Analysis of Concurrency in Synchronous Systems

Masterarbeit von

Daniel Thielsch

1. Juni 2014

Referent: Prof. Dr. Klaus Schneider

Betreuer: Yu Bai

Hiermit versichere ich, dass ich die von mir vorgelegte Arbeit mit dem Thema "Analysis of Concurrency in Synchronous Systems" selbstständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit – einschließlich Tabellen, Karten 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 02. Juni 2014

(Unterschrift)

____

Daniel Thielsch

Abstract

This thesis concentrates on two particular issues. The first one concerns the dependency relation and its actual extraction from a synchronous Quartz program. While the definition of the dependency relation includes many subtleties, the actual finding of any dependency is accomplished by a various methods, each responsible for a certain type of dependency. More concerned with the environmental limitations and its implications towards synchronous programs is the second part of the thesis. It tries to provide solutions for running unqualified synchronous program in this type of environment and introduces a new independecy notion, suitable to the requirements of Quartz programs.

Zusammenfassung

Diese Arbeit konzentriert sich auf zwei bestimmte Aspekte. Der erste betrifft die Abhängigkeitsrelation and ihre tatsächliche Gewinning aus synchronen Quartz Programmen. Während die Definition der Abhängigkeitsrelation von vielfachen Feinheiten geprägt ist, wird das eigentliche Finden der Abhängigkeitsrelation mittles diverser Methode bewerkstelligt, die sich jeweils für einen bestimmten Typ von Abhängigkeit verantwortlich zeichnen. Der zweite Teil der Arbeit beschäftigt sich hingegen mehr mit den Limitierungen der Umwelt und deren Auswirkungen auf synchrone Programme. Während versucht wird Lösungen aufzuzeigen, die es erlauben synchrone Programme auch in dieser Art von Umwelt auszuführen, wird außerdem ein neuer Begriff der Unabhängigkeit geprägt, der entsprechend auf die Erfordernisse von Quartz Programmen zugeschnitten ist.

Acknowledgements: I would like to take the opportunity to thank both my supervisors professor Schneider and Yu Bai. They have supported me, even though the deadline had to be postponed and have allowed me to complete this thesis to the best of my knowledge. For that I'm truly grateful. On this occasion I would also like thank my parents and my sister, which have supported me since my early school days in every possible regard and have suffered enough through this thesis. I will always be happy to have you folks around.

Contents

1 Introduction **11**

1.1 Motivation 11

1.2 Outline 12

2 Related Work **13**

2.1 References 13

2.2 Synchrony 13

2.2.1 Quartz 16

2.3 Guarded Actions 18

2.4 Execution Model 20

2.5 Definitions 27

3 Dependency/Independency **31**

3.1 Dependency relation 32

3.2 SCC-Algorithms 35

3.3 Finding Dependencies 42

3.3.1 Data-flow Equations 42

3.3.2 Building dependency equations 44

3.3.3 Non-linear Solver 46

3.4 Decomposing EFSM 65

4 Implications of Asynchronous Environment **68**

4.1 Interpreter 70

4.2 Shared Reading 77

5 Summary **81**

5.1 Conclusion 81

List of Algorithms

1 Synchronous Execution Model 25
2 Clocked Synchronous Execution Model 26

3 Extended Tarjan Algorithm - Function: SCCWalk + Intialization 39
4 Extended Tarjan Algorithm - Function: ComputeSCC 40
5 Hierarchical SCC-Coloring Algorithm 41
6 Helper Function - Transformation 50
7 Helper Function - topological Sorting 51
8 Helper Function - process state environment individually 52
9 Helper Function - process state intervals individually 53
10 DoIntervalAnalysis 54
11 Fucntion 'ProcessSCC' 55
12 Helper Functions - Extracting different SCC's 60
13 additional upper, lower bounds 62

14 Basic Interpreter 72
15 Helper Functions Interpreter - Part I 73
16 Helper Functions Interpreter - Part II 74
17 Synchronous Computation of Reaction 75
18 Clocked Synchronous Execution of Reaction 76

"What we call the beginning is often the end.
And to make an end is to make a beginning.
The end is where we start from."

T. S. Eliot

1 Introduction

1.1 Motivation

It would not be exaggerated to call the embedded systems domain one of the driving forces of today's main innovations. Being used throughout most industries, ranging from avionics to industrial automation, especially the reactive systems are nowadays integral for the goal of building 'smart' devices. At their heart those 'smart' devices often consist of a different array of reactive systems, most of them becoming increasingly complex and highly integrated with one another. While this development does certainly not impose new requirements on the embedded systems domain, some have become more prevalent than others. That is, methods for handling the ongoing integration of various reactive systems or optimizations for lowering the runtime are needed more than ever. Fortunately progress has been made towards the goal of integration. As most of those reactive system agree with the semantics of the synchronous paradigm, integration usually boils down in running all synchronous reactive systems at the same clock speed. However in the endeavour to compensate for any possible communication delays between those systems, the concept of GALS architectures has been introduced. These globally asynchronous - locally synchronous architectures allow each synchronous systems to run on their own, provided they fulfill certain correctness criteria. Satisfying these criteria guarantees that the behaviours of those systems in the asynchronous environment is comparable to the behaviours of a synchronous setting or even alike. With many concepts and criteria available, some are more relevant than others, for the simple fact that they apply to a larger set of systems and allow the system to run on a higher clock speed. Gaining higher clock speeds can also achieved by exploiting inherent concurrency. To do so, the most general method lies in finding concurrent (independent) computations and utilizing those for various optimization techniques. While the usage of the dependency

relation largely depends on the used optimization, the actual finding of concurrent computations can prove a challenging task. Especially if the underlying systems inherits the synchronous paradigm and all available data-structures, including indexed expressions, are taken into account. Given these issues, it is of utmost importance to provide the formal definitions as well as algorithms, which can handle the problem of finding concurrent behaviours in synchronous programs adequately. Including the GALS architectures, it is also interesting to see if there are 'new' concepts around, which allow to abstract or dismiss certain premises, in the hope of gaining even higher clocks speeds. Both goals in mind, the following chapters are concerned with resolving aforementioned issues to a satisfiable degree.

1.2 Outline

Before diving into into the topic, a short outline will help to clarify the structure of this thesis. Chapter 2 explains the synchronous paradigm in more detail and provides the reader with a solid understanding of the synchronous language Quartz including its abstractions, the guarded actions. While those guarded actions, can be executed in more than one way, different execution models are introduced along with a set of definitions providing the formal groundwork as well as additional insights into these models. Chapter 3 on the other hand addresses the issue of finding various kinds of dependency relations for synchronous programs. Finding those dependencies requires a all kinds of algorithms, which all have to deal with a particular problem. In contrast to Chapter 3, the following Chapter 4, discards the notion of a synchronous environment for an asynchronous approach and looks at the implications this has. Since not all synchronous programs are suited to be embedded into such an environment, various classifications as well as wrapper constructs can be envisioned for handling any kind of integration. Though Chapter 5 summaries and concludes the thesis, it is followed by a small appendix, listing helpful definitions and rewrite rules in order to supplement the understanding of the reader.

2 Related Work

2.1 References

Most of this thesis is build on the foundations of Quartz and its distinct representations given in [27]. Build on the synchronous paradigm [11] and [5] provide extensive elaborations on the synchronous idea itself. In contrast papers [6],[3] and [8] are much more concerned with a particular representation known as guarded actions, while the work of [3] serves as building block for many abstractions used throughout this thesis. Considered one of the standard works when it comes to concurrency issues, [13] offers exceptional detailed solutions for any arising problem. Dependency analysis of indexed expression is handled by the works of [30] and [26]. With [26] introducing one of the most utilized dependency algorithms, [30] addresses the particular problem of finding dependencies among non-linear expressions. The source of the monotonic dependency criterium can be found in [31], while symbolical execution is the main topic of [10]. As this thesis does also contribute to the expressiveness of GALS architectures and its implications on concepts like endochrony or weak endochrony, the works of [21, 19, 9] are all highly recommended, as they provide deep insights into these rather unfamiliar notions.

2.2 Synchrony

Synchrony as a concept is not limited to computer science. It is often used on a daily basis for describing the temporal coincidence of tasks carried out. But unlike concurrency, which merely states a sort of simultaneous execution of those tasks, synchrony implies a perfect timed execution with precise beginning and ending for all tasks. At best it even requires the tasks to be uniform, not differing in their structure. Given this perspective on the synchrony, it is

easier to understand how the concept of synchrony relates to a set of computer systems, referred to as reactive systems. The computer systems, acknowledged as reactive systems, distinguish themselves from the class of transformational systems, by continuously interacting with the environment, mapping input values to output values based on the current internal state. In comparison transformational systems apply transformations only once without regard to any previous interactions. While this serves to differentiate transformational and reactive systems, the third class of systems, namely the interactive ones can be described very much the same way. Yet they differ in their interaction with the environment, as reactive systems cautiously await input triggers and are obliged to fulfill their computations in a fixed time frame whereas interactive systems provide outputs on their own account. Still the question remains how the concept of synchrony relates to the behaviour of a so called reactive system. Thinking along this line leads to consider the interactions of a reactive system. Being triggered by environmental stimuli, each interaction is bound to a specific starting and end instant of time. Because it is not unreasonable to assume a single reactive system handling at most one request at a time, all occurring interactions can be ordered decently by simply attributing each interaction its physical start and end instances and comparing them to one another. This is the point where the synchronous paradigm comes into play. The paradigm abstracts from those interactions by replacing the physical notion of time in favour for logical time instances. Each interaction is assigned one logical time instance, encompassing all the environmental inputs and the computations carried out during this interaction. Because logical time adopts the order relation imposed by physical time, computations executed in one interaction are thought to be instantaneous. This means they are thought to take 'zero' time. Keeping with the idea of environmental inputs and instantaneous processing, an interaction can also be viewed as some sort of reaction, modelling the system behaviour and being totally ordered by logical time. Build on top of these abstractions, the synchronous paradigm describes systems[19]:

- evolving along an possibly infinite sequence of reactions, totally ordered by a logical clock
- computing the outputs of all system components by reading given inputs and running computations instantaneously based on the internal

component state

- messaging all communication values synchronously between system components within the same reaction

It should be clear by now that the synchronous paradigm is certainly not unsuited for modelling the behaviour of a reactive system. Not only does the behaviour of a reactive system consist of a sequence of reaction, moreover each reaction is produced with its current state and inputs in mind. Things get interesting, if the reactive system consists of multitude of interacting components as compared to a monolithic system. While some components might depend on each other in the way, that is some produce outputs further required as inputs to different components, such issues are resolved by assuming that all inter-component messaging is done synchronously. This goes so far, that it is not required not send an output/input value at all. Though this may seem irrational, it makes perfect sense, keeping synchronous communications in mind. Sending no value can be regarded as a value itself. Since all inputs arrive at the same time, the absence of some value is easily recognizable and might be used to exercise some different reaction as opposed to the reception of some variable value. Indeed this is one way to cope with such situations, often known as 'reaction to absence'. Though there are different alternatives, it is actually best to think of absence as a real value, denoted as $\perp$ and extending the domain of each variable used throughout the system. Later on, the way of handling or not handling $\perp$ becomes a large issue in classifying synchronous systems, but for now $\perp$ is only a representative value, of indicating the concept of sending/receiving nothing (absence). With synchronous communications and instantaneous computations, the overall system reaction is produced in no time. For the creator of a reactive system, the synchronous paradigm provides some very helpful abstractions. Neither does he have to worry about interdependencies between components nor do timing issues play any significant role in defining the system behaviour. Most work can simply be spend on specification or analysis of the proposed system behaviour. However these issues become important, when a reactive system is to be run on its target architecture. As all abstractions mentioned so far must be somehow adapted to the real world, not only timing issues but also interdependencies must be found and dealt with.

2.2.1 Quartz

Although there are many imperative synchronous languages around, this thesis concentrates on a language, called Quartz. Quartz provides many sophisticated language constructs for managing synchronous behaviours, like delayed assignments or asynchronous thread execution and shares a great deal of its statement types with Esterel, a rather wide-spread synchronous language. Inheriting the synchronous paradigm, Quartz progresses along a sequence of so called macro-steps. For each macro-step the execution procedure is equivalent: All inputs are read, a series of assignments, micro-steps, are executed simultaneously and finally outputs are produced. Given a program description in Quartz, macro-steps are indicated by *pause* statements. The control-flow of a program moves from pause statement to pause statement. Everything between two pause statements belongs to the current executed macro-step and is executed in zero time. Reaching a pause statement marks the end of one macro-step and the beginning of a new one. The entry point of each Quartz program is the definition of a module with a corresponding interface, much like in other languages a 'function' or 'method' definition would look like. Yet while most 'function' definitions only specify input variables accompanied by some type information, a Quartz interface most often specify in- and output variables, along with type information and a classification called storage type . Next to type information and input/output declarations, the storage type classifies each variable as either memorized (mem) or event (evt) variable. Roughly described, it tells if a variable keeps its value over a sequence of macro-steps or is reset to some default value each macro-step. Note that variables declared as either input or output variables can only be read or written, as the interface serves as junction point between environment and program. Having defined the interface, the module body of Quartz program is build using a set of core statements, depicted in table 2.1. The explanation of these language constructs is as follows: Statement *nothing* is actually indicative of nothing and does not have any effects. *Pause* statements are employed for delimiting macro-steps and hence have direct effects on the control-flow. For assigning a value to some variable either the instantaneous assignment $x := \sigma$ or the delayed assignment $next(x) := \tau$ can be used. Either assignment is evaluated during the current macro-step, but only $x := \sigma$ is executed instantaneously,

while the assignment of *next(x)* $:= \tau$ is carried out in the following macro-step. The conditional statement *if(*σ*)* S_1 *else* S_2 checks for the boolean condition σ and executes either statement S_1 or statement S_2, depending if σ evaluates to true or false. Concatenation of statements S_1 and S_2 is done by the sequence operator *;* yielding the sequence of statements $S_1; S_2$ to be executed. If a certain statement is to be repeated, the *do S while(*σ*)* loop construct, allows for repeated execution of statement S, depending on the evaluation of the boolean condition σ. Since loop conditions might always evaluate to true, statement S must feature at least one pause statement, preventing an infinite loop from being executed without a halt. Abortion and suspension statements *abort S when(*σ*)* and *abort S when(*σ*)* are much more complex constructs. Without any weak/immediate modifiers, the abortion statement executes S and checks at the beginning of each new macro-step for boolean condition σ. If σ holds, the whole execution of S is aborted and the control-flow moves on, otherwise the control-flow inside S simply moves on to its next pause statement. The suspension works very much the same, with the small difference that the execution of S is not aborted but suspended as long as σ holds. Now for the modifiers, *weak* allows for a much more subtle control of checking σ at the beginning or end of a macro-step or allow, while *immediate* indicates if σ is already checked upon entering an abortion/suspension statement or not. Concurrent behaviour for statements S_1 and S_2 can be created by using construct $S_1||S_2$, which basically runs two threads in parallel, executing S_1 and S_2 in either thread in lockstep. As many threads can be run this way, the control-flow of a progam might actually reach several pause statements simultaneously after one logical clock tick, essentially synchronizing each running thread. Finally the last language construct allows the declaration of local variables by using $\{\alpha$ x; S$\}$. The interesting thing about local variables is, that they can be both written and read, in contrast to the ones defined in the interface. They are also of particular interest, when it comes to finding dependencies amongst macro-steps, as they largely behave like sequential programming variables do. So far the core language constructs have been covered, what's missing, is a more closer look at the way macro-steps are executed and behaviours are constructed. But to do so, another abstraction is needed, the intermediate format, known as guarded actions.

