guarded by a transition condition, which serve as the control flow between the states of the EFSM or the according macro-steps of the program respectively. With each `pause` statement ending the current macro-step, the transition condition leading to the state that is identified by the relevant `pause` is defined as the conjunction of all conditionals that lead to this `pause` statement. If there are paths that reach the end of the statement without a `pause` in between, the conditionals on these paths define the `instantC` expression of the state, which can also be interpreted as transition to an ending(/sink) state. As the transformation to the EFSM already incorporates the transition conditions as part of the guards for the actions in the same state, they do not need to be checked explicitly for causality. However, these conditions are still needed for a reachability analysis, as reachability does in fact influence causality indirectly. Assuming there is a state in the EFSM of a given program in which causality is not maintained, the program would still be causal if this state is not reachable by any means.

4.3. Generating State Sets

The following section concentrates on ordering and pre-processing the guarded actions of a single macro-step to make them more accessible for the symbolic analysis explained in Section 4.4. Further, partially complete expressions are generated for the used variables, which, together with the completing expressions will form the core elements of the causality analysis. All of the following procedures are done separately for each state of the EFSM.

4.3.1. Filtering Actions

With `surfActs` being totally unordered in regard to different actions and different variables in assignments, it makes sense to categorize the actions and filter the assignments according to the assigned variables. This is done by getting all relevant variables from the `declStmt` of the `QModule` and dividing them into input and writable variables based on their data flow. These two sets of variables are disjoint as input variables are fixed and can not be overwritten. A dictionary is then used to map all writable variables to the set of all immediate assignments of the respective variable:

```
let instDict = new Dictionary<(QName * Decl), Set<GrdAction>>()
```

In this definition, the original set of guarded actions from `surfActs` is filtered to only contain immediate assignments to the variable identified by the tuple of its name and declaration. This can be done analogously to receive a dictionary of all delayed assignments to writable variables (`nextDict`). Last but not least, a set of all assertions can be filtered from `surfActs`, which is thus named `assertions` in the following. As other actions are not needed for the symbolic analysis, they are not filtered to either of those three groups.

4.3.2. Variable Expressions

As for each variable the set of all (guarded) immediate assignments is known, each of those guarded assignments can be represented by the Trigger-operator, i.e. a guarded action with an immediate assignment to a variable x:

$$\langle \varphi \implies x = \tau \rangle$$

is represented by

$$x = (\varphi \Rightarrowtail \tau)$$

where φ is the Boolean guard and τ is an expression of type L^* which must be equivalent to the declared type of variable x. If there is more than one immediate assignment for a variable, all of those must be considered equally. For this, the Sup-operator is used by combining the multiple guarded actions $\langle \varphi_i \implies x = \tau_i \rangle$ as

$$x = \mathrm{Sup}\,\{(\varphi_1 \Rightarrowtail \tau_1), (\varphi_2 \Rightarrowtail \tau_2), (\varphi_3 \Rightarrowtail \tau_3), \dots\}$$

In this (extended) expression, x is assigned the value of all guarded assignments for which the guard is true, and in case there are multiple different values, x is assigned $\top$ due to the Sup-operator. This expression however is still not complete, as its missing the dependency to previous states given by delayed assignments and the reaction to absence, if neither an immediate assignment in this step nor a delayed assignment in the previous step is or has been executed.

Carrier Variables

This problem is solved by making use of helper variables δ_x and x' for each variable x which act as carrier between two macro-steps. With

$$\langle \varphi_i \implies x = \tau_i \rangle, \quad 1 \leq i \leq p$$

being the immediate assignments and

$$\langle \chi_i \implies \texttt{next}(x) = \pi_i \rangle, \quad 1 \leq i \leq q$$

being the delayed assignments, the following equation system can be defined ([Led19; Sch21]):

$$\texttt{next}(x') = \text{Sup}\left\{ (\chi_1 \Rightarrowtriangle \pi_1), \ldots, (\chi_q \Rightarrowtriangle \pi_q), \left(\neg \bigvee_{i=1}^{q} \chi_i \Rightarrowtriangle \text{Abs}(x) \right) \right\}$$

$$\texttt{next}(\delta_x) = \bigvee_{i=1}^{q} \chi_i$$

$$x = \text{Sup}\left\{ (\varphi_1 \Rightarrowtriangle \tau_1), \ldots, (\varphi_p \Rightarrowtriangle \tau_p), \left(\delta_x \vee \neg \bigvee_{i=1}^{p} \varphi_i \Rightarrowtriangle x' \right) \right\}$$

where $\text{Abs}(x)$ is defined as absence to reaction of variable x. Depending on the storage type of variable x, either $\text{Abs}(x) = x$ for `Memorized` variables or $\text{Abs}(x) = \text{Default}(x)$ where $\text{Default}(x)$ is the default value of the variable's data type (0 for Boolean and Numeric). Similar to the immediate case, x' combines all delayed assignments, together with an according absence to reaction value if neither of the delayed assignment's guards is fulfilled. Due to these assignments being delayed, x' as carrier variable is also delayed. The same holds for δ_x which simply acts as signal if any of the original delayed guarded assignments has taken place. The original assignment to x is now extended to also include any delayed assignment from the previous step (δ_x is true in this case) and also effectively including the absence to reaction value if neither δ_x nor any immediate assignment guard holds. To make this equation system complete, the initial values of x' and δ_x are:

$$\text{init}(x') = \text{Default}(x) \qquad \text{and} \qquad \text{init}(\delta_x) = 0$$

Generating Partial Expressions

In theory, this equation system incorporates everything that is needed to essentially calculate the symbolic expressions for each variable, however, during this pre-processing step, only the guarded actions of the same state are known and thus the value expression of x' and δ_x are not known and specifically depend on the former path through the EFSM. Thus they can not be precomputed in a static way. As F# supports partial function application, a new dictionary is created with:

```
let varOps = new Dictionary<(QName * Decl), stateVarOp>()
```

where `stateVarOp` is the record-type:

```
type stateVarOp = {
    nextOp: extendExpr -> Sup * BoolExpr;
    instOp: BoolExpr * extendExpr -> Sup
}
```

`nextOp` represents the first and second equation of the system, which, given the extended expression for Abs(x), returns the expressions for x' (which is of type Sup) and δ_x (which is Boolean expression). On the other side, `instOp` represents the third equation, which, given δ_x and x' (as extended expression) returns the Sup-expression for x. These partial expression or rather functions that generate the complete expression are linked to the affected variable in the dictionary and can thus be used during the symbolic analysis.