statement	explanation
nothing	empty statement
l:pause	new macro step
x := τ, next(x) := τ	initial/delayed assignment
if(σ) S_1 else S_2	conditional
$S_1; S_2$	sequence
do S while(σ)	loop
[weak] [immediate] abort S when(σ)	abortion statements
[weak] [immediate] suspend S when(σ)	suspension statements
$S_1 \vert\vert S_2$	synchronous concurrency
{α x; S}	local variable declaration

Table 2.1: Core Quartz Statements

2.3 Guarded Actions

The format known as guarded action, over a given variable set V, is defined as[27]:
$g_i(\vec{x}, \vec{b}, \vec{a}[t_i]) \Rightarrow$ (x | next(x) | a[t_k] | next(a[t_k])) := $f_i(\vec{x'}, \vec{b'}, \vec{a'}[t'_i])$
with:

- $\vec{x} \in \mathbb{Z}^{|\vec{x}|} \wedge \vec{x'} \in \mathbb{Z}^{|\vec{x'}|}$
- $\vec{b} \in \mathbb{B}^{|\vec{b}|} \wedge \vec{b'} \in \mathbb{B}^{|\vec{b'}|}$
- $t_i, t'_i, t_k : \mathbb{Z}^n \rightarrow \mathbb{N}\backslash\{0\}$
- $a_i : (\mathbb{N}\backslash\{0\}) \rightarrow \mathit{type}(a_i) \wedge a'_i : (\mathbb{N}\backslash\{0\}) \rightarrow \mathit{type}(a'_i)$
- $g_i : (\mathbb{Z}^{|\vec{x}|} \times \mathbb{B}^{|\vec{b}|} \times \mathit{type}(a_i)^{|\vec{a}|}) \rightarrow \mathbb{B}$
- $f_i : (\mathbb{Z}^{|\vec{x'}|} \times \mathbb{B}^{|\vec{b'}|} \times \mathit{type}(a'_i)^{|\vec{a'}|}) \rightarrow$ (type(x) | type(a))

This rather formal definition can be divided into two parts. First there is the so called guard, defined by function g_i. For a most general approach, g_i is assumed to depend on a large number of different typed variables. However, independent of the variables used, g_i must always evaluate to some boolean

value. That is, any variables of different type than a boolean, must be used in some kind of predicate, which itself evaluates to a boolean value. Now next to the guard there is the assignment of some variable to the function value of f_i, again depending on a vast array on variables. Yet the only condition on f_i is, that the function evaluation of f_i returns a value type-consistent to the variable, it is assigned to. With the assignment being called the action, guard and action form the so called guarded action. The execution of a guarded action is comparable to a predicated statement: If the guard g_i evaluates to true, the action, that is the assignment, must be carried out. If however g_i evaluates to false, nothing else must be done, since the action will not be executed. For a more formal definition of the various parts, a guarded actions gA is made of, see the following definitions:

- grd(gA) $:= g_i$
- rhs(gA) $:= f_i$
- lhs(gA) := x | next(x) | a[k] | next(a[k])
- action(gA) := (lhs(gA) := rhs(gA))
- wrVar(gA) := x | a
- rdVar(gA) := FV(grd(gA)) $\cup$ FV(rhs(gA)) $\cup$ (FV(lhs(gA)$\setminus$wrVar(gA))
- usedVar(gA) := wrVar(gA) $\cup$ rdVar(gA)
- usedarrayVar(gA) $:= \bigcup_i a_i[j] \cup \bigcup_i a'_i[j'] \cup a[k]$
- delayed(gA) := (lhs(gA) = next(x) $\vee$ lhs(gA) = next(a[k])) ? true : false

  with FV(f) := $\{x|x \in V \wedge$ x is a identifier in f $\}$

Definitions *grd, lhs, rhs* and *action* serve to select individual parts of the guarded actions. This is different to *wrVar, rdVar, usedVar* and *usedarrayVar*, which return the set of variables occurring in the respective parts. Furthermore all previous definitions are naturally lifted to the set of guarded actions. Later on these definitions come increasingly helpful as certain parts or variable set are scrutinized for special characteristics. But for now it is of more

interest to discuss the link between guarded actions and a Quartz program. As already mentioned, when describing guarded actions as a intermediate format, a Quartz program can be translated to an equivalent set of guarded actions. In spite of being hard to believe at first, it becomes more obvious when a Quartz program is thought to be some kind of attributed state machine. Such a attributed state machine is made up of states and states transitions plus a function that attributes each state some properties. In our case the states of this state machine can be thought as combinations of pause statements, reached after each macro-step is executed in the Quartz program. The transition relation on the other hand is thought to mimic the control-flow of the initial Quartz program, that is the sequence of macro-steps that might be executed. Yet the real behaviour of the program is determined by the micro-steps executed during each macro-step. Those can be imagined as attributes, attributing each state with a set of computations, that are run if the control-flow reaches this state. Since micro-steps are mostly concerned with assigning values to some variable, if some given condition is satisfied, their translation to the format of guarded actions is not very hard. By contrast encoding the control-flow of Quartz program into guarded actions becomes much more tiresome, as most language constructs of Quartz are tasked with diverting the control-flow. But since states are made up of labelled pause statements, the idea of assigning boolean values to those labels neatly resolves the encoding of the control-flow. No further details will be provided at this point, as the translation of Quartz to the format of guarded actions is not a trivial task and underlies many difficulties.[1]. It is perfectly sufficient to assume an equivalent set of guarded actions for each Quartz program, as each set of guarded action will from now on resemble some Quartz program.

2.4 Execution Model

Having coarsely described the execution of Quartz programs, it is time for more detailed look at the way Quartz behaviours are generated. As seen in the previous section, a Quartz program always has an equivalent representation in the form of guarded actions. Programs will therefore be thought

[1]For details see...

to be sets of guarded actions. How those guarded actions are executed is mostly predetermined by the synchronous paradigm. That is, reading input variables, executing all guarded actions synchronously and writing output variables. Though simple in description, the details of such an execution model can vary a lot. Because of that there are actually two execution models applied in Quartz, the *synchronous execution model*[6] and the *clocked synchronous execution model*. Both of course adhere to the synchronous paradigm in producing program behaviours, but have subtle differences, which requires some explaining. The synchronous execution model assumes, that input variables are always present, when a macro-step is executed. While this seems to be a reasonable approach for reactive systems, one should add that input values are not carried over to the next macro-step. Hence all input variables are effectively of storage-type event. They must be provided by the environment in each macro-step. Clearly the environment will provide some inputs, but it remains doubtful if 'every' input must be provided in each macro-step. Not only might communication between environment and system be unnecessary, it could also waste precious resources, given the case that multiple systems share the same network resources. Apart from the input variables, local and output variables are handled separately. While input variables are assumed to be read all the time, local and output variables are assumed to be written in each macro-step. That is, each macro-step must provide an value for all those variables, even though the set of guarded actions might not execute an actual assignment to the variable itself. To resolve such situations, the execution model provides a mechanism named *reaction to absence*. As soon as it is known that no guarded action of the current macro-step will assign a value to the variable, the execution model steps in and assigns a predefined value, based on the storage type of this variable. For event variables that means the assignment of the default value of its type signature, while memorized variables are more like 'ordinary' variables in getting assigned the value of the preceding macro-step. It becomes clear, that this input and output behaviour of the synchronous execution model is only one option to handle synchronous programs. An alternative to the synchronous execution model, is the clocked synchronous execution model. Primarily it extends the synchronous execution model by introducing yet another classification, which separates all variables into clocked and non-clocked variables. Clocked variables have the big advan-

tage that those variables are actually allowed to be absent, that is no variable value must be provided, but it can be provided. For input variables that means, that the environment is not obliged to provide an input value, while local and output variables do not underlie reaction to absence. If no variable value is provided or produced, the variable is simply absent. Non-clocked variables by contrast behave exactly like variables of the synchronous execution model. All non-clocked input variables must always be read anew and local and output variables underlie reaction to absence, if no value is assigned. Yet allowing the absence of variables values is not without consequences. As the programmer might not entirely be sure, that certain input values are provided or output values are produced, he needs a way to check, if a variable value is present or not. For that purpose the clocked synchronous execution model introduces a function $Clk : V \rightarrow \mathbb{B}$, which takes as an argument any variable of the program and determines if there is a value or not. Using this function, the programmer can now check the 'clock' of a variable, before actually accessing the variable value. This is indeed very important, as absence $\perp$ is not a value per se, but only used for modelling purposes. Accessing a variable, whose clock is false, might yield an arbitrary value and is very much comparable to invalid memory accesses in usual programs. Since the results are always unknown, such cases are strictly forbidden and must be ruled out from the beginning. This concept of accessing variables values, only if the clock of the variable is true, is called clock consistency. Only clock consistent programs are considered to be executed by the clocked execution model[2].

On top of clock consistency, all programs executed are also thought to be causally correct. The term causal correctness 'assures that for all reachable states and all possible inputs, the micro steps can be executed in an ordering where all values are known when needed'[27]. Basically this says, that the guarded actions of all macro-steps can always be ordered according to their data-flow. Of two guarded actions, one writing a variable and the other reading this variable, the first one must be executed first. If however guarded actions depend on each other, the reading of input values or potential reactions to absence will resolve any dependencies for sure. Programs, for which such

[2]The formal definition of clock-consistency is the following:
$\forall s \in S. \forall r \in Rct(\mathcal{L}(s)). \forall v \in V : used_r^s(v) \rightarrow [Clk(v)]_r$
with $used_r^s(v) := \begin{cases} true & \exists g \in \mathcal{L}(s) : ([grd(g)]_r = false \wedge v \in grdVar(g)) \vee ([grd(g)]_r = true \wedge v \in usedVar(g)) \\ false & otherwise \end{cases}$

orderings do not exist, are not considered causally correct[3]. Both execution models require programs to be causally correct, otherwise they are disregarded. The same holds for programs, writing variables twice during a macro-step. As a variable can only carry one value during macro-step, programs allowing for multiple assignments to happen are discarded from both execution models.

Now given a set of guarded actions, both execution models, depicted in algorithm 1 and 2, follow the same outline. After an short initialization phase (1), setting up so called variable environments and defaults clocks for the clocked model, each macro step consists of the following steps: First of all, current input variables are read in and combined with the set of delayed assignments of the previous macro-step. Based on these input values, the clocked model derives the clocks of all input variables and keeps them stored (2). Having set up the current clocks and environments, the processing of the guarded actions starts. Running over all guarded actions, the execution model only executes the instantaneous assignments, if the corresponding guard evaluates to true. If a variable is indeed written, the variable environment is updated and its clocked is set to true. Beyond that, non-clocked variables of both execution models underlie the reaction to absence. If no assignment can take place, either the default value or the value of the previous variable environment are assigned to this variable. This whole process is repeated as long as there are changes to the variable environment or the clocks (3). As soon as no changes occur anymore, all guarded actions with delayed assignments get executes (4). Delayed assignments however are written into a different variable environment as they only effect the next macro-step. After executing them too, the last step only involves checked, if checking for clock consistency, broken assertions or if the produced environment contains invalid values (5). If all went right, current values of the output variables are given to the environment and the next macro-step is being started (2). With both execution models sufficiently detailed in algorithms 1 and 2, some small comments shall be made on the clocked synchronous execution model. First it should be noticed, that the

[3]The requirement for causal correctness comes from the fact, that all execution models ultimately run on hardware, which on the very basic level execute computations sequentially. At least this holds for most of today's hardware architectures, but might change in the future with ongoing research toward 'new' hardware architectures

clocks of all clocked variables are implicitly implied by the variable values. That is if a variable is read in or written, the clock is accordingly set to true. However that is only way to set the clock of a variable. A different way could be to provide the clocks of all clocked variables as inputs to the programs. While this might lead to clock-inconsistent behaviours, if not handled properly, it might also allow to write clocked output variables, similar to the reaction to absence for non-clocked variables. It remains undecided for this thesis, if such behaviours are more constructive than others, yet the point is that there are several approaches in providing clocks and each comes with a different consequences.

Algorithm 1 Synchronous Execution Model

function SIMULATEGUARDEDACTIONS
(1) initialization
$\forall x \in V$: $\epsilon^{cur}(x) :=$ default(x);
$\epsilon^{nxt} := \emptyset$;
repeat
(2) begin of macro-step
$\epsilon^{prv} := \epsilon^{cur}$;
$\epsilon^{cur} := \epsilon^{nxt} \sqcup$ ReadInputValues();
$\epsilon^{nxt} := \emptyset$;

(3) immediate actions
repeat
$\epsilon' := \epsilon^{cur}$;

for all $< \gamma \Rightarrow x := \sigma >$ **do**
if $([\gamma]_{\epsilon^{cur}} = true) \wedge ([\sigma]_{\epsilon^{cur}} \notin \{\bot, \top\})$ **then**
$\epsilon^{cur}(x) := \epsilon^{cur}(x) \sqcup [\sigma]_{\epsilon^{cur}}$;
end if
end for
for all $x \in V$ **do**
if $\forall < \gamma \Rightarrow x := \sigma >$: $([\gamma]_{\epsilon^{cur}} = false)$ **then**
$\epsilon^{cur}(x) := \epsilon^{cur}(x) \sqcup$ (type(x) == evt ? default(x) : $\epsilon^{prv}(x)$);
end if
end for
until $(\epsilon^{cur} \neq \epsilon')$

(4) delayed actions
for all $< \gamma \Rightarrow next(x) := \sigma >$ **do**
if $([\gamma]_{\epsilon^{cur}} = true) \wedge ([\sigma]_{\epsilon^{cur}} \notin \{\bot, \top\})$ **then**
$\epsilon^{nxt}(x) := \epsilon^{nxt}(x) \sqcup [\sigma]_{\epsilon^{cur}}$;
end if
end for

(5) end of macro-step
if $\exists x \in V$: $\epsilon^{cur}(x) \in \{\bot, \top\}$ **then** Fail();
end if
if $\exists < \gamma \Rightarrow assume(\sigma) >$: $[\gamma \wedge \neg\sigma]_{\epsilon^{cur}} = true$ **then** Fail();
end if
if $\exists < \gamma \Rightarrow assert(\sigma) >$: $[\gamma \wedge \neg\sigma]_{\epsilon^{cur}} = true$ **then** Fail();
end if
WriteOutputs(ϵ^{cur});
until (true)
end function

Algorithm 2 Clocked Synchronous Execution Model

function SimulateClockedGuardedActions
(1) initialization
$\forall x \in V \backslash V_c$: $\epsilon^{cur}(x) :=$ default(x);
$\forall x \in V \backslash V_c$: $Clk^{def}(x) :=$ default(x);
$\epsilon^{nxt} := \emptyset$;
$Clk^{nxt} := Clk^{def}$;
repeat
(2) begin of macro-step
$\epsilon^{prv} := \epsilon^{cur}$;
$\epsilon^{cur} := \epsilon^{nxt} \sqcup$ ReadInputValues();
$\epsilon^{nxt} := \emptyset$;
$Clk^{prv} := Clk^{nxt}$;
$Clk^{cur} := Clk^{nxt} \sqcup$ DeriveInputClocks(ϵ^{cur});
$Clk^{nxt} := Clk^{def}$;

(3) immediate actions
repeat
$\epsilon' := \epsilon^{cur}$;
$Clk' := Clk^{cur}$;

for all $< \gamma \Rightarrow x := \sigma >$ **do**
if $([\gamma]_{\epsilon^{cur},Clk^{cur}} = true) \wedge ([\sigma]_{\epsilon^{cur},Clk^{cur}} \notin \{\bot, \top\})$ **then**
$\epsilon^{cur}(x) := \epsilon^{cur}(x) \sqcup [\sigma]_{\epsilon^{cur},Clk^{cur}}$;
$Clk^{cur}(x) := Clk^{cur}(x) \sqcup true$;
end if
end for
for all $x \in V$ **do**
if $\forall < \gamma \Rightarrow x := \sigma >$: $([\gamma]_{\epsilon^{cur},Clk^{cur}} = false \wedge Clk^{cur}(x) = true \wedge \epsilon^{cur}(x) = \bot)$ **then**
$\epsilon^{cur}(x) := \epsilon^{cur}(x) \sqcup$ (type(x) == evt ? default(x) : $\epsilon^{prv}(x)$);
end if
end for
until $(\epsilon^{cur} \neq \epsilon' \wedge Clk^{cur} \neq Clk')$

(4) delayed actions
for all $< \gamma \Rightarrow next(x) := \sigma >$ **do**
if $([\gamma]_{\epsilon^{cur},Clk^{cur}} = true) \wedge ([\sigma]_{\epsilon^{cur},Clk^{cur}} \notin \{\bot, \top\})$ **then**
$\epsilon^{nxt}(x) := \epsilon^{nxt}(x) \sqcup [\sigma]_{\epsilon^{cur},Clk^{cur}}$;
$Clk^{nxt}(x) := Clk^{nxt}(x) \sqcup true$;
end if
end for

(5) end of macro-step
if $\exists < \gamma \Rightarrow A > .\exists x \in Var(\gamma)$: $\epsilon^{cur}(x) \in \{\bot, \top\}$ **then** Fail();
end if
if $\exists x \in V$: $\epsilon^{cur}(x) \in \{\bot, \top\} \wedge Clk^{cur}(x) = true$ **then** Fail();
end if
if $\exists < \gamma \Rightarrow clk(x) := \sigma >$: $[\gamma]_{\epsilon^{cur},Clk^{cur}} = true \wedge [\sigma]_{\epsilon^{cur},Clk^{cur}} \neq \bot$ **then** Fail();
end if
if $\exists < \gamma \Rightarrow assume(\sigma) >$: $[\gamma \wedge \neg\sigma]_{\epsilon^{cur},Clk^{cur}} = true$ **then** Fail();
end if
if $\exists < \gamma \Rightarrow assert(\sigma) >$: $[\gamma \wedge \neg\sigma]_{\epsilon^{cur},Clk^{cur}} = true$ **then** Fail();
end if
WriteOutputs($\epsilon^{cur}, Clk^{cur}$);
until (true)
end function

2.5 Definitions

After the groundwork has been laid by the (clocked) synchronous execution model, it is about time to formalize many notions and concepts introduced in the previous sections. Since most upcoming sections build on those definitions, it is best to keep them as formal as possible. Now the set of variables V, for any given set of guarded actions, can be separated into three subsets $V := V^{in} \cup V^{out} \cup V^{loc}$, namely the set of input, output and local variables. Each macro-step of program is sufficiently described by its variable environment, which is called an reaction $r : V \rightarrow (\mathbb{B}^n \cup \mathbb{Z}^n \cup \{*, \perp, \top\}), n \in \mathbb{N}$. A reaction is simply a function assigning each variable a type-consistent value or the values unknown ($*$), absent ($\perp$) or invalid ($\top$). Value unknown $*$ indicates that a variable is not used at all in a program, while an invalid $\top$ value shows that some assignment has been executed the wrong way. The signature of a reaction $sig(r) := \{v \in V | r(v) \notin \{*, \top\}$ on the other hand denotes those variables that are valid for the reaction and $type(v)$ assigns each used variable its actual domain, boolean, integer and so on. Given some reaction r and some function f, $[f]_r$ is a shortcut for evaluating f by assigning the variable values of r to the variables used in f. It was already mentioned that given some reaction r, a guarded action is satisfied that is $[g]_r = true$ iff $([g]_r = true \wedge [rhs(g)] \in type(wrVar(g))) \vee [grd(g)]_r = false$, which means either the guard is satisfied and the action type-consistent or the guard evaluates to false. Reactions r are considered valid for some set of guarded actions P, that is $r \in Rct(P) :\Leftrightarrow sig(r) = Var(P) \wedge \forall g \in P : [g]_r = true$, iff they confirm with the signature and satisfy all guarded actions in P. In consequence behaviours, are nothing more than a sequence of valid reactions, that is for a guarded actions P, the behaviour of P is defined as $bhv(P) := \{(r^t)_{t \leq n} | n \leq \infty \wedge \forall t \leq n : sig(r^t) = Var(P)\}$. Rather useful is also the projection of a reaction r on a subset $V' \subseteq V$, $namelyr|^{V'}$, which is defined as $\forall V \in V' : r|^{V'}(v) := \begin{cases} r(v) & v \in V' \\ \perp & otherwise \end{cases}$

This can be extended to behaviours p, with $p|^{v'} := (r^t|^{V'})_{t \leq n}$. A special case of a projection is the projection of a behaviour p on a single variable $v \in V$, which is a called $s_v := p|^{\{v\}} := (r^t|^{\{v\}})_{t \leq n}$, the signal of v. Because a signal is only relying on one variable, it basically returns for each time instance t, the

corresponding value $s_v(t)$. For a signal s_v over some variable $v \in V$, the clock of s_v is defined as $\forall t \in \mathbb{N} : Clk(t) := \begin{cases} true & s_v(t) \in type(v) \\ false & otherwise \end{cases}$
As seen in the section 2.4, reactions are build by an iterating process, running over the set of guarded actions and updating the variable environment until the environment stays fixed. Interestingly this staying fixed is nothing more than a fixpoint, which is based on the partial order of all variable values plus the values unknown, absence and invalid, that is $\forall x \in (\mathbb{B} \cup \mathbb{Z}) : * \leq \bot \leq x \leq \top$. This partial order forms a finite lattice and is even preserved if extended variable wise to reactions, that is $r_1 \leq r_2 :\Leftrightarrow \forall x \in V : r_1(x) \leq r_2(x)$. Doing so, makes the computation of any reaction continuous, as it now also forms a finite lattice and terminates with a fixpoint, if iterated properly.[4]
Aside from partial orders and definitions regarding reactions, it was explained that each Quartz program is equivalently represented by a set of guarded actions. The main idea behind this intermediate format, is to think of a synchronous Quartz program as some kind of state machine. While the set of guarded actions incorporates the micro-steps run in each state, it also abstracts from the control-flow itself by defining the complete control-flow in terms of guarded actions, assigning values to state labels and predicating associated guarded actions appropriately. This leads to the situation, that executing a program by a execution model always requires to execute the "whole" set of guarded actions. From the point of an intermediate format this is fine abstraction, however this embedding of the control-flow complicates eventual optimization efforts, where relations between guarded actions of different states are concerned. Addressing and resolving this particular issue, requires the use of a so called *extended finite state machine* (EFSM)[4]. An EFSM manages to keep the control-explicit, utilizing states and a transition relation, just as an ordinary finite state machine does. Yet it attributes each state with a set of guarded actions, only executed in this particular state. Therefore it is called an 'extended' finite state state machine. The definition of an EFSM M, over the set of guarded actions gA is the following:
M := (s_0, S_M, T_M, $\mathcal{L}$) with

$s_0 \in S_M$, the starting state s_0

[4]It is even possible to extend this relation to behaviours $p_1 \leq p_2 :\Leftrightarrow \forall t \in \mathbb{N} : r_1^t \leq r_2^t$, but of more practical use is the lattice for reactions.

S_M :=$\{s_0, \ldots, s_n\}$, the set of states S

$T_M \subseteq S \times C \times S$, the conditional transition relation T

$C : (\mathbb{Z}^n \times \mathbb{B}^n) \to \mathbb{B}$, the condition for each transition t, defined over a large set of variables

$\mathcal{L} : S_M \to gA$, attributing each state its associated guarded actions

All transitions of an EFSM M are conditional, that is for $t \in (s_i, c_i, s_{i+1})$, condition c_i must hold in state s_i for the transition t to happen. As the control-flow in an EFSM M is explicitly represented, the control-flow graph (CFG) of M, namely CFG_M can be defined as CFG_M := (V, E) with V := S_M and $\forall t := (s_i, c_i, s_j) \in T_M \Rightarrow (s_i, s_j) \in E$. That is the control-flow graph CFG_M is made up of the set of vertices (nodes) V and the set of edges E. With V resembling the state set S_M, the set of edges mimics the transition relation T minus the conditional guards c_i. CFG's are very helpful for analysing the structure of a program. They show if there are states repeatedly executed, essentially forming a loop, or if certain states can even be reached from the starting state. As most optimization techniques require such information, a whole section of this thesis is devoted to analysing loop structures in general. Talking about CFG's and transition relations, makes it necessary to talk a little bit about graph specific concepts. For starters, given any graph G := (V, E), a path $\pi := v_i \cdot v_{i+1} \cdot \ldots \cdot v_k$ denotes a sequence of states such that for all states, $\forall_{i \leq j \leq k} v_j \in V \wedge \forall_{i \leq j \leq k} (v_j, v_{j+1}) \in E$, the states are in V and two succeeding states are connected via an edge. A path π of the form $\pi := v_i \cdot \ldots \cdot v_i$, that starts and ends in the same node v_i contains a cycle (loop). If a path π contains only distinct nodes, that is $\pi := v_i \cdot \ldots \cdot v_k$ with $\forall_{i \leq j_1, j_2 \leq k} (v_{j_1} \neq v_{j_2})$, then the path is said to be simple. If a path contains a cycle, but removal of start or end node yields an simple path, it is called a simple cycle. A node v_j is called reachable from node v_i, iff there exists a path $\pi := v_i \cdot \ldots \cdot v_j$ from v_i to v_j in the graph. In addition the set of nodes $V' \subseteq V$ is called strongly connected iff for any two nodes $v_i, v_j \in V. \exists (\pi := v_i \cdot \ldots \cdot v_j \wedge \pi' := v_j \cdot \ldots \cdot v_j)$ with paths π, π' only consisting of nodes of V'. In different words, that means that all vertices of V' must be reachable from one another via a path lying entirely in V'. More the type of a characterization is, that in the context of an edge e := (v_i, v_j) the node v_i is said to be the predecessor of v_j, while v_j is called

the successor of v_i. Based on the definitions of paths, CFG's and EFSM's, the final behaviour of a single EFSM M is defined by

$$\text{Bhv(M)} := \bigcup_{\exists \pi_i \in CFG_M} Bhv(\pi_i)$$

with $\text{Bhv}(\pi) := \{(r^t)_{t \leq n} | \forall t \leq n : r^t \in Rct(\pi^t) \wedge \exists(\pi^t, c_t, \pi^{t+1}) : [c_t]_{r^t} = true\}$ for CFG_M.
Note that as much as a Quartz program is represented by a certain set of guarded actions, this set of guarded actions can be equivalently represented by an EFSM. While the actual transformation from guarded actions to an EFSM is not entirely straight-forward, due to the extraction of the control-flow, the transformation will be omitted at this point as the general idea has been given. It is far more interesting to see, that a synchronous Quartz program can actually have three different representations, all indistinguishable on behavioural terms. Although any representation can be chosen, the EFSM representation is more suited for optimization analysis, as it clearly separates guarded actions of different states and also incorporates any possible loop structures of the CFG. A fact, which should not be neglected, since Quartz representations of programs might include cycles though no loop language constructs (do-while loop) have been used.