Adding the Symbolic Carrier Variables

As the carrier variables are not part of the original program but are still needed to distinguish their expressions from the expressions of the original variables, they are added to the namespace of the current `QModule` as derived `QName`s from the original variable's name. Further a last dictionary is created, which maps the original variables to their carrier variables and their `Ecausal` record with initial value expressions:

```
let vars = new Dictionary<varDecl, varSet>()

type varSet = {
    nextdvar: QName * Ecausal;
    nextVar': QName * Ecausal
}
```

4.4. Symbolic Execution

Together with all the information from the pre-processing steps, the symbolic execution can now be executed. This is done by means of a depth-first-search, starting with the initial state of the EFSM, and then alternatingly checking causality in this state and checking reachability for all possible following states, which if reachable continue the depth-first-search from these states up to a given maximum depth. After giving a short breakdown of all parameters needed for the `dfs` function in Section 4.4.1, each iteration of the depth-first-search can be divided into the following steps: Completing the partial expressions (Section 4.4.2), iterating variable substitutions to handle dependency cycles (Section 4.4.3), checking causality by using Z3 (Section 4.4.4), updating the carrier variables (Section 4.4.5) and finally checking the reachability of successor states in Section 4.4.6.

4.4.1. DFS Parameters

Each recursive call of the depth-first-search is initiated with these parameters:

stateIndex This is the index of the current state in the EFSM which is also used as identifier for the state.

vars The dictionary mapping writable variables to the respective carrier variables with their `Ecausal` record (`varSet`).

cond The complete path condition as `BoolExpr` up to this state, which is the conjunction of all transition conditions on this path. The condition is already fully substituted and thus only dependent on the input variables.

opList A list of dictionaries mapping writable variables to their respective partial expression functions (`stateVarOp`) for each state.

states This is essentially the whole EFSM given by a list of `NodeOfEFSM`. All information regarding the current state can be accessed together with `stateIndex`.

edictInit A dictionary with all initially known variables and their respective `Ecausal`. Known in this context means non-writable, i.e. only input variables which are assumed to be known and causal when executing the Quartz program.

inputVars The list of all input variables.

writeVars The list of all writable variables (which is disjoint to `inputVars`).

visitedStates This list is the trace of the current path of the depth-first-search in reversed order (the initial state is at the end of the list).

depth The current depth of the depth-first-search.

maxDepth The maximum depth of the depth-first-search to limit traversal.

Some of them contain only static or pre-processed information and thus do not change between different iterations (`opList`, `states`, `edictInit`, `inputVars`, `writeVars` and `maxDepth`). For the symbolic analysis, the most import dynamic parameter is `vars`, as it contains the current expression values of the carrier variables from the previous step, together with `stateIndex` as it indirectly changes the lookup in `opList` (which itself remains static). Beside the static components which are pre-computed, the depth-first-search is started with the following parameters:

$$\begin{aligned}\texttt{stateIndex} = \texttt{depth} &= 0\\ \texttt{visitedStates} &= []\\ \texttt{cond} &= \texttt{BoolConst(true)}\\ \texttt{vars} &= \texttt{vars}\ \text{(with the precomputed initial carrier variables)}\end{aligned}$$

The parameter `maxDepth` can be chosen arbitrarily as tuning parameter.

4.4.2. Expression Completion

In comparison to the pre-processing steps, the expression values (given as `Ecausal`) of all carrier variables are known in each iteration of the depth-first-search, as they were either computed in the previous iteration or are set to their initial values in the first state. Thus the expression for the immediate assignment can be completed for each writable variable. This is done by accessing the `stateVarOp` record for each writable variable x in `opList.[stateIndex]` and calling the `instOp` function on newly created variable expressions (`BoolVar`, `NatVar`, ...) for the carrier variables δ_x and x'. The result of this function call will be the complete Sup-expression including the symbolic carrier variables. All of these completed expressions are collected in a new dictionary mapping the variables to these expressions (`fullVarOps`). To make the `Ecausal`s of the carrier variables accessible, they need to be added to the dictionary of known variables, which is given by `edict` and initially contains only the input variables. As the input variables stay the same in each iteration of the DFS while the writable and carrier variables change, `edict` is created from `edictInit` directly to maintain an unchanged copy.

```
let mutable edict = new Dictionary<QName, Ecausal>(edictInit)
```

After all carrier variables and their expression values are added to `edict`, the symbolic variable substitution can commence.

4.4.3. Iterated Variable Substitution

The idea of this part is to effectively perform a fixpoint computation (see Section 3.4) by repeatedly replacing the occurrences of writable variables with their expression value from the previous (fixpoint-)iteration to finally determine the current symbolic values of these variables if possible (i.e. if these values are causal). For this, all writable variables are assumed to be unknown at first and thus have the value $\bot$. In each iteration, for each writable variable the completed expression is recursively transformed to an `Ecausal` record by joining the operands' `Ecausal`s according to the operator definition (see Section 4.1.2). When this recursion encounters a variable definition for any type, two cases can happen: Either the variable is known and thus already in the dictionary `edict` – in this case the `Ecausal` given by the dictionary is applied – or the variable is unknown and thus an `Ecausal` with `isBot` $= 1$ is applied. In either case, after the recursion terminates, the top-level `Ecausal` is added to `edict` for the according variable or replaces the entry if it already exists. To maintain the same knowledge base inside each iteration for all variables, these additions or updates are in fact done to a local copy `edict'`, which merges into `edict` only after the whole iteration concludes. The next iteration then starts with the updated expression values in `edict`.