3 Dependency/Independency

As already mentioned in the beginning of the thesis, one of the main goals of this thesis is to provide methods and abstractions, for finding the dependency relation of a synchronous Quartz program. The following sections will therefore define the different notions of dependency, mostly in reference to guarded actions, as well provide different algorithms for finding them. Any Quartz program is assumed to be given in its EFSM representation and may even utilize arrays, that is indexed variables. While this may sound rather obvious, note that most algorithms dealing with the finding of dependency relations disregard any indexed variables. Even if they do, those index expressions are nearly always assumed to be linear in mathematical terms, with any occurring variables of this expression having clear defined upper and lower bounds. By comparison the methods presented here, will not rely on such preconditions. Neither are variables required to bounded upfront nor do index expression have to be linear. Non-linear expressions are handled just as well as linear expression as long as there are no symbolic constants. Unfortunately the downside of such an most general approach is that it can not compare to algorithms which are restricted to handling special cases. Tailored towards those special cases, these algorithms perform much more efficient, when proving or disproving dependencies than the algorithms presented here. But since they have been developed with sequential programming languages in mind and rely heavily on certain assumption they are not really the best choice to finding all dependencies of a synchronous programs. Based on this understanding it is therefore best to provide a most general technique, as done in the following sections, while resorting to some of the more efficient sequential algorithms, if their requirements are met.

Given the states s_1, s_2 s.t. $(s_1, s_2) \in T$ and with guarded actions

	s_1	s_2
gA_1^s	$x \Rightarrow o_1 :=$ true	$x \wedge \neg l \Rightarrow a[0] :=$ true
gA_2^s	$\neg x \wedge \neg l \Rightarrow o_2 :=$ true	$\neg x \wedge a[0] \Rightarrow o_1 :=$ true
gA_3^s	$\neg x \Rightarrow l :=$ false	$l \Rightarrow o_2 :=$ true

over the variables set V $:= \{x\}^{in} \cup \{o_1, o_2\}^{out} \cup \{a, z\}^{loc}$.

Then the following dependencies can be found:

dependency	explanation
$gA_i \Leftrightarrow^s gA_j$	holds for $gA_1^{s_2} \Leftrightarrow_a^s gA_2^{s_2}$, as $a \in V^{loc}$ and $a \in wrVar(gA_1^{s_2}) \cap grdVar(gA_2^{s_2})$
$s_i \leq^s s_j$	holds for $s_1 \leq_l^s s_2$ and $s_1 \leq_{o_2}^s s_2$
$gA_i \Leftrightarrow^b gA_j$	holds for $gA_2^{s_1} \Leftrightarrow_l^b gA_2^{s_3}$, as for $\neg x$, $l \in wrVar(gA_2^{s_1}) \cap grdVar(gA_3^{s_1})$
	but $gA_1^{s_2} \not\Leftrightarrow_a^b gA_2^{s_2}$ does not hold, since $[grd(gA_1^{s_2}) \wedge grd(gA_2^{s_2})]$ unsatisfiable
$s_i \leq^b s_j$	holds for $s_1 \leq_l^b s_2$, $l \in wrVar(s_1) \cap grdVar(s_2)$
	but $s_1 \not\leq_{o_2}^b s_2$ does not hold, since $o_2 \notin wrVar(s_1) \cap wrVar(s_2)$

Figure 3.1: Examples for dependency relation

3.1 Dependency relation

Although a great deal has been talked about dependency relations, now thorough definition has been provided up to this point, explaining how dependency relates to synchronous programs. For that purpose actually two dependency relations are defined. The first one is named syntactic dependency and is only concerned with the appearance and classification of variables among guarded actions or composing states. Quite on the opposite lies the second one, named behavioural dependency, which is concerned to define all dependencies, which are bound to occur in some program execution. While maybe not entirely obvious, syntactic dependency defines all possible dependencies, whereas behavioural dependency actually checks if there are behaviours of the program, such that a dependency can exist. Given that, the syntactic dependency is defined as following: For a set of guarded actions gA $:= \{gA_1, \ldots gA_n\}$ over Var(gA), gA_i, gA_j are said to be syntactical-dependent on some variable v $\in$ Var(gA), that is
$gA_i \Leftrightarrow_v^s gA_j$, iff

$(v \in V^{in} \rightarrow v \in (rdVar(gA_i) \cap rdVar(gA_j))\wedge$
$(v \in V^{loc} \rightarrow (v \in Var(gA_i) \cap Var(gA_j) \wedge v \notin wrVar(gA_i) \cap wrVar(gA_j)))$
Given some state set S := $\{s_1, \ldots, s_n\}$ with $s_i := \{gA_{i1}, \ldots, gA_{in_i}\}$ over Var(S) with transition relation $T \subseteq S \times S$, then s_i is said to be syntactical-dependent on s_j over variable v, that is $s_i \leq^s_v s_j$, iff
$\exists \pi := s_i \ldots s_j : \forall_{i \leq k < j}(s_k, s_{k+1}) \in T.$
$\exists gA_{ix} \in s_i.\exists gA_{jy} \in s_j.\forall v \in Var(S)$:
$v \in V^{in} \rightarrow gA_{ix} \Leftrightarrow^s_v gA_{jy}\wedge$
$v \in V^{out} \rightarrow v \in (wrVar(gA_{ix}) \cap wrVar(gA_{jy})\wedge$
$v \in V^{loc} \rightarrow v \in (wrVar(gA_{ix}) \cup wrVar(gA_{jy})$
As mentioned the syntactic dependency is solely restricted to sets of variables occurring in guarded actions and consequently in states. It does not consider however, if those guarded actions are satisfiable, or if a certain state is actually reachable by some behaviour or not. Those things are covered by the definition of behavioural dependency.

The definition of behavioural dependency is given by: Given a set of guarded actions gA = $\{gA_1, \ldots, gA_n\}$ over Var(gA), gA_i, gA_j are said to be behavioural-dependent over a non-indexed variable v, that is $gA_i \Leftrightarrow^b_v gA_j$, iff
$gA_i \Leftrightarrow^s_v gA_j\wedge$
$\exists r_i \in Rct(\{gA_i\}).\exists r_j \in Rct(\{gA_j\}) : [gA_i]_{r_i \sqcup r_j} = [gA_j]_{r_i \sqcup r_j} = true\wedge$
$(r_i(v) = r_j(v)) \neq \bot$
over an indexed variable v[k], $gA_i \Leftrightarrow^b_{v[k]} gA_j$, iff
$gA_i \Leftrightarrow^s_v gA_j\wedge$
$\exists t_i, t_j \in Expr$: $(v[t_i] \in usedarrayVar(gA_i) \wedge usedarrayVar(gA_j))\wedge$
$(\exists r_i \in Rct(\{gA_i\}).\exists r_j \in Rct(\{gA_j\})$:
$[gA_i]_{r_i \sqcup r_j} = [gA_j]_{r_i \sqcup r_j} = true\wedge$
$[t_i]_{r_i} = [t_j]_{r_j} = k\wedge$
$([v[t_i]]_{r_i} = [v[t_j]]_{r_j}) \neq \bot)$
For array variables v it follows that
$gA_i \Leftrightarrow^b_v gA_j$, iff $\exists k \in \mathbb{N} : gA_i \Leftrightarrow^b_{v[k]} gA_j$

Given a state set S := $\{s_1, \ldots, s_n\}$ with $s_i := \{gA_i 1, \ldots gA_{in_i}\}$ over Var(S) with conditional transition relation $T \subseteq S \times C \times S$, then s_i is said to be behavioural-dependent on s_j over non-indexed variable v, that is $s_i \leq^b_v s_j$, iff

$\exists \pi := s_i \dots s_j . \exists b \in Bhv(\pi)$: $(\forall_{i \leq k < j} (s_i, c_i, s_{k+1}) \in T \wedge [c_k]_{b^k} = true) \wedge$
$s_i \leq_v^b s_j \wedge$
$b_i(v), b_j(v) \notin \{\bot, \top\}$
over an indexed variable v[k], $s_i \leq_{v[k]}^b s_j$, iff
$\exists \pi := s_i \dots s_j . \exists b \in Bhv(\pi)$: $(\forall_{i \leq k < j} (s_i, c_i, s_{k+1}) \in T \wedge [c_k]_{b^k} = true) \wedge$
$s_i \leq_v^b s_j \wedge$
$\exists gA_{ix} \in s_i . \exists gA_{jy} \in s_j$:
$\exists t_i, t_j \in Expr$: $(v[t_i] \in usedarrayVar(gA_{ix}) \wedge usedarrayVar(gA_{jy})) \wedge$
$([v[t_i]]_{b^i}, [v[t_j]]_{b^j}) \neq \bot)$
$[t_i]_{r_i} = [t_j]_{r_j} = k$
For array variables v it holds that
$s_i \leq_v^b s_j$, iff $\exists k \in \mathbb{N} : s_i \leq_{v[k]}^b s_j$

For a direct dependency relation from some state s_i to some state s_j[1] over variable v, such that $s_i \leq_v^* s_j$, the dependency condition must be extended by $\wedge \forall_{i<k<j} s_k$: $s_i \not\leq_v^* s_k \wedge s_k \not\leq_v^* s_j$.
Note that the general dependency relation can be constructed from the direct dependency relation by applying transitive closure.

Now both dependency relations have been defined, without any particular execution model in mind. The reason for that is, that each execution model introduces different dependencies by his input/output behaviour. While the synchronous model requires all input variables to be read and all output variable to be written in each state, the same thing can only be said for the non-clocked variables of the clocked synchronous model. Hence a different set of dependencies is introduced by both models on top of the ones, already given by the program. Additional dependencies imposed by the corresponding execution model:
$\rightarrow$ synchronous execution model:
Constraints:

- read all inputs variables in every state

  $\overset{syntac.}{\rightarrow} \forall v \in V^{in} . \forall s_i, s_j \in S . \exists \pi := s_i \dots s_j$: $s_i \leq_v^s s_j$

  $\overset{behav.}{\rightarrow} \forall v \in V^{in} . \forall s_i, s_j \in S . (\exists \pi := s_i \dots s_j \wedge \exists b \in Bhv(\pi)$: $s_i \leq_v^s s_j$

[1]As the concept of direct dependency can be applied to both the syntactical as well as behavioural dependency the following definitions simply use a * for both types

- write all output variables in every state

  $\overset{syntac.}{\rightarrow}$ $\forall v \in (V^{loc} \cup V^{out}.\forall s_i, s_j \in S.\exists \pi := s_i \dots s_j\colon\ s_i \leq_v^s s_j$

  $\overset{behav.}{\rightarrow}$ $\forall v \in (V^{loc} \cup V^{out}).\forall s_i, s_j \in S.(\exists \pi := s_i \dots s_j \wedge \exists b \in Bhv(\pi)\colon\ s_i \leq_v^s s_j$

$\rightarrow$ clocked synchronous execution model:
Constraints:

- read all non-clocked variables in every state
- write all non-clocked variables in every state

  $\overset{syntac.}{\rightarrow}$ $\forall v \in (V \backslash V^c).\forall s_i, s_j \in S.\exists \pi := s_i \dots s_j\colon\ s_i \leq_v^s s_j$

  $\overset{behav.}{\rightarrow}$ $\forall v \in (V \backslash V^c).\forall s_i, s_j \in S.(\exists \pi := s_i \dots s_j \wedge \exists b \in Bhv(\pi)\colon\ s_i \leq_v^s s_j$

As previously told, all additional constraints of the clocked execution model are restricted to non-clocked variables. Programs utilizing only clocked variables are not constrained by those restrictions and are more rewarding towards concurrency, as even whole macro steps might be independent. Hence most of the following examples and methods will assume a clocked synchronous execution model, since it simply allows for more concurrency.

3.2 SCC-Algorithms

While dependencies between guarded actions are rather easy to find, the real challenge lies in checking state dependencies, as this involves checking paths for some transition relation. Problems arise, if the control-flow graph of the states (CFG), does not contain only simple paths but also paths with cycles. Cycles make it possible that a finite program keeps running on forever or that simple computations iterated over some time, calculate incredibly complex formulas. Yet they can lead to situations where states only become dependent after they

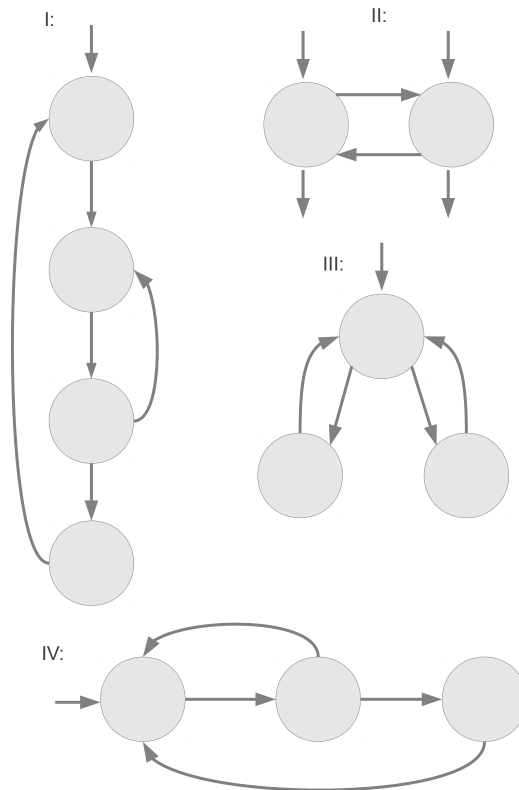

Figure 3.2: Different control-flow graph with particular structures

Example:	Extendend Tarjan:	SCC-Coloring
I	x	x
II	x	
III	x	
IV	x	x

Figure 3.3: Examples for structures handled by both extended tarjan algorithms or SCC-coloring algorithm

have been visited a certain number of times. These so called loop-carried dependencies are not easy to find, as any cyclic path in the CFG theoretically spawns an infinite number of cyclic paths. Thankfully, most computations are only iterated a certain number of times, as they depend on conditions to be fulfilling. Finding these conditions and utilizing them accordingly for dependency analysis will be the task of the non-linear dependency solver. For him to work, it is necessary to know how simple cycles in the CFG can be found and how they relate to each other, that is cycles may encompass cycles, making it an inner or nested cycle. Simple cycles in any CFG are made up of a finite number of nodes. They are the building blocks of all cyclic paths and hence perfectly sufficient for analysis. Finding them is usually accomplished by using so called strongly connected components (SCC's). Those are sets of nodes, which are strongly connected, that is each node can reach every other node inside the set. The interesting thing about strongly connected components is, that SCC's group all the nodes, forming a simple cycle, together. While there are many methods for finding the SCC's of a graph, they all opt to return only the largest SCC's. Any SCC's inside another one is simply neglected. This particular issue is resolved by the following algorithms, named *Extended Tarjan's Algorithm* and *Hierarchical SCC-Coloring Algorithm*. The extended tarjan algorithm 3.2 based on tarjan's algorithm[29]. The algorithm starts with the entry state and utilizes a recursive depth-first search for traversing the nodes of the graph. Each state visited, is put on a stack and numbered according to the order of encountered states. The key idea is, that each state is only kept on the stack, as long as it has a path to some state encountered earlier. If that is not the case, the stack is removed by as many states as necessary, such that the topmost state again has a path to some earlier state. The numbering of the states due to their visiting order is extremely helpful as it allows to detect, if the successor of some state has already been visited earlier and still resides on the stack, which implies a strongly connected component. The nodes popped from the stack, until the stack condition is again satisfied, resemble one strongly connected component. While the original tarjan algorithm stops here at handling these nodes any further, the extended algorithm applies the original tarjan algorithm again on these nodes, searching for any nested SCC's. But before this is done, the algorithm is required to delete so called back edges. Those are edges connecting nodes of the SCC with all entry

nodes of the SCC. Though it would be more accurate do delete all paths leading from an exit node of the SCC to an entry node of the SCC, this checking for corresponding paths proves to be far to computational intensive. With those back-arcs deleted and the set of edges projected onto the set of nodes of the SCC, the original tarjan algorithm is again invoked and starts looking for more SCC's. The great advantage of the extendend tarjan algorithm is that all found SCC's are not required to share any structural similarities. If there is a unique entry node or a large number of entry nodes, it makes no difference to the algorithm. This is quite different to most algorithms dealing with loop structures in control-flow graphs, as they often assume single entry nodes and single exit nodes. Adopting these requirements, a much more simpler version of the tarjan extended algorithm, the hierarchical SCC-coloring algorithm 3.2 can be formulated. Besides the fact, that both algorithm search for the set of all nested SCC's, both do not share any significant structural similarities. Where tarjan's extended algorithm has to recursively check each new found SCC for nested components, the coloring algorithm must traverse the graph only once. If a node is identified to a be a single entry node of an SCC [2], all its non-visited predecessor nodes are marked as exit nodes. Though the assumption clearly states that only one exit node is allowed for a SCC, an entry node might be shared by nested SCC's (the same holds for exit nodes) , making it the successor of several exit nodes at once. As soon as a node is marked as exit node, the exit node as well as its accompanying entry node are given as inputs to the second algorithm *backpropagation*. The algorithm backpropagation starts traversing the 'reversed CFG'[3] from the exit node, until it encounters its corresponding entry node. Every node traversed during this search is colored commonly, to highlight its membership to this particular SCC. After the reversed CFG has been traversed for every exit node, each set of nodes commonly colored is designated as a SCC and the union of those SCC's returned as result. While depicted in algorithms 3.2 and 3.2, example 3.3shows which structures are actually handled by both algorithm and which are prone to fail with the SCC-coloring algorithm.

[2]That is it has a connection to an already visited node, but also possesses an predecessor node not visited yet

[3]Reversing a graph, means keeping the set of vertices V, but replacing each edge e := (s_1, s_2) with (s_2, s_1), that is reversing the direction of the edge

Algorithm 3 Extended Tarjan Algorithm - Function: SCCWalk + Intialization

```
for all v ∈ V do
    lhNode[v] := false;
    baNode[v] := false;
    level[v] := 0;
end for
function SCCWalk(V,E)
    SCC_v := ∅
    σ := new Stack();
    i := 1;
    for all v ∈ V do
        L[v] := 0;
        pre[v] := 0;
    end for
    while ∃v ∈ V : pre[v] = 0 do
        ComputeSCC(v);
    end while
end function
```

Algorithm 4 Extended Tarjan Algorithm - Function: ComputeSCC

```
function COMPUTESCC(v)
    pre[v] := i;
    i := i + 1;
    L[v] := pre[v];
    push x onto σ
    for all w ∈ succ[v] do                                   ▷ Tarjan's algorithm
        if pre[w] = 0 then
            ComputeSCC(w);
            if L[w] < L[v] then
                L[v] = L[w];
            end if
        end if
        if pre[v] < pre[w] then
            Nop;                                             ▷ do nothing
        else
            if w is on σ ∧ pre[w] < L[v] then
                L[v] = pre[w];
            end if
        end if
    end for
    if L[v] = pre[v] then
        C = ∅;
        repeat
            pop z from σ
            C = C ∪ {z};
        until (z = v)
        if |C| ≥ 2 ∨ (|C| = 1 ∧ v ∈ pred[v]) then
            SCC_v = SCC_v ∪ {C};
            E_c = E;
            for all n ∈ C do                                 ▷ delete backarcs
                if (∃n′ ∈ pre(C) ∩ G\C) ∨ (pre(n) = ∅) then
                    for all n_p ∈ (pred[n] ∩ C) do
                        E_c = E_c\(n_p, n);
                        baNode[n_p] = true;
                        lhNode[n] = true;
                    end for
                end if
            end for

            for all n ∈ C do                 ▷ projection of (V,E) on C nodes
                for all n_s ∈ (succ[n] ∩ G\C) do
                    E_c = E_c\(n, n_s);
                end for
                for all n_p ∈ (pre[n] ∩ G\C) do
                    E_c = E_c\(n_p, n);
                end for
                if ∃n′ ∈ C: ((pre(n′) ∪ succ(n′)) ∩ C = ∅) then
                    V_c = V_c\{n′};
                end if
            end for
            if V_c ≠ ∅ then
                for all v_c ∈ V_c do
                    level[v_c] = level[v_c] + 1;
                end for
                SCCWalk(V_c, E_c);
            end if
        end if
    end if
end function
```

Algorithm 5 Hierarchical SCC-Coloring Algorithm

```
function Initialize(V, E, s)
    SetOfSCCs := ∅;
    for all v ∈ V do
        lhNode[v] := false;
        baNode[v] := false;
        visited[v] := false;
        level[v] := 0;
    end for
    i := 0;

    Expand(s);

    for j := 0 to i do
        SetOfSCCs := SetOfSCCs ∪{C_i};
    end for

    return SetOfSCCs;
end function

function Expand(v)
    visited[v] := true;
    if v ∈ pred[v] then
        C_i := {v};
        i := i + 1;
        level[v] := level[v] + 1;
    end if

    if ∃v_p ∈ pred[v]: visited[v_p] = false then
        lhNode[v] = true;

        for all v_p ∈ (pred[v]\{v}): visited[v_p] = false do
            baNode[v_p] := true;
            C_i := ∅;
            i := i + 1;
            Backpropagate(v_p, v, i);
        end for
    end if

    if ∃v′ ∈ V.∀v_p ∈ pred[v′]: visited[v_p] = true ∧ visited[v′] = false then
        Expand(v’);
    end if
end function

function Backpropagate(n, v, i)
    C_i := C_i ∪ {n};
    level[n] = level[n] + 1;

    if n = v then
        return;
    else
        for all n_p ∈ (pred[n]\{n}) do
            Backpropagate(n_p, v, i);
        end for
    end if
end function
```

3.3 Finding Dependencies

Finding dependencies is as much dependent on the type of dependency to be found, as on the data types utilized in the program. This is easily seen, when thinking of data types like boolean or integer. In contrast to boolean values which are either true or false, integer types pretty much allow to choose from an infinite amount of values. While the dependency analysis for integer variables is not affected by this, since only the usage of variable is of interest, finding dependencies among indexed variables gets way harder. In fact two index expression for the same variable can be arbitrarily complex, especially when many integer variables are used throughout these expression. Finding a dependency for such an variable essentially requires to solve an equation system, such that all utilized variables carry values such that both index expressions evaluate to the same value. As can be imagined of such approaches, they often prove to be very costly, making it more intelligent to resort to easier methods, checking if a possible dependency can be disproved rather than showing that a dependency exists. For that purpose the non-linear solver will be introduced in a short while, checking for behavioural dependencies between two given index expression of the same variable. But since the non-linear solver is only concerned with index expressions, the next two sections will provide methods for finding syntactic dependencies as well as behavioural dependencies with regard to all variables.

3.3.1 Data-flow Equations

The methods for finding syntactic dependencies usually employ some kind of data-flow equations. Data-flow equations are often used in sequential programs for analysing consecutive statements for certain data-flow relations like available expressions, reachability of definitions and so on[**?**, defaultwork] The trick is to assign each node of the CFG, labelled by a statement, a transfer function, which takes some input and computes some output used by either preceding or succeeding statement nodes. By means of several theoretical constructs, it can be shown, that constructing those transfer function a certain way and iterating them a number of times will eventually lead to a fixpoint. That is, no changes regarding input or ouput will occur any more. The outcome of

such an analysis might prove for example, that a preceding variable definition might never reach a certain statement. Being the case for the reachability analysis, this is due to the fact, that from two definitions executed on the same control path one definition will overwrite the other one. However in case of an EFSM, that is not true. Since all variable assignments are predicated, one must assume for every variable both to be written and not be written. In consequence every read of a variable depends in theory on every write, which might be executed before the current read is reached. The very same can be said for writes, which in turn can depend on every preceding read. As this results in arbitrary many dependencies, the proposed data-flow equations are only concerned in finding direct dependencies. Since the transitive closure of all direct dependencies reveals all possible dependencies, this is no drawback. The transfer functions are defined the following way:
$\forall v \in (V^{in} \cup V^{out}).\forall s \in S)$:

$$in_s(v) := \bigcup_{s_p \in pre[s]} out_{s_p}(v)$$

$$out_s(v) := \begin{cases} in_s(v) & \neg used_s(v) \\ \{s\} & otherwise \end{cases}$$

$\forall v \in V^{loc}.\forall s \in S$:

$$in^r_s(v) := \bigcup_{s_p \in pre[s]} out^r_{s_p}(v)$$

$$in^w_s(v) := \bigcup_{s_p \in pre[s]} out^w_{s_p}(v)$$

$$out^r_s(v) := \begin{cases} in^r_s(v) & \neg used_s(v) \\ in^r_s(v) \cup \{s\} & rd_s(v) \wedge \neg wr_s(v) \\ \emptyset & \neg rd_s(v) \wedge wr_s(v) \\ \{s\} & rd_s(v) \wedge wr_s(v) \end{cases}$$

$$out^w_s(v) := \begin{cases} in^w_s(v) & \neg wr_s(v) \\ \{s\} & wr_s(v) \end{cases}$$

<u>with</u>:
$rd_s(v) := (v \in rdVar(\mathcal{L}(s)) = true)$
$wr_s(v) := (v \in wrVar(\mathcal{L}(s)) = true)$

$used_s(v) := rd_s(v) \vee wr_s(v)$

This gives rise to following dependencies:
$\forall v \in (V^{in} \cup V^{out}).\forall s \in S.\forall s' \in in_s(v)$: $s' \leq_v^s s$ and
$\forall v \in V^{loc}.\forall s \in S$:

$$(rd_s(v) \wedge \neg wr_s(v) \rightarrow (\forall s' \in in_s^w(v) : s' \leq_v^s s)) \wedge$$

$$(wr_s(v) \rightarrow (\forall s' \in in_s^r(v) : s' \leq_v^s s))$$

While input and output variables are easy to handle, as they are only read or written, it gets difficult for local variables. Since finding read after write (wr $\rightarrow$ rd) dependencies is not enough for determining all dependencies, all reads before a write have to be amassed. This explains the twofold in-/output transfer functions, which have to capture either reads or writes. It is important to note, that the data-flow equations introduced do not account for additional dependencies originating from either synchronous or clocked synchronous execution model. The dependencies caused by modelling reaction to absence or state dependent inputs are simply added to those dependencies already found by the analysis. That way the data-flow equations can be kept more precise and are better to comprehend, than otherwise.

3.3.2 Building dependency equations

With syntactic dependencies handled, it is time to look at the behavioural dependencies. Finding behavioural dependencies is much harder than syntactic ones, as there is no way simple way to abstract from behaviours other than to check for given inputs if a behavioural dependency exists or not. Basically this says that one either has to have a behaviour to check for a given dependency or one has to check for a possible behaviour of the program that shows such a dependency. While this is very costly, as mentioned in the beginning in the section, it is non the less done. Often this requires the use of so called model-checkers, which take logical formulas of certain classes as input and decide whether they are satisfiable or not. Using these model-checkers requires to formulate its questions in terms of a formula that can be checked for satisfiability. With regard to dependencies this means, that if some particular

syntactic dependency is given, one is highly interested to know if this dependency also carries over to some actual behaviour or not. In order to answer this question, for each path between the two states of the structural dependency, a formula must be constructed mimicking not only the behaviour of this path but all its possible evaluations too. This formula can then be given to model checker, which tries to find some behaviour for this path such that the dependency is fulfilled. A formula for any given EFSM can be constructed the following way: Given path $\pi := s_h \dots s_j$ in EFSM M, define $\mathcal{L}_i(s_i) := \mathcal{L}(s_i)$ and $\forall_{h \leq i \leq j} s_i . \forall g \in \mathcal{L}_i(s_i)$:

- For g $:= (\gamma \Rightarrow x := \sigma)$, replace each variable occurrences v in γ, σ by $[v]^v_{v_i}$ and x by $[x]^x_{x_{i+1}}$

and $\forall_{(s_i, c_i, s_{i+1})} c_i$:

- Replace each variable occurrence v by $[v]^v_{v_i]}$

Further encode reaction to absence explicitly into the states, by adding to each state a certain set of guarded actions. This is done by redefining $\mathcal{L}_i(s_i)$ as:

$$\mathcal{L}_i(s_i) := \mathcal{L}(s_i) \cup \bigcup_{x \in wrVar(s_i)|^{evt}} \{(\bigwedge_{g \in g_x} \neg g) \Rightarrow x_i := false\} \cup \bigcup_{x \in wrVar(s_i)|^{mem}} (\{(\bigwedge_{g \in g_x} \neg g) \Rightarrow x_i := x_{t_i}\} \cup \{true \Rightarrow x_{t_{i+1}} := x_i\})$$

with $g_x := \{g \in \mathcal{L}(s) | x \in wrVar(g)\}$

Those redefinitions do not only encode reaction to absence but also serve the purpose of making each state along the path π unique. This goes so far, that the path is allowed to contain cycles, since the set of guarded actions appended to each state, crucially depends on the exact position of this state along path π. Given these transformations, all necessary building blocks for defining a dependency system for path *pi* have been introduced. Hence the dependency system for a single state s_i is formalized the following way[8]:

$$ds_i := \bigwedge_{g \in \mathcal{L}_i(s_i)} grd(g) \Rightarrow action(g),$$

which in turn allows to define the dependency system for a path π as

$$ds_\pi := ds_{s_h \dots s_k} := (\bigwedge_{h \leq i \leq k} ds_i) \wedge (\bigwedge_{h \leq i \leq k} c_i)$$

A dependency system is called satisfiable, iff there exists a model T, which assigns each variable occuring in the dependency system a type-consistent value, such that the dependency system is satisfied. Given two states s_h, s_k, there exists a dependency from s_h to s_k, iff there exists a path $\pi := s_h \dots s_k$, such that the corresponding path dependency system ds_π is satisfiable. Any syntactical dependency between two states $s_i \leq_v^s s_j$ over variable v is also an behavioural dependency, iff s_i is dependent on s_j by a path π and the dependency system assigns values to both v_i as well v_j. For syntactic dependencies of the form $s_i \leq_v^s [k]s_j$, with array access $v[id_i] \in usedarrayVar(s_i), v[id_j] \in usedarrayVar(s_j)$, the dependency system must assign both values to $v_i[id_i]$ and $v_j[id_j]$ as well as check if expressions id_i and id_j are evaluated to k. Otherwise the syntactic dependency does not translate into a behavioural one. Especially for EFSM's relying mostly on boolean variables, checking such dependency systems for satisfiability is a feasible option for determining the behavioural dependency relation.