As formulas are not necessarily simplified, even if a fixpoint is reached semantically, the formulas may not be equal to the previous iteration syntactically. Further, as dependency cycles may exists between numeric variables, a

fixpoint may not be reached at all (see Section 3.4.2). For these reasons, instead of checking that a fixpoint is reached after each iteration, the number of iterations is fixed to be equal to the number of writable variables. The reason for this is that no dependency cycle is longer than the number of variables, as this already implies that the cycle contains every variable. If all dependencies are represented in a dependency graph, this would analogously mean that each dependency(-edge) is taken into account, thus capturing the full information. After this number of iterations, either all cyclic dependencies are resolved or it is impossible to resolve them (for example if a numeric dependency cycle exists). If all dependencies are acyclic, this problem does not occur and a fixpoint is reached in any case (at least semantically). With each variable having a final record of type `Ecausal`, which implicitly includes all cyclic dependencies (resolvable or not), the variables can be checked for causality in the next step.

4.4.4. Causality Check

As each variable is associated to an `Ecausal` record, causality can verified by checking if there is any variable assignment of the input variables that makes any of the according Boolean expressions for `isBot` or `isTop` true. This is the step that is done by the SMT-solver, which, given a Boolean expression and the types for the variables, returns one of three results:

SATISFIABLE The Boolean expression is satisfiable, which means there is a variable assignment making the verified variable $\bot$ or $\top$ respectively. In this case the variable and thus the program is not causal due to the found issue.

UNSATISFIABLE The Boolean expression is certified unsatisfiable, rendering the verified variable as causal in the current macro-step, if this result holds for both expressions `isBot` and `isTop`.

UNKNOWN In this case, the SMT-Solver did neither find a satisfiable variable assignment nor could it certify that there is no such assignment. For the symbolic analysis this result will be treated the same as `UNSATISFIABLE` to continue the analysis, although all following results should be taken with care due to this uncertainty.

If all variables are causal (or assumed to be causal due to an `UNKNOWN` result) the symbolic analysis can continue with the next step. Otherwise the current function call of `dfs` returns due to the found problem. The detailed transition of the relevant Boolean expressions to feed the SMT-solver is explained in Chapter 5.1.

4.4.5. Update Carrier Variables

This part is similar to Section 4.4.2, because now the expressions for the carrier variables are completed by using the reaction-to-absence values of the original

writable variables, which are either the default values for `Event` storage types or the current values for `Memorized` storage types, which were just computed in the previous steps. Instead of collecting these expressions in a dictionary as in Section 4.4.2, they can directly be transformed to `Ecausal`s by substituting the variables. As the carrier variables are updated in a delayed manner, i.e. for the next macro-step, there can be no cyclic dependencies or any dependencies between different carrier variables at all. Thus only one iteration is needed for all substitutions, which makes intermediate uses of the expressions redundant. The generated `Ecausal`s however are collected to form the new `vars` parameter for the future recursive calls to `dfs`.

4.4.6. Reachability Check

At first it is checked whether `maxdepth` is already reached, if yes the current function returns, otherwise each transition in `state.rsdStmts` is checked for reachability. For this the transition condition is added to the current path condition by conjunction. This new path condition, which only depends on already known variables, is transformed to `Ecausal` and then its `cnVal` can be checked for satisfiability. It has to be noted, that during the transformation, effectively only the variables are substituted which were previously shown to be causal, thus the causality of the path condition does not need to be checked.

The reachability check works essentially the same as the causality check by putting the path condition together with the variable types into the SMT-solver, with the sole difference that `SATISFIABLE` and `UNKNOWN` render the successor state reachable (presumably in the second case) while `UNSATISFIABLE` certifies that the path condition can in no way be fulfilled and thus the relevant successor state is not reachable. Finally, on all states that are assumed to be reachable, `dfs` is called to continue the traversal of the depth-first-search. If at some point a state is reached in which no successor states exists, the current function call also returns, due to having reached the end of the program.

4.5. Example

An extensive example is used to demonstrate the different steps during the execution of the tool, however, as the expressions can get very large, they are simplified such that they are better readable. Especially $\neg(x = y)$ is written as $x \neq y$ and $\neg(x \leq y)$ is written as $x > y$ For this, the program `EuclidMod.qrz` is used which calculates the greatest common divisor of two numbers by using the Euclidian algorithm. The Quartz program and the resulting EFSM look like this:

```
macro N = 1000;
module EuclidMod(nat{N} ?a,?b,x,event !rdy) {
    nat{N} y;
    x = a;
    y = b;
    do {
        if(x>=y)
            next(x) = x % y;
        else
            next(y) = y % x;
        lop:pause;
    } while(x!=0 & y!=0);
    if(x==0) next(x) = y;
    end:pause;
    emit(rdy);
}
```