3.3.3 Non-linear Solver

Although model checking can be utilized for checking behavioural dependencies on indexed variables, there are other methods[26], which try to exploit certain mathematical properties of the corresponding index expressions. The non-linear solver is one of them. Its basic idea is to execute a synchronous program symbolically and to evaluate index expressions based on the variable environments created during this execution. For this purpose the solver tries to identify how much any execution of the program might increase or decrease an integer variable. This allows to maximize or minimize index expressions by simply inserting the maximum or minimum bounds a variable might have at this point of the program execution. The approach for symbolically executing the program as well as finding corresponding bounds for variables is depicted in algorithm 12. It starts with transforming the underlying EFSM by assigning all delayed assignments of one state to all its succeeding states. This is done since the solver is only concerned with each state individually but does not want to keep track of how this state was reached in the first place. After that the extended tarjan algorithm is applied. It finds all maximal SCC's of the CFG as tarjan's original algorithm would do including decompositions of

those maximal SCC into its nested components. This algorithm is an essential element of the solver, since the solver will handle the execution of the program not a state basis but on the level of SCC's. It executes each maximal SCC the number of times given by a so called boundary analysis and then proceeds with the next SCC. The catch about this method is, that the graph of maximal SCC's is acyclic, this means there actually exists a nice order, such that each SCC can be run one after another. While this again is based on some nice graph theoretic concepts, it allows to be more concerned with SCC's then arbitrary program paths. Before the execution begins evaluating each SCC step by step, it tries to determine for each state of the program and for all its variables, the growth a variable might see in this state. Then when the symbolic execution of the program starts, the idea is to combine all effects of all states for one SCC, to see how much a variable might decrease or increase in value during one run through the SCC. After the upper and lower bounds for the effects of this variable during one run have been determined, these computations are repeated as often as estimated by the boundary analysis. This iteration number is found by checking the conditions of those state transitions that might lead from a node inside the SCC to an entry node of the same SCC. Either by inserting upper,lower bounds of the variables determined so far or by running a so called interval analysis. This interval analysis much like the symbolic execution itself runs over all states of the program and tries to extract from all predicates the boundaries of the variable value. Though this may sound strange, it yields startling good results for a heuristic and is hence employed if any estimation is required. With the program symbolically executed, the actual independency analysis can then be done as follows:
For given array accesses a$[id_i]$ in state s_i and a$[id_j]$ in state s_j, no dependency exists iff

$$[id_i]_{min_b^{s_i}} R_1 [id_j]_{max_b^{s_j}} \wedge [id_i]_{min_a^{s_i}} R_1 [id_j]_{max_a^{s_j}} \vee$$

$$[id_i]_{max_b^{s_i}} R_1 [id_j]_{min_b^{s_j}} \wedge [id_i]_{max_a^{s_i}} R_1 [id_j]_{min_a^{s_j}}$$

with $R_1, R_2 \in \{<, >\}$

and the following definitions:

$\forall s \in \{s_i, s_j\}$:

C := findMaxSCC(s_i, S);

min^s_b := lowerBound($Env^{out/lb}_C$);

min^s_a := lowerBound(Env^{out}_C);

max^s_b := upperBound($Env^{out/lb}_C$);

max^s_a := upperBound(Env^{out}_C);

Above definition of two accesses being independent relies on some simple observations. First of all the states of these accesses will very likely occur in some SCC and are therefore prone to be executed a certain number of times. In consequence it is probable that the index expression will evaluate to different values based on the number of iterations they have undergone. If those expressions now exhibit a so called monotonic behaviour, that is they either keep always increasing or always decreasing in value with the number of iterations, then two index expressions differing in value at the beginning of the first iteration will also differ after the last iteration if they exhibit opposing monotonic behaviours. But they can also differ, if the difference between both index evaluations is great enough, that the smaller index expression cannot catch up during all iterations to the starting value of the greater index expression[?]. Basically that is what above definitions says. For both states s_i and s_j the variable environments before a possible loop iteration $Env^{out/lb}$ and after the iterations Env^{out} are retrieved and upper,lower bounds determined. Evaluating the index expressions under these environments yields finally values, which might be checked for independency. This is also seen in example 3.5, where a modified bubblesort-algorithm is checked dependencies among two array accesses. If the relations however do not imply any independence, one has to fall back to different methods[26, 30] like other dependency algorithms or even model checking, if necessary.

function EXECUTESYSTEMSYMBOLICALLY(M := (s_0,S,T,$\mathcal{L}$) , Env^{in})
    Transformation(M);
    SCCWalk(CFG_M);
    for all $s \in S$ **do**
        tS_s := TopologicalSorting(s);
        ProcessingStateEnv(s, tS_s);
        ProcessingStateInterval(s);
    end for

    DoIntervalAnalysis(S);
    processSCC(S, Env^{in});
end function

Explanation:
Function 'ExecutesSystemSymbolically' is the main function of the non-linear solver, as it transform the underlying EFSM, starts the extended tarjan algorithm and evaluates each state for its effects on variables. After this is done the algorithm runs the interval analysis, for loop iteration estimations and starts executing the program symbolically.

Algorithm 6 Helper Function - Transformation

function TRANSFORMATION(M)
  for all $s \in S$ **do**
    $\mathcal{L}'(s) := (\emptyset, \mathcal{L}(s))$;
    visited[s] := false;
  end for

  Transform(s_0);
end function

function TRANSFORM(s)
  visited[s] := true;
  $D_s := \emptyset$;
  for all $g \in \mathcal{L}(s)$ **do**
    if delayed(g) **then**
      $D_s := D_s \cup \{g\}$;
    end if
  end for
  $\mathcal{L}'(s) := (\mathcal{L}'(s)_1, \mathcal{L}'(s)_2 \backslash D_s)$;
  for all $s' \in succ[s]$ **do** $\mathcal{L}'(s')$:=($\mathcal{L}'(s')_1 \cup D_s, \mathcal{L}'(s')_2$);
  end for
  pick $s_f \in$ S: visited[s_f] = false;
  Transform(s_f);
end function

Explanation:
Functions 'Transformation' and 'Transform' serve to assign each state the set of delayed guarded actions of its preceding states.

Algorithm 7 Helper Function - topological Sorting

```
function TopologicalSorting(s)
    for all v ∈ usedVar(action(L′(s)_2)) do
        pred[v] := ∅;
        succ[v] := ∅;
    end for
    for all g ∈ L′(s)_2 do
        V_req := rdVar(action(g));
        x := wrVar(g);
        pred[x] := V_req;
        for all v' ∈ V_req do
            succ[v'] := succ[v']∪{x};
        end for
        tS := emptyList;
        Sort(pred, succ);
        return tS;
    end for
end function

function Sort(pred,succ)
    for all v ∈ (pred[v]=∅) do
        append v to tS;
        for all v' ∈ succ[v] do
            pred[v'] := pred[v']\ {v};
        end for
    end for
    Sort(pred, succ);
end function
```

Explanation:
Since guarded actions of one state might depend on each other, a variable can only evaluated, if all its guard and action variables have been evaluated. For this purpose algorithms 'topologicalSorting' and 'Sort' order the set of variables occurring in one state by establishing an total order, such that executing guarded actions according to this order resolves all dependencies amongst these guarded actions.

Algorithm 8 Helper Function - process state environment individually

function PROCESSINGSTATEENV(s,tS)
  $Env_s := \emptyset$;
  gA := $\mathcal{L}'(s)_2$;
  dgA := $\mathcal{L}'(s)_1$;
  for all $v \in usedVar(gA \cup dgA)$ **do**
    $Env_s := Env_s \cup \{(v, v)\}$;
  end for
  for all $dg \in dgA$ **do**
    $Env_s := Env_s \cup \{(wrVar(dg), rhs(dg))\}$;
  end for

  for i := 1 to |tS| **do**
    x := tS_i;
    for all g ∈ gA: wrVar(g) = x **do**
      gA := gA\{g};
      Env_s := Env_s ∪ Substitute((x,rhs(g)), Env_s, FV(rhs(g)));
    end for
  end for

  $Env_{\{s\}} := Env_s$;
end function

function SUBSTITUTE((x, t), Env, V_t)
  if $v_t = \emptyset$ **then**
    return {(x, t)};
  end if
  pick v ∈ V_t;
  $V_t := V_t \backslash \{v\}$;
  return $\bigcup_{(v,u) \in Env}$ Substitute((x,$[t]^u_v$), Env, V_t);
end function

Explanation:
Functions 'ProcessingStateEnv' and 'Substitute' actually build up the variable environment of one state, by simply executing all guarded actions occurring inside this state. The idea is that all effects on variables are preserved until a whole SCC has been processed. Just then are functions like upper- and lowerBound are applied. Function 'Substitute' is often employed, since it allows to replace a variable by its value of the environment.

Algorithm 9 Helper Function - process state intervals individually

function ComputeFPInterval(min_s, max_s, usedVar)
  f_p := false;
  while $\neg f_p$ **do**
    for all $x \in usedVar$ **do**
      $min'_s(x) := [min(min_s(x))]_{min_s,max_s}$;
      $max'_s(x) := [max(max_s(x))]_{min_s,max_s}$;
      if $x \in min'_s(x)$ **then**
        $min'_s(x) := [min'_s(x)]^0_x$;
      end if
      if $x \in max'_s(x)$ **then**
        $max'_s(x) := [max'_s(x)]^0_x$;
      end if
    end for

    if $min_s = min'_s \vee max_s = max'_s$ **then**
      f_p := true;
    end if
    $min_s := min'_s$;
    $max_s := max'_s$;
  end while

  return(min_s, max_s);
end function

function ProcessingStateInterval(s)
  gA := $\mathcal{L}(s)_1 \cup \mathcal{L}(s)_2$;
  for all $x \in usedVar(gA)$ **do**
    $min_s(x) := 0$;
    $max_s(x) := \bot$;
  end for
  for all $g \in gA$ **do**
    grd_g := DNF(grd(g)); ▷ DNF = disjunctive normal form
    for all $x \in rdVar(grd_g)$ **do**
      if $\exists p(\vec{x})$: $(x < p(\vec{x})) \in grd_g \vee (x > p(\vec{x})) \in grd_g$ **then**
        $min_s(x) := min_s(x) \sqcap (p(\vec{x}) - 1)$;
        $max_s(x) := max_s(x) \sqcup (p(\vec{x}) - 1)$;
      end if
      if $\exists p(\vec{x})$: $(x \leq p(\vec{x})) \in grd_g \vee (x \geq p(\vec{x})) \in grd_g$ **then**
        $min_s(x) := min_s(x) \sqcap p(\vec{x})$;
        $max_s(x) := max_s(x) \sqcup p(\vec{x})$;
      end if

    end for
  end for
  (min_s, max_s) := ComputeFPInterval(min_s, max_s, usedVar(gA));

Algorithm 10 DoIntervalAnalysis

function DoIntervalAnalysis(C)
    $min_C := \bot$;
    $max_C := \bot$;
    for all $s \in C$ **do**
        if $\{s\} \notin SCC_C$ **then**
            $min_C := min_C \sqcap min_s$; ▷ operator ⊓ denotes Min() for integer variables
            $max_C := max_C \sqcup max_s$; ▷ operator ⊔ denotes Max() for integer variables
        end if
    end for
    for all $C' \in SCC_C$ **do**
        DoIntervalAnalysis(C');
        $min_C := min_C \sqcap min_{C'}$;
        $max_C := max_C \sqcup max_{C'}$;
    end for

    $(min_C, max_C) :=$ ComputeFPInterval(min_C, max_C, $\bigcup_{s \in C} UsedVar(s)$);

    for all $s \in C$ **do**
        $min_s := min_C$;
        $max_s := max_C$;
    end for
end function

Explanation:
Function 'DoIntervalAnalysis' serves as main function for the interval analysis. It recursively checks for each encountered SCC the interval values of all variables and combines the intervals by utilizing upper and smaller bound operators. The intervals of an SCC are build by combining the intervals of its nested components and the intervals of the nodes belonging only to this SCC.

Algorithm 11 Fucntion 'ProcessSCC'

function PROCESSSCC(C, Env_0)
  WorkList := ∅;
  Visited := ∅;
  (pre,succ) := BuildSCCGraph(C, E) ▷ E = set of edges of CFG_M
  for all $C' \in SCC_C$ **do**
    if pre[C'] = ∅ **then**
      WorkList := WorkList ∪ {C'};
      $Env_{C'}^{in} := Env_0$;
    end if
  end for

  while WorkList ≠ ∅ **do**
    pick w ∈ WorkList;
    WorkList := WorkList \{w};
    Visited := Visited∪{w};
    $Env_w^{in} := Env_w^{in} \cup \bigcup_{c' \in pre[C]} Env_{C'}^{out}$;
    Env_w^{out} := IterateFixpoint(w, Env_w^{out});

    for all $C \in SCC_{C_{graph}}$ **do** ▷ missing: Define $SCC_{C_{graph}}$
      if pred[C] ⊆ Visited **then**
        WorkList := WorkList ∪ {C};
      end if
    end for
  end while
end function

Explanation:
Function 'processSCC' builds up the graph of maximal SCC's and starts traversing each SCC to compute its effects symbolically. Yet it does not compute the effects of a SCC itself but relies on function 'IterateFixpoint' for this task. Hence it mainly serves as framework for the actual execution of the SCC's.

function ITERATEFIXPOINT(C_s, Env^{in})
  $SCC^{G}_{C_s}$ = GraphSCC(C_s);
  for all $C \in SCC^{G}_{C_S}$ **do**
    $Env^{in}_{C} := \emptyset$;
    $Env^{out}_{C} := \emptyset$;
    if $(pre[C] \cap G \backslash C_s) \neq \emptyset$ **then**
      $Env^{in}_{C} := Env^{in}$;
    end if
  end for
  for i := 1 to maxIteration **do**
    for all $C \in SCC^{G}_{C_s}$ **do**
      $Env^{in}_{C} := \bigcup_{C' \in pre[C]} Env^{out}_{C'}$;
    end for
    for all $C \in (SCC^{G}_{C_s} \backslash SCC_{C_s})$ **do**
      for all $(x,t) \in Env_C$ **do**
        $Env^{out}_{C} := Env^{out}_{C} \cup$ Substitute((x, t), Env^{in}, FV(t));
      end for
    end for
    for all $C \in SCC_{C_s}$ **do**
      Env^{out}_{C} := IterateFixpoint(C, Env^{in});
    end for
    for all $C \in SCC^{G}_{C_s}$ **do**
      $Env^{out/lh}_{C} := Env^{out}_{C}$;
    end for
  end for
  basicIV := max($\bigcup_{C \in SCC_{C_{graph}}} countIV(Env^{out}_{C})$;
  loopCond := false;
  for all $s' \in C_s$ **do**
    if $\exists n \in (pre[s'] \cap G \backslash C_s)$ **then**
      loopCond := loopCond $\vee \bigvee_{s_p \in (pre[s'] \cap C_s)} getLoopCondition(s_p, s')$;
    end if
  end for
  (*) $\Leftarrow$ here might be inserted *ADD 1*
  loopCount := $([BA(loopCond)]_{min_s, max_s})^{basicIV}$;
  for j := 1 to loopCount **do**
    for all $C \in SCC^{G}_{C_s}$ **do**
      for all $(x,t) \in Env^{out}_{C}$ **do**
        $nEnv^{out}_{C}$:= Substitute((x, t), $Env^{out}_{C}, FV(t)$);
      end for
      $Env^{out}_{C} := nEnv^{out}_{C}$;
    end for
  end for
  for all $C \in SCC^{G}_{C_s}$ **do**
    if $(succ[C] \cap G \backslash C_s) \neq \emptyset$ **then**
      $Env^{out}_{C_s} := Env^{out}_{C_s} \cup Env^{out}_{C}$;
    end if
  end for
  $Env^{out}_{C_s}$:= lowerBound($Env^{out}_{C_s}$) $\cup$ upperBound($Env^{out}_{C_s}$);

  return $Env^{out}_{C_s}$;
end function

Explanation:
Function 'IterateFixpoint' executes a SCC and all its nesting components symbolically. For this task it relies on varies helper functions, providing bounds, loop estimations or variable substitutions. A SCC is thereby executed much like a data-flow equation, which leads to states being executed quite a number of time. This is due to the particular assumption, that SCC's do not have to confirm to certain structural constraints. To guarantee that the effects of one state have reached all other states of the SCC, it is mandatory to run the computation of an SCC until a fixpoint has been found. After this fixpoint is established, loop estimations must be retrieved to decide how often the SCC is meant to be run. When this and the loop iterations themselves are done, upper- and lower bounds on the variable environments are determined and assigned to the corresponding states.

function UPPERBOUNDS(Env, V)
  for all $v \in V$ **do**
    upperbound := ⊥;
    $maxdegree_{int}$:= 0;
    $maxdegree_{sym}$:= 0;
    for all $(v, t) \in Env$ **do**
      if $\exists \alpha_0, \ldots, \alpha_m \in SymTerm.\exists c_0, \ldots, c_n \in \mathbb{Z}$:
t := $\alpha_0 + \alpha_1 k^1 + \ldots + \alpha_m k^m + c_0 + c_1 k^1 + \ldots + c_n k^n$ with $n, m \in N$ **then**
        if $maxdegree_{int} < n$ **then**
          $maxdegree_{int}$:= n;
        end if
        if $maxdegree_{sym} < m$ **then**
          $maxdegree_{int}$:= m;
        end if
      end if
    end for
    $symbound_v$:= 0;
    $intbound_v$:= 0;
    for i := 0 to $maxdegree_{int}$ **do**
      max := 0;
      for all $(v, t) \in Env$ **do**
        if max $< t|^{c_i}$ **then**
          max := c_i;
        end if
      end for
      $intbound_v := intbound_v + max * k^i$;
    end for

    for i := 0 to $maxdegree_{sym}$ **do**
      sum := 0;
      for all $(v, t) \in Env$ **do**
        sum := sum + abs($t|^{\alpha_i}$);
      end for
      $symbound_v := symbound_v + sum * k^i$;
    end for

    upperbound(v) :=$symbound_v$ +$intbound_v$;
  end for

  return upperbound;
end function

Explanation:
Function *upperBound* serves to compute the upper bound for some environment Env over a variable set V, by adding up all the minimal effect a variable might have due to the environment.

```
function lowerBounds(Env, V)
    for all v ∈ V do
        lowerbound := ⊥;
        maxdegree_int := 0;
        maxdegree_sym := 0;
        for all (v,t) ∈ Env do
            if ∃α_0,...,α_m ∈ SymTerm.∃c_0,...,c_n ∈ ℤ:
t := α_0 + α_1k^1 + ... + α_mk^m + c_0 + c_1k^1 + ... + c_nk^n with n,m ∈ N then
                if maxdegree_int < n then
                    maxdegree_int := n;
                end if
                if maxdegree_sym < m then
                    maxdegree_int := m;
                end if
            end if
        end for
        symbound_v := 0;
        intbound_v := 0;
        for i := 0 to maxdegree_int do
            min := 0;
            for all (v,t) ∈ Env do
                if min > t|^{c_i} then
                    min := c_i;
                end if
            end for
            intbound_v := intbound_v + min * k^i;
        end for

        for i := 0 to maxdegree_sym do
            sum := 0;
            for all (v,t) ∈ Env do
                sum := sum + (-abs(t|^{α_i}));
            end for
            symbound_v := symbound_v + sum * k^i;
        end for

        lowerbound(v) :=symbound_v +intbound_v;
    end for
```



```
    return lowerbound;
end function
```

Explanation:
Function *lowerBound* serves to compute the lower bound for some environment Env over a variable set V, by adding up all the minimal effect a variable might have due to the environment.

Algorithm 12 Helper Functions - Extracting different SCC's

function BUILDSCCGRAPH(V, E)
  for all $C_1, C_2 \in SCC_V$ **do**
    if $\exists s_1 \in C_1 . \exists s_2 \in C2$: $(s_1, s_2) \in E$ **then**
      $succ[C_1] := succ[C_1] \cup \{C_2\}$;
      $pre[C_2] := pre[C_2] \cup \{C_1\}$;
    end if
  end for
  for all $C \in SCC_v . \forall s \in (V \setminus \bigcup_{C' \in SCC_v} C')$ **do**
    if $\exists s_c \in C$: $(s_c, s) \in E$ **then**
      $succ[C] := succ[C] \cup \{\{s\}\}$;
      $pre[\{s\}] := pre[\{s\}] \cup \{C\}$;
    end if
    if $\exists n_c \in C$: $(n, n_c) \in E$ **then**
      $succ[\{s\}] := succ[\{s\}] \cup \{C\}$;
      $pre[C] := pre[C] \cup \{\{s\}\}$;
    end if
  end for
  return (pre,succ);
end function

function GRAPHSCC(C)
  $C_t := emptyset$;
  for all $s \in ((\bigcup_{C' \in SCC_C} C') \cap C)$ **do**
    $C_t := C_t \cup \{s\}$;
  end for
  return $(C_t \cup SCC_C)$;
end function

function FINDMAXSCC(s,C)
  $C_s := \{\text{s}\}$;
  if $\exists C' \in SCC_C$ **then**
    $C_s :=$ C';
  end if
  return C_s;
end function

Explanation:
While Function *BuildSCCGraph* actually builds the acyclic graph of maximal SCC's, function 'GraphSCC' serves to include all nodes into the decomposition of an SCC, which do not form an SCC themselves. Such nodes are omitted from the decomposition of a set of nodes, as they are not included in any nested SCC. Function 'findMaxSCC' on the other side, does exactly what its name suggests and returns the maximal SCC, including the given state s.

Extensions to the non-linear solver

AD 1:

The following algorithm can be inserted as extension to non-linear solver in function 'IterateFixpoint' at position *. Instead of iterating all the variable effects of the SCC, this extension only iterates the upper and lower variable bounds. While this might serve to decrease the overall runtime, it certainly allows for much simpler variable environments.

AD 2:

Finding the largest simple path of an SCC is NP-hard, therefore the following heuristic is used: $|C_s|$ - 1 $\leq l_{max}$
with l_{max} denoting the largest simple path inside the SCC_{C_s}

AD 3:

There might the cases were loop conditions simply evaluate to boolean variables, like 'while($\not$fixpoint)' and so on. Since boundary analysis fails for such variables, the predicates of this boolean variables are checked for any reasonable loop estimates. The following extension resolves this issue: $b_c :=$ getLoopCondition(s, n); if($b_c \in Var(C_s)$)

loopCond := loopCond $\vee$ upperBound($Env^{out}_{C_s}$)(b_c)

AD 4:

Performance issues:

All SCC's found by first run of Tarjan algorithm, can be run independently on symbolic environments(all algorithms allow symbolic evaluation). Start with lowest SCC's and use IntervalAnalysis to estimate number of loop iterations. The outer SCC are computed by reusing the results of all nested SCC's and iterating the corresponding effects loop iteration-times. That way, the function 'processSCC' must only replace symbolic values with the ones provided by the environment instead of computing all states again.

AD 5:

Instead of pursuing every path in CFG of EFSM and allowing every guarded action to propagate its effects, define priorities, which allow to rank path conditions (transition conditions) or guarded actions (writing the same variable). The likelihood of a given guarded action gA and a path condition pc are determined using a so called 'rank' function. It shall be noted, that the rank

Algorithm 13 additional upper, lower bounds

for all $C \in SCC$ **do**
$Env^{out}_{C_{min}}, lowEnv^{out}_{C}$:= lowerBounds(Env^{out}_{C});
$Env^{out}_{C_{max}}, upEnv^{out}_{C}$:= upperBounds(Env^{out}_{C});
end for
for all $v \in usedVar(C_s)$ **do**
$sort^{min}_{C_s} := sort^{min}_{C_s} \cup \{true \Rightarrow v := lowEnv^{out}_{C_s}(v)\}$;
$sort^{max}_{C_s} := sort^{max}_{C_s} \cup \{true \Rightarrow v := upEnv^{out}_{C_s}(v)\}$;
end for
$tS^{min}_{C_s}$:= topologicalSorting($sort^{min}_{C_s}$);
$tS^{max}_{C_s}$:= topologicalSorting($sort^{max}_{C_s}$);
for **do**
for all j := 1 to loopCount **do**
for i_{min} := 1 to $|tS^{min}_{C_s}|$ **do**
$var_{i_{min}} := tS^{min}_{C_s}|^{i_{min}}$;
(v, t) := Substitute($(var_{i_{min}}, Env^{out}_{C_{min}}(var_{i_m in}))$, $lowEnv^{out}_{C}$, $FV(Env^{out}_{C_{min}}(var_{i_{min}})$);
$Env^{out}_{C_{min}}(var_{i_{min}})$:= t;
end for

for i_{max} := 1 to $|tS^{max}_{C_s}|$ **do**
$var_{i_{max}} := tS^{max}_{C_s}|^{i_{max}}$;
(v, t) := Substitute($(var_{i_{max}}, Env^{out}_{C_{max}}(var_{i_m ax}))$, $upEnv^{out}_{C}$, $FV(Env^{out}_{C_{max}}(var_{i_{max}})$);
$Env^{out}_{C_{max}}(var_{i_{max}})$:= t;
end for
end for
end for
for all $C \in SCC^{G}_{C_s}$ **do**
$Env^{out}_{C} := Env^{out}_{C_{min}} \cup Env^{out}_{C_{max}}$;
end for

function assumes boolean formulas to contain no predicates dependent on integer variables. As the distribution of integer variables is hard to handle, such formulas are not handled. Assume such a boolean formula $b : V \to \mathbb{B}$ is given (with no unsatisfiable clauses, than there exists an equivalent DNF of b, called b' with
b' := $\bigvee_i \bigwedge_j \alpha_{ij}, \alpha_{ij} \in (V \cup \{true, false\})$
Rank(b') is then defined as,
Rank(b') := $\sum_i \prod_j P(\alpha_{ij})$
with $\forall v \in V : P(v) := 1/2, P(false) := 0, P(true) := 1$
Given now guarded action gA := $g \Rightarrow x := f$ with $g : V \to \mathbb{B}, f : V \to \mathbb{B}/\mathbb{Z}$, replacing all predicates in g, f yields g', f', which now allows to define the rank of guarded action as
rank(gA) := $\begin{cases} rank(g) * rank(f) & type(x) \in \mathbb{B} \\ rank(g)^2 & otherwise \end{cases}$ and given some path condition pc, replacing the predicates by true yielding pc' gives rank(pc'). Using this rank-function, the non-linear solver for example can try to determine paths or guarded actions that are more likely to be executed than others, helping in managing the complexity of the dependency analysis. Note that this heuristic determines the likelihood of an execution solely on the unprejudiced chance to satisfy a boolean formula. This is equivalent to assuming all boolean variables to be uniformly distributed. But of course this must not hold, as some variable values might be more likely as others or dependent on different variables. That is the reason why this method is really just an heuristic method and nothing more.