Code 10: *EuclidMod.qrz*

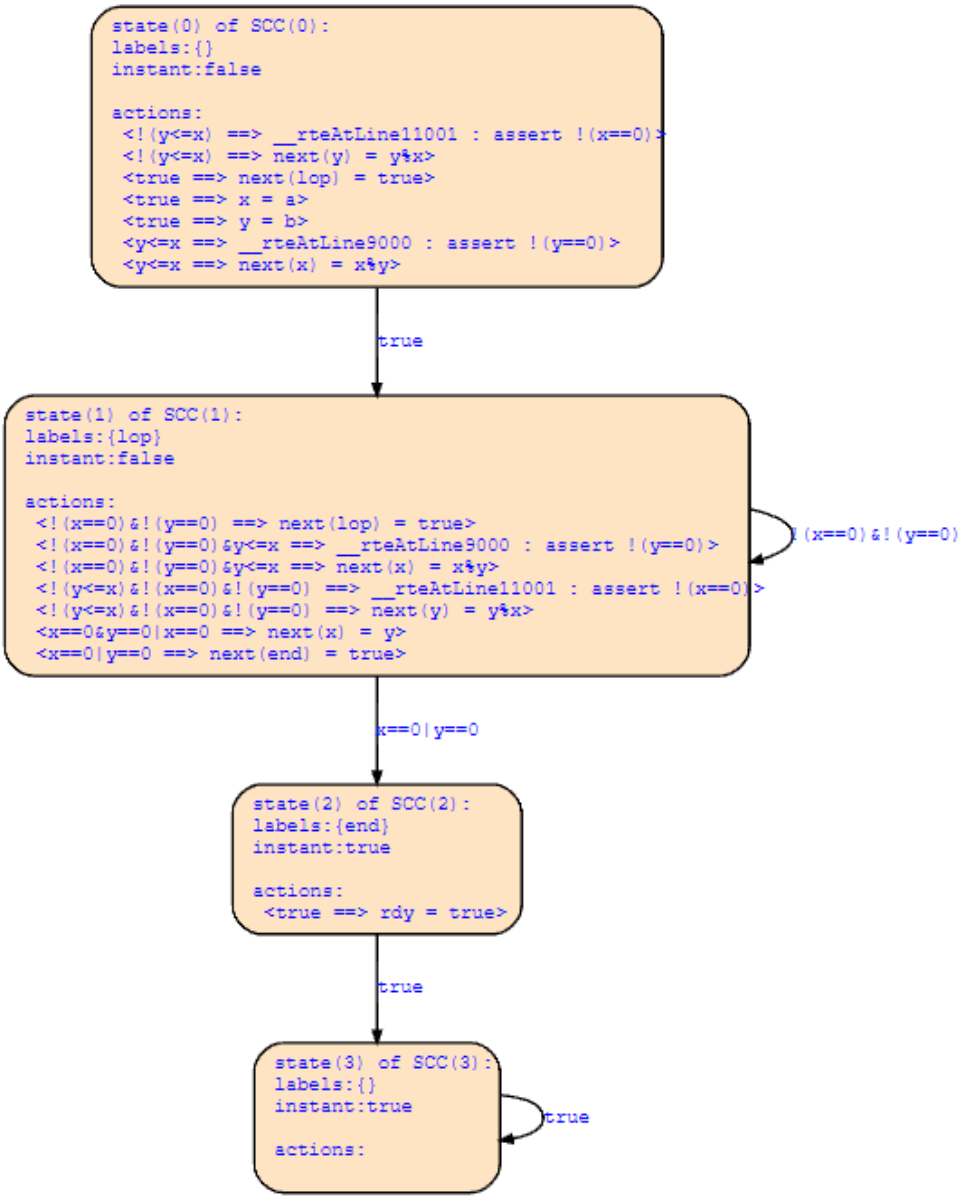

Figure 4.1.: *EFSM of* `EuclidMod.qrz` *with sink state*

For state 0 the equation system looks like this for all non-label variables:

$$\begin{aligned}
\texttt{next}(x') &= \mathrm{Sup}\,\{(y \le x \Rightarrow x\%y), (\neg(y \le x) \Rightarrow \mathrm{Abs}(x))\} \\
\texttt{next}(\delta_x) &= y \le x \\
x &= \mathrm{Sup}\,\big\{(\texttt{true} \Rightarrow a), \big(\delta_x \vee \neg\texttt{true} \Rightarrow x'\big)\big\} \\
\texttt{next}(y') &= \mathrm{Sup}\,\{(y > x \Rightarrow y\%x), (\neg(y > x) \Rightarrow \mathrm{Abs}(y))\} \\
\texttt{next}(\delta_y) &= y > x \\
y &= \mathrm{Sup}\,\big\{(\texttt{true} \Rightarrow b), \big(\delta_y \vee \neg\texttt{true} \Rightarrow y'\big)\big\} \\
\texttt{next}(\mathrm{rdy}') &= \mathrm{Sup}\,\{(\neg\texttt{false} \Rightarrow \mathrm{Abs}(\mathrm{rdy}))\} \\
\texttt{next}(\delta_{\mathrm{rdy}}) &= \texttt{false} \\
\mathrm{rdy} &= \mathrm{Sup}\,\big\{\big(\delta_{\mathrm{rdy}} \vee \neg\texttt{false} \Rightarrow \mathrm{rdy}'\big)\big\}
\end{aligned}$$

For state 1 however, the equation system is as follows:

$$\begin{aligned}
\texttt{next}(x') &= \mathrm{Sup}\big\{(x \neq 0 \wedge y \neq 0 \wedge y \le x \Rightarrow x\%y), \\
&\qquad (x = 0 \wedge (y = 0 \vee x = 0) \Rightarrow y), \\
&\qquad (\neg(x \neq 0 \wedge y \neq 0 \wedge y \le x \ \vee \ x = 0 \wedge (y = 0 \vee x = 0)) \Rightarrow \mathrm{Abs}(x))\big\} \\
\texttt{next}(\delta_x) &= x \neq 0 \wedge y \neq 0 \wedge y \le x \ \vee \ x = 0 \wedge (y = 0 \vee x = 0) \\
x &= \mathrm{Sup}\,\big\{\big(\delta_x \vee \neg\texttt{false} \Rightarrow x'\big)\big\} \\
\texttt{next}(y') &= \mathrm{Sup}\big\{(x \neq 0 \wedge y \neq 0 \wedge y > x \Rightarrow y\%x), \\
&\qquad (\neg(x \neq 0 \wedge y \neq 0 \wedge y > x) \Rightarrow \mathrm{Abs}(y))\big\} \\
\texttt{next}(\delta_y) &= x \neq 0 \wedge y \neq 0 \wedge y > x \\
y &= \mathrm{Sup}\,\big\{\big(\delta_y \vee \neg\texttt{false} \Rightarrow y'\big)\big\} \\
\texttt{next}(\mathrm{rdy}') &= \mathrm{Sup}\,\{(\neg\texttt{false} \Rightarrow \mathrm{Abs}(\mathrm{rdy}))\} \\
\texttt{next}(\delta_{\mathrm{rdy}}) &= \texttt{false} \\
\mathrm{rdy} &= \mathrm{Sup}\,\big\{\big(\delta_{\mathrm{rdy}} \vee \neg\texttt{false} \Rightarrow \mathrm{rdy}'\big)\big\}
\end{aligned}$$

For the final state 2 (which is not the sink state), the following holds:

$$\begin{aligned}
\texttt{next}(x') &= \mathrm{Sup}\,\{(\neg\texttt{false} \Rightarrow \mathrm{Abs}(x))\} \\
\texttt{next}(\delta_x) &= \texttt{false} \\
x &= \mathrm{Sup}\,\big\{\big(\delta_x \vee \neg\texttt{false} \Rightarrow x'\big)\big\} \\
\texttt{next}(y') &= \mathrm{Sup}\,\{(\neg\texttt{false} \Rightarrow \mathrm{Abs}(y))\} \\
\texttt{next}(\delta_y) &= \texttt{false} \\
y &= \mathrm{Sup}\,\big\{\big(\delta_y \vee \neg\texttt{false} \Rightarrow y'\big)\big\} \\
\texttt{next}(\mathrm{rdy}') &= \mathrm{Sup}\,\{(\neg\texttt{false} \Rightarrow \mathrm{Abs}(\mathrm{rdy}))\} \\
\texttt{next}(\delta_{\mathrm{rdy}}) &= \texttt{false} \\
\mathrm{rdy} &= \mathrm{Sup}\,\big\{(\texttt{true} \Rightarrow \texttt{true}), \big(\delta_{\mathrm{rdy}} \vee \neg\texttt{true} \Rightarrow \mathrm{rdy}'\big)\big\}
\end{aligned}$$

Starting with these expressions (which are already completed symbolically) and the following initial `Ecausal` records, (written as $E(x) = [\langle\texttt{isBot}\rangle, \langle\texttt{isTop}\rangle, \langle\texttt{cnVal}\rangle]$):

$$E(x') = E(y') = [\texttt{false}, \texttt{false}, 0]$$

$$E(\delta_x) = E(\delta_y) = E(\delta_{\text{rdy}}) = E(\text{rdy}') = [\texttt{false}, \texttt{false}, \texttt{false}]$$

$$E(a) = [\texttt{false}, \texttt{false}, a] \qquad E(b) = [\texttt{false}, \texttt{false}, b]$$

the depth-first-search commences with the initial state. After the first iteration of the fixpoint analysis by substitution, the variables have the following `Ecausals`:

$$\begin{aligned} E(x) &= [\texttt{false}, \texttt{false}, a] \\ E(y) &= [\texttt{false}, \texttt{false}, b] \\ E(\text{rdy}) &= [\texttt{false}, \texttt{false}, \texttt{false}] \end{aligned}$$

To concretize this example, for variable x the following intermediate computations are made:

$$\begin{aligned} E(\delta_x \vee \neg\texttt{true}) &= [E(\delta_x).\texttt{isBot} \wedge E(\neg\texttt{true}).\texttt{isBot} \vee \\ &\qquad E(\delta_x).\texttt{isBot} \wedge \neg E(\neg\texttt{true}).\texttt{cnVal} \vee \\ &\qquad \neg E(\delta_x).\texttt{cnVal} \wedge E(\neg\texttt{true}).\texttt{isBot}, \\ &\quad E(\delta_x).\texttt{isTop} \vee E(\neg\texttt{true}).\texttt{isTop}, \\ &\quad E(\delta_x).\texttt{cnVal} \vee E(\neg\texttt{true}).\texttt{cnVal}] \\ &= [\texttt{false} \wedge \texttt{false} \ \vee \ \texttt{false} \wedge \texttt{true} \ \vee \ \texttt{true} \wedge \texttt{false}, \\ &\quad \texttt{false} \vee \texttt{false}, \\ &\quad \texttt{false} \vee \texttt{false}] \\ &= [\texttt{false}, \texttt{false}, \texttt{false}] \\ E(\delta_x \vee \neg\texttt{true} \Rightarrow x') &= (E(\delta_x \vee \neg\texttt{true}) \Rightarrow E(x')) \\ &= [\texttt{true}, \texttt{false}, 0] \\ E(\texttt{true} \Rightarrow a) &= (E(\texttt{true}) \Rightarrow E(a)) \\ &= [\texttt{false}, \texttt{false}, a] \end{aligned}$$

and finally with the definition for the Sup-operator given in Section 4.1.2 this results in:

$$\begin{aligned} E\big(\text{Sup}\big\{(\texttt{true} \Rightarrow a), \big(\delta_x \vee \neg\texttt{true} \Rightarrow x'\big)\big\}\big) \\ &= [\texttt{false} \wedge \texttt{false}, \\ &\quad (\texttt{false} \vee \texttt{false}) \ \vee \ (\neg\texttt{false} \wedge \neg\texttt{true} \wedge \neg(a = 0)), \\ &\quad (\texttt{false} \ ?\ 0 : a)] \\ &= [\texttt{false}, \texttt{false}, a] \end{aligned}$$

As there are no dependencies between the writable variables, the fixpoint is directly reached, although the tool iterates two more times through the same

substitution with the same final result. Without further checking it is clear that all three variables are causal, due to all `isBot` and `isTop` entries having the constant value `false`. In the next step, the carrier variables are updated, which has the following result:

$$\begin{aligned}
E(\texttt{next}(x')) &= E\left(\mathrm{Sup}\left\{(b \le a \Rightarrow a\%b), (\neg(b \le a) \Rightarrow a)\right\}\right) \\
&= [\texttt{false}, \texttt{false}, (\neg(b \le a)\ ?\ a : a\%b)] \\
\texttt{next}(\delta_x) &= [\texttt{false}, \texttt{false}, b \le a] \\
\texttt{next}(y') &= [\texttt{false}, \texttt{false}, (b \le a\ ?\ b : b\%a)] \\
\texttt{next}(\delta_y) &= [\texttt{false}, \texttt{false}, \neg(b \le a)] \\
\texttt{next}(\mathrm{rdy}') &= [\texttt{false}, \texttt{false}, \texttt{false}] \\
\texttt{next}(\delta_{\mathrm{rdy}}) &= [\texttt{false}, \texttt{false}, \texttt{false}]
\end{aligned}$$

with the intermediate steps for x':

$$\begin{aligned}
E((b \le a \Rightarrow a\%b)) &= [\neg(b \le a), \texttt{false}, a\%b] \\
E((\neg(b \le a) \Rightarrow a\%b)) &= [b \le a, \texttt{false}, a]
\end{aligned}$$

and

$$\begin{aligned}
&E\big(\mathrm{Sup}\big\{(b \le a \Rightarrow a\%b), (\neg(b \le a) \Rightarrow a)\big\}\big) \\
&\qquad = [\neg(b \le a) \wedge b \le a, \\
&\qquad\quad (\texttt{false} \vee \texttt{false}) \ \vee\ (b \le a \wedge \neg(b \le a) \wedge \neg(a\%b = a)), \\
&\qquad\quad (\neg(b \le a)\ ?\ a : a\%b)] \\
&\qquad = [\texttt{false}, \texttt{false}, (\neg(b \le a)\ ?\ a : a\%b)]
\end{aligned}$$

This simple example already shows that the values of the carrier variables are now dependent on the input variables due to the if-then-else statements in x' and y'. The next step to close one iteration of the depth-first-search is to check reachability of all the successor states of the EFSM. However, as there is only one successor state which can always be reached (due to the transition condition being `true`), this computation is trivial.

5. Condition Verification

5.1. Translating Verification Conditions

With the iterated variable substitution in Section 4.4.3 having concluded, all variables are related to an `Ecausal` record which can be checked for causality by verifying that there is no (input) variable assignment fulfilling either `isBot` or `isTop`. As soon as causality of all variables is ensured, the transition conditions leading to the next state are checked for reachability, because no further analysis must be performed on unreachable states. The following subsections explain the translation of the Boolean expressions to work in the framework provided by Z3.

5.1.1. Causality Check

Causality is checked by the `checkBot` function, which needs the following parameters:

isBot The Boolean expression of the `isBot` component of an `Ecausal` which is to be checked for satisfiability.

cond The complete path condition up to this state (which is equal to `cond` in the current iteration of `bfs`).

inputVars The list of all input variables (which means their name and declaration).

writeVars The list of all writbale variables.

debug A Boolean flag indicating if debug messages (i.e. the results of the verification) should be printed.

debugString A debug string that is prepended to the results to give more information about the verified expression (for example which variable in which state is checked at the moment).

mode A mode string which is only used in the debug message to denote the mode (should be either “bot” or “top”).

With this information, a new `Context` containing a solver is created in Z3s framework, which can then be enriched with variable definitions and expressions. First, for all variables in `inputVars` and `writeVars` a related variable with the same type is created in the framework (which is called a constant). For this declaration it has to be noted that variables of the `Real` type in Quartz is translated to floating point variables in the Z3 framework, while

`Nat` variables in Quartz are translated to integers in the framework as no natural number type exists there. To compensate this, for each variable x of type `Nat` an expression of the form $x \geq 0$ is created which ensures that only non-negative values are valid.

In the next step, the Boolean expression for `cond` is translated into the framework. This is done similarly to the variable substitutions which created the `Ecausals`, by recursively going through the expression. For each operation in Quartz, the recursively calculated Z3 expressions of the operands are connected by the according Z3 operation. Constants in the expression are created on the go, while variables are replaced by their Z3 variant (which can be accessed through a dictionary that was created during the first step). The resulting Z3 expression together with all the expressions ensuring that `Nat` variables are non-negative form a precondition for the following causality check and thus need to be asserted. Any variable assignment that does not fulfill this precondition will not have lead the execution to the currently examined state, thus they can (and must) be ignored.

Last but not least the actual verification condition `isBot` is translated to the Z3 framework the same way as `cond` and added to the assertions. Calling the `Check` function on the solver starts the verification process, resulting in either of the three statuses that are explained in Section 4.4.4. If the whole asserted expression is satisfiable, the satisfying values can be extracted from the model to receive a valid variable assignment which leads to the currently checked causality problem.

It has to be noted that this whole function can be used exactly the same way for checking the Boolean expressions for `isBot` and `isTop`, the only difference being the interpretation of the result as either $\bot$ or $\top$. Furthermore it needs to be explained why `writeVars` are included during this computation, as values may directly only depend on the input variables. The reason for this is simply that statements like `x = x + 1` which are directly cyclic dependent do not get rid of the `cnVal` expression containing this variable during the fixpoint iteration (as they can not be rewritten to be only dependent on input variables). However this value is surely not valid because `isBot` is 1 in this case and thus any symbolic expression using the `cnVal` prospectively will implicitly also check the `isBot` value which directly reasons the non-causality. Thus the inclusion of writable variables is only important such that during the translation process no error is encountered due to an undefined variable. As a final note, this inclusion logically also introduces more tweakable parameters for Z3 to satisfy the given Boolean expression, thus giving even more reasoning to find a way that makes the expression `true` which implies that the verified variable is not causal. If the expression however is still not satisfiable even with those theoretical, extra parameters, then there is in fact no reason to assume that the variable is not causal.

5.1.2. State Reachability

State reachability can be done nearly exactly the same way as the causality check, the only difference is that there is no additional verification condition

beside the path condition `cond`. However this path condition is already one of the new path conditions for the possible successor states for which reachability should be checked. As this step is only done after causality of all variables has been verified, each variable of the path condition has a uniquely determined value which only depends on the input variables. Therefore any satisfying assignment does in fact show reachability and also gives a concrete value example for all input variables, which can be used for further verification purposes.

5.2. Results

The tool has been tested on various Quartz programs, which did in fact produce nearly all of the expected results for different types of causality issues. A list of many examples including the expected and produced result can be found in the Appendix A. Program `P16` however led to a different result than expected which thus needs to be analyzed further.

In the given program, the only emission of o is behind two conditionals which need the value of o to be known to compute the result. This is a direct causality cycle which can logically not be resolved by using a fixpoint-analysis. However, the guard that must hold for this emission would be $o \wedge \neg o$ which can never be fulfilled. When checking the generated EFSM, it can be seen that the function generating the EFSM catches this condition and reduces it to `false` , thus making the resulting EFSM trivial. As the implemented tool uses the output of this function (i.e. the trivial EFSM), the problematic condition is not present anymore and thus the tool gives the correct answer for the trivial EFSM, which is constructive causality.

Other than this edge case, the tool also works correctly on the `EuclidMod` example (Code 10), however as soon the maximum depth extends a certain threshold, the running time of Z3 quickly gets very large for each single verification procedure. Looking at the checked formulas, the reason for this is directly visible, as they grew exponentially larger in each step. This is due to multiple if-then-else statements for the Sup-operator acting on another, effectively doubles the number of different cases (which however is also expected for loops). To make the computation more feasible, the approach could be enhanced by replacing certain expressions (like if-then-else) which appear multiple times in another expression with a symbolical variable which is connected to the replaced expression. This way, each of those expressions must be only computed once, while their values can be broadcast to all occurrences of the symbolical variables.

6. Conclusion

In synchronous languages there are many different ways to define causality cycles, which can be seen by the various examples given in the Appendix A. However the results show that almost all non-causal cycles can be detected by the presented tool, with some of those depending on the concrete definition of constructiveness or the actual used semantics. While the formally defined ways of using the Sup-operator and the Trigger-Operator to express the final value of a variable (including the two symbolical values $\bot$ and $\top$ of the extended lattice) make the symbolic execution manageable, the iterated application of those expression in later macro-steps still leads to an exponential blowup which can not be prevented.

To mitigate this problem, an improved version of the tool could use additional symbolic variables for expression which have already been calculated to reduce the computational load on Z3, as proposed at the end of Section 5.2. As a further extension, composite types like arrays or tuples could be added, making the Quartz type set complete. Finally, by symbolically connecting the two opposite conditions of any conditional or loop, the computational load could be reduced further, as those contrary conditions (which are uncoupled in guarded actions) logically can never be true at the same time.

Bibliography

[Ber00] Gérard Berry. *The Esterel v5 language primer: version v5_91*. Centre de mathématiques appliquées, Ecole des mines and INRIA, 2000.

[Ber89] Gérard Berry. "Real time programming: Special purpose or general purpose languages". PhD thesis. INRIA, 1989.

[BFT17] Clark Barrett, Pascal Fontaine, and Cesare Tinelli. *The SMT-LIB Standard: Version 2.6*. Tech. rep. Available at `www.SMT-LIB.org`. Department of Computer Science, The University of Iowa, 2017.

[BG92] Gérard Berry and Georges Gonthier. "The Esterel synchronous programming language: design, semantics, implementation". In: *Science of Computer Programming* 19.2 (1992), pp. 87–152. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/0167-6423(92)90005-V`. URL: `https://www.sciencedirect.com/science/article/pii/016764239290005V`.

[BS91] F. Boussinot and R. de Simone. "The ESTEREL language". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1293–1304. DOI: `10.1109/5.97299`.

[Hal+91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. "The synchronous data flow programming language LUSTRE". In: *Proceedings of the IEEE* 79.9 (1991), pp. 1305–1320. DOI: `10.1109/5.97300`.

[Hal98] Nicolas Halbwachs. "Synchronous programming of reactive systems". In: *Computer Aided Verification*. Ed. by Alan J. Hu and Moshe Y. Vardi. Berlin, Heidelberg: Springer Berlin Heidelberg, 1998, pp. 1–16. ISBN: 978-3-540-69339-0.

[Har87] David Harel. "Statecharts: a visual formalism for complex systems". In: *Science of Computer Programming* 8.3 (1987), pp. 231–274. ISSN: 0167-6423. DOI: `https://doi.org/10.1016/0167-6423(87)90035-9`. URL: `https://www.sciencedirect.com/science/article/pii/0167642387900359`.

[HM95] Nicolas Halbwachs and Florence Maraninchi. "On the symbolic analysis of combinational loops in circuits and synchronous programs". In: Citeseer. 1995.

[Le +86] P. Le Guernic, A. Benveniste, P. Bournai, and T. Gautier. "Signal–A data flow-oriented language for signal processing". In: *IEEE Transactions on Acoustics, Speech, and Signal Processing* 34.2 (1986), pp. 362–374. DOI: `10.1109/TASSP.1986.1164809`.

[Led19] Matthias Lederer. "Causal Correctness as a Safety Property". Bachelor Thesis. TU Kaiserslautern, 2019.

[PHP87] Daniel Pilaud, N Halbwachs, and JA Plaice. "LUSTRE: A declarative language for programming synchronous systems". In: *Proceedings of the 14th Annual ACM Symposium on Principles of Programming Languages (14th POPL 1987). ACM, New York, NY.* Vol. 178. 1987, p. 188.

[SB17] Klaus Schneider and Jens Brandt. "Quartz: A Synchronous Language for Model-Based Design of Reactive Embedded Systems". In: *Handbook of Hardware/Software Codesign.* Ed. by Soonhoi Ha and Jürgen Teich. Dordrecht: Springer Netherlands, 2017, pp. 1–30. ISBN: 978-94-017-7358-4. DOI: 10.1007/978-94-017-7358-4_3-1. URL: https://doi.org/10.1007/978-94-017-7358-4_3-1.

[SBS04] Klaus Schneider, Jens Brandt, and Tobias Schuele. "Causality analysis of synchronous programs with delayed actions". In: *Proceedings of the 2004 international conference on Compilers, architecture, and synthesis for embedded systems.* 2004, pp. 179–189.

[SBT96] T.R. Shiple, G. Berry, and H. Touati. "Constructive analysis of cyclic circuits". In: *Proceedings ED TC European Design and Test Conference.* 1996, pp. 328–333. DOI: 10.1109/EDTC.1996.494321.

[Sch09] Klaus Schneider. *The Synchronous Programming Language Quartz.* Language Specification. TU Kaiserslautern, 2009.

[Sch20] Klaus Schneider. *Symbolic Causality Analysis.* Chapter of the lecture Model-based Design of Embedded Systems. 2020.

[Sch21] Klaus Schneider. *Compiling Quartz to Synchronous Guarded Actions and Equation Systems.* Chapter of the lecture Model-based Design of Embedded Systems. 2021.

[TS04] O. Tardieu and R. de Simone. "Curing schizophrenia by program rewriting in Esterel". In: *Proceedings. Second ACM and IEEE International Conference on Formal Methods and Models for Co-Design, 2004. MEMOCODE '04.* 2004, pp. 39–48. DOI: 10.1109/MEMCOD.2004.1459813.

A. Examples

In this Appendix, multiple examples are given which are taken from the original Quartz paper [Sch09], showing diverse causality problems. For each program, the expected result as well as the result from the tool are given. The cases in which the tool's result differs from the expected result, the reason for this is explained in 5.2.

```
module P01(event ?i, o1, o2, o3) {
  if(i) emit(o1);
  if(!o1) emit(o2);
  if(o2) emit(o3);
}
```

- Expected result: `P01` is constructive
- Actual result: o_1, o_2 and o_3 are causal $\Rightarrow$ constructive

```
module P02(event o1, o2) {
  emit(o2);
  if(!o1) {
    if(o2) w: pause;
    emit(o1);
  }
}
```

- Expected result: `P02` is constructive
- Actual result: o_1 and o_2 are causal $\Rightarrow$ constructive

```
module P03(event o) {
  if(!o) emit(o);
}
```

- Expected result: `P03` is **not** constructive
- Actual result: o is $\bot$ $\Rightarrow$ **not** constructive

```
module P04(event o) {
  if(o) emit(o);
}
```

- Expected result: `P04` is **not** constructive
- Actual result: o is $\bot$ $\Rightarrow$ **not** constructive

```
module P05(event o1, o2) {
  if(o1) emit(o1);
  if(!o2) emit(o2);
}
```

- Expected result: `P05` is **not** constructive
- Actual result: o_1 and o_2 are $\bot$ $\Rightarrow$ **not** constructive

```
module P06(event o1, o2) {
  if(o1) emit(o2);
  if(o2) emit(o1);
}
```

- Expected result: `P06` is **not** constructive
- Actual result: o_1 and o_2 are $\perp \Rightarrow$ **not** constructive

```
module P07(event o) {
  if(o) w: pause;
  emit(o);
}
```

- Expected result: `P07` is **not** constructive
- Actual result: o is $\perp \Rightarrow$ **not** constructive

```
module P08(event ?i, o1, o2) {
  weak immediate abort {
      {
      if(!i) w: pause;
      emit(o1);
      }
    ||
      if(o1) emit(o2);
  } when(o2);
  emit(o1);
}
```

- Expected result: `P08` is **not** constructive
- Actual result: o_1 and o_2 are $\perp$ when $i = 0 \Rightarrow$ **not** constructive

```
module P09(event o1,o2) {
  if(o1) emit(o1);
||
  if(o1)
    if(o2) nothing;
    else emit(o2);
}
```

- Expected result: `P09` is **not** constructive
- Actual result: o_1 and o_2 are $\perp \Rightarrow$ **not** constructive

```
module P10(event o) {
  if(o) nothing;
  emit(o);
}
```

- Expected result: `P10` is constructive[1]
- Actual result: o_1 and o_2 are causal $\Rightarrow$ constructive

[1] This example is dependent on the way constructivity is approached, as the equation system approach will lead to `P10` being not constructive. The emission however is instantaneous in any case and not dependent on the conditional, which makes it reasonable to assume constructivity.

```
module P11(event o1, o2) {
  if(o1) {
    emit(o2);
    if(o2) w: pause;
    emit(o1);
  }
}
```

- Expected result: `P11` is **not** constructive
- Actual result: o_1 and o_2 are $\perp$ $\Rightarrow$ **not** constructive

```
module P12(event o) {
  if(o) emit(o);
  else emit(o);
}
```

- Expected result: `P12` is **not** constructive
- Actual result: o is $\perp$ $\Rightarrow$ **not** constructive

```
module P13(event ?i, o1, o2) {
  if(i) {
    if(o1) emit(o2);
  } else {
    if(o2) emit(o1);
  }
}
```

- Expected result: `P13` is constructive
- Actual result: o_1 and o_2 are causal $\Rightarrow$ constructive

```
module P14(event o1, o2) {
  if(o1) emit(o2);
  w: pause;
  if(!o2) emit(o1);
}
```

- Expected result: `P14` is constructive
- Actual result: o_1 and o_2 are causal in both macro-steps $\Rightarrow$ constructive

```
module P15(event o1, o2) {
  emit(o2);
  if(o1)
    if(!o2) emit(o1);
}
```

- Expected result: `P15` is constructive
- Actual result: o_1 and o_2 are causal $\Rightarrow$ constructive

```
module P16(event o) {
  if(o)
    if(!o) emit(o);
}
```

- Expected result: `P16` is **not** constructive
- Actual result: o is causal $\Rightarrow$ constructive $\lightning$

```
module P17(event o1, o2) {
  if(o1) {
    emit(o2);
    if(!o2) emit(o1);
  }
}
```

- Expected result: `P17` is **not** constructive
- Actual result: o_1 and o_2 are $\perp \quad \Rightarrow \quad$ **not** constructive