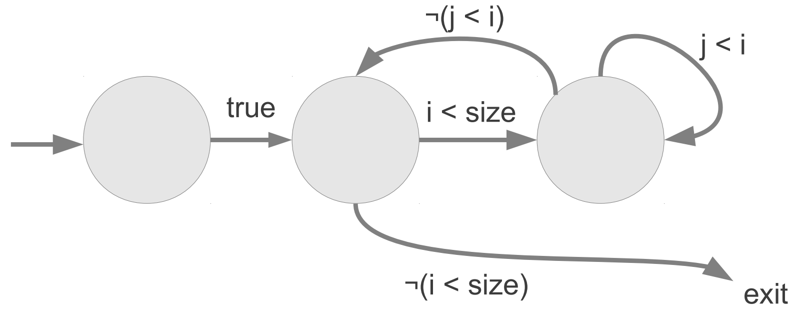

Figure 3.4: Control-flow graph of EFSM M

EFSM M := $(st, \{st, l_1, l_2\}, T, \mathcal{L})$ with

$$\mathcal{L}(st) := true \Rightarrow next(i) := 0$$

$$\mathcal{L}(l_1) := \begin{array}{l} true \Rightarrow next(i) := \text{i+1} \\ true \Rightarrow next(j) := 0 \end{array}$$

$$\mathcal{L}(l_2) := \begin{array}{l} true \Rightarrow next(j) := \text{j+1} \\ b[j] > b[j+1] \Rightarrow next(b[j]) := \text{b[j+1]} \\ b[j] > b[j+1] \Rightarrow next(b[j+1]) := \text{b[j]} \\ b[j] > b[j+1] \Rightarrow next(a[size^2 + i]) := \text{a[j]} \end{array}$$

and conditional transition T depicted in the picture above 3.4.

Application of extended tarjan algorithm reveals the following SCC's:

$$SCC_{\{st,l_1,l_2\}} := \{\{l_1, l_2\}\}$$

$$SCC_{\{l_1,l_2\}} := \{\{l_2\}\}$$

Application of interval analysis yields:

i ∈ [0, size]

j ∈ [0, i] ⇒ [0, max[i] := size]

Application of boundary analysis for edges (l_2, l_1) and (l_2, l_2) yields:

BA[¬(j < i)] := BA[j < i] := BA[max[i] - min[j]] = size

BA[j < i] := BA[max[i] - min[j]] = size

Processing of SCC $\{l_1, l_2\}$ with loop estimates yields:

before the loop:

i $\in [0, 0]_b$

j $\in [0, 0]_b$

after the loop:

i $\in [0, (i+1)^s ize]_a := [0, size]_a$

j $\in [0, (j+1)^s ize]_a := [0, size]_a$

Finally dependency analysis for array accesses a$[size^2 + i]$, a[j] in state l_2 reveals:

$$size^2 + min[i]_b := size^2 + 0 > 0 =: max[j]_b \wedge$$

$$size^2 + min[i]_a := size^2 + 0 > \text{size} =: max[j]_a$$

3.4 Decomposing EFSM

While previous methods and algorithms have been mostly concerned with actual dependency analysis, the following criterion allows to decompose a single extended finite state machines, into two EFSM's. Hence it is not about dependency but more about independency, as independent variable partitions are searched.They in turn allow each state to be separated such that two EFSM can run in parallel, executing thereby each one part of the state. The nice thing about this criterion is that it can also be applied to parts of an EFSM. This allows to run some states in parallel, while others are simply executed sequentially.

Now the criterion states that given EFSM M := (s_0, S, T, $\mathcal{L}$), a decomposition into $M := M_1||M_2$ is possible iff
$\exists V_1, V_2 := V^{loc}_{1/2} \cup V^{in}_{1/2} \cup V^{out}_{1/2} \subseteq V$:

$$(V^{loc}_{1/2} \cap V^{in}_{1/2} \cap V^{out}_{1/2} = \emptyset)\wedge$$

$$(V^{in}_1 \cap V^{in}_2 = \emptyset) \wedge (V^{out}_1 \cap V^{out}_2 = \emptyset)\wedge$$

$$(\forall v(\in V_1 \cap V_2)\colon\ (v \in V^{out}_1 \Rightarrow v \in V^{in}_2) \wedge (v \in V^{out}_2 \Rightarrow v \in V^{in}_1))\wedge$$

$$\forall s \in S : (\mathcal{L}(s) := \mathcal{L}(s)|^{V_1} \cup \mathcal{L}(s)^{V_2}) \wedge (\mathcal{L}(s)|^{V_1} \cap \mathcal{L}(s)^{V_2} = \emptyset)$$

with $\mathcal{L}(s)|^V := \{gA_1, \ldots, gA_n\} := \bigcup_{i \leq n} gA_i|^V$ and

$$gA|^V := \begin{cases} gA & \text{for V} := V^{loc} \cup V^{in} \cup V^{out} \\ & V^{loc}(gA) \subseteq V^{loc}\wedge \\ & V^{in}(gA) \subseteq V^{in}\wedge \\ & V^{out}(gA) \subseteq V^{out} \\ \emptyset & otherwise \end{cases} \qquad \text{Idea:}$$

Construction of the composition of two EFSM's M_1, M_2 with
$M_{1/2} := (s_{01/02}, S_{1/2}, T_{1/2}, \mathcal{L}_{1/2})$ is defined as
$M_{1||2} := ((s_{01}, s_{02}), S_1 \times S_2, T', \mathcal{L}')$ with

$$\text{t} := ((s_1, s_2), (c_1 \wedge c_2), (s'_1, s'_2)) \in T \text{ iff } (s_1, c_1, s'_1) \in T_1 \wedge (s_2, c_2, s'_2) \in T_2$$

$$\mathcal{L}'((s_1, s_2)) := \mathcal{L}_1(s_1) \cup \mathcal{L}_2(s_2)$$

This construction is allowed as long as $V^{in}(S_1) \cap V^{in}(S_2) = \emptyset$ and $V^{out}(S_1) \cap V^{out}(S_2) = \emptyset$. Since the EFSM's are derived from synchronous systems, which only allow one input/output channel per variable, this fact is trivially true. Now the basis of each EFSM are the guarded actions it executes in each state and the variables sets used in those states. For each EFSM, there is the maximal set V_{max}, which is compromised of all variables used over all states in the EFSM. Furthermore, variables can always be classified into either input, output or local variables. Hence the set V_{max} can be decomposed into $V_{max} := V^{in}_{max} \cup V^{out}_{max} \cup V^{local}_{max}$. Now two variable sets V_1, V_2 have to exists, which possess such a decomposition and only share variables in an input/output relationship. Furthermore for each state of the composed EFSM it must hold, that the set of guarded actions of this state can be separated into two disjunct sets, with one set fulfilling the variable classifications of V_1 and the other one matching the classifications of V_2. If those two sets V_1, V_2 actually exist, then a decomposition is naturally possible. Basically one can think of two EFSM's running along side each other, each having a clear defined interface, classifying variables by either V_1 or V_2 and building a transition relation by synchronizing individual transitions.

Given now tow variable sets V_1 and V_2 and an EFSM M, which fulfills above preposition, then two EFSM's M_1, M_2 with $M_1 || M_2 := M$ can be reconstructed the following way:
$M_{1/2} := (s_{01/02}, S_{1/2}, T_{1/2}, \mathcal{L}_{1/2})$ with

$$s_{01/02} := s_M$$

$$S_{1/2} := S_M$$

$$\forall (s_M, c_M, s'_M) \in T_M \colon\ (s_M, c_M|^{V_{1/2}}, s'_M) \in T_{1/2}$$

$$\forall s \in S_M \colon\ \mathcal{L}_{1/2}(s) := \mathcal{L}_M(s)|^{V_{1/2}}$$

Given EFSM M $:= (s_1, \{s_1, s_2\}, \{(s_1, s_2)\}, \mathcal{L})$ with:

$$\mathcal{L}(s_1) := \begin{array}{l} x_1 \wedge y_1 \Rightarrow z_1 := \text{true} \\ x_2 \wedge z_2 \Rightarrow o_1 := \text{true} \end{array}$$

$$\mathcal{L}(s_2) := \begin{array}{l} \neg x_1 \wedge y_1 \Rightarrow o_2 := \text{true} \\ \neg z_1 \Rightarrow o_1 := \text{true} \end{array}$$

then variable sets $V_1 := \{x_1, y_1\}^{in} \cup \{z_1, o_2\}^{out}$ and $V_2 := \{x_2, z_1\}^{in} \cup \{o_1\}^{out}$ allow to decompose M into $M_1 || M_2$:

$$M_1 := (s_1, \{s_1, s_2\}, \{(s_1, s_2)\}, \mathcal{L}_{M_1}) \text{ with}$$

$$\mathcal{L}_{M_1}(s_1) := x_1 \wedge y_1 \Rightarrow z_1 := \text{true}$$

$$\mathcal{L}_{M_1}(s_2) := \neg x_1 \wedge y_1 \Rightarrow o_2 := \text{true}$$

$$M_2 := (s_1, \{s_1, s_2\}, \{(s_1, s_2)\}, \mathcal{L}_{M_2}) \text{ with}$$

$$\mathcal{L}_{M_2}(s_1) := x_2 \wedge z_2 \Rightarrow o_1 := \text{true}$$

$$\mathcal{L}_{M_2}(s_2) := \neg z_1 \Rightarrow o_1 := \text{true}$$

Figure 3.6: Example for decomposition of given EFSM

4 Implications of Asynchronous Environment

While the previous chapter tried to answer the question of finding dependencies for synchronous programs, the following sections are more concerned with the actual environment of a program and the implication this has to a synchronous program. The interesting thing is, that up to this point all synchronous programs were assumed to be run in a synchronous environment. Hence communication between different programs was carried out in synchrony, allowing the program to determine if a variable value was present or not. Now with regard to real world environments, this turns out to be a fairly optimistic assumption. Most environments suffer from sort of delay, that is values leaving the sender at one point in time are very unlikely to arrive at the sender synchronously. Though communications are assumed to be reliable, that is no values are lost, one can hardly predict at which point of the time communication values will arrive. For synchronous programs embedded into an asynchronous environments, an environment that suffers from such delays, this leads to certain problems. As receiving systems can usually argue that a variable value was actually not send by the sender, due to the synchronous environment, it is now unknown, if the value was just delayed and has not arrived yet or if there actually was no value at all. Does this imply that synchronous programs are unqualified for the use in asynchronous environments per se ? Fortunately, there are classes of systems that can be run in so called GALS (globally asynchronous - locally synchronous) architectures[9][1]. Those classes are given by a

[1]That is each of them locally computes its reactions synchronously while the communication between system is viewed as asynchronous. The advantage of this approach is, that some systems can actually run faster than others, that is they can start computing not based on some global signal, but on their own clock signal

set of conditions a system must fulfill in order to fall into this particular class. All classes make certain that the asynchronous behaviour of a system produces the same variable values than the synchronous one would. Yet the timing of those variable values, that is in which reactions step they are actually read or produced, differs from class to class. This allows for terms like endochrony, flow-insensitivity or weak endochrony[27, 21]. With the first two very sensitive to timing issues and relevant to only a small class of system, weak endochrony actually proves to be the most interesting in the interest of this thesis. Not only does weak endochrony encompass endochrony and flow-insensitivity, making it more general, it also allows to separate or merge independent computations based on the circumstances. Since both execution models largely introduce dependencies, separating independent computations has a higher priority than actual merging. The notion of independence for weak endochrony is hereby based on so called support sets[27], which are defined for some reaction r as $\mathit{support}(r) := \{v \in V | r(v) \notin \{*, \bot, \top\}\}$, the set of variables actually present in some reaction. This implies the independency of two reactions r_1, r_2 with $r_1 \bowtie r_2$ iff $\mathit{support}(r_1) \cap \mathit{support}(r_2) = \emptyset$, that is both reactions do not share any present variables. Translated to the concept of guarded actions, which behaviours make up the reactions of the program, guarded actions gA_1, gA_2

$$gA_1 := x \Rightarrow o_1 := \text{true}$$

$$gA_2 := y \Rightarrow o_2 := \text{true}$$

are found to be independent if executed with the clocked synchronous model, since both guarded actions do not share any variables and hence can be executed one after another without any disturbances to each other. Of course there is rather formal definition to the concept of weak endochrony, but as the last section of this chapter will introduce state-based weak endochrony, a concept closely resembling weak endochrony, the definition will be omitted at this point, since the idea is basically given by the independence notion. Yet the problem remains, that not every synchronous program can be correctly run on its own within some asynchronous environment. To cope with that, the next section introduces a wrapper algorithm, called the interpreter.

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

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

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

Figure 4.1: Truth tables for values $\mathbb{B} \cup \{\bot, \top\}$ and operator basis $\{\neg, \wedge, \vee\}$

4.1 Interpreter

The idea of the interpreter of this section is to execute each synchronous program based on its execution model, while simultaneously buffering all incoming environmental values. Most of this execution is done in the main function 'Interpret', which combines the reading, computations and writing of variables into one function. To shield each macro-step of the program from the asynchronous environment, the interpreter relies on buffers filled by the environment. These buffers can not only be read but also emptied to a certain degree. That is if a input variable value is assumed to be consumed by some macro-step, the interpreter can delete this value from the buffer, causing buffer values for this variable to be forwarded one position ahead. So while function 'ReadInputs' actually abstracts from the idea that the environment writes to the buffers, function 'ReadFromBuffer', reads buffer values required for the computation of the current state. If however the buffer cannot provide any value for a required variable, the variable is assumed to be absent. The main work then is carried out by the function 'ComputeReaction', which might either lead to a call of function 'ComputeReactionSynchronously' or 'ComputeReactionClockedSynchronously', depending on the execution model. Both functions model the computation of one macro-step according to the execution model. Yet they also allow to split one state, into its set of independent guarded actions, with each set executed in its own macro-step. Sounding odd at first, splitting up a macro-step into various macro-step gives the opportunity to output variable values as soon as they are computed and not, if

the all guarded actions have been executed as it would be normally the case. As a consequence systems receiving those values might start working ahead or might even be dependent on this particular value to start a computation at all. The option of splitting a macro-step is given by setting flags 'partialReaction' or 'sharedReadingFlag' in beginning of the initialization phase. With only 'partialReaction' set, macro-steps are split according the independence notion, that resulting reactions must be independent on present variables. In contrast sharedReadingFlag + partialReaction, allows more independence, since also reactions depending on variables only read during the reaction are considered independent. This can be better described as sharing input variables among guarded actions thus deleting some of the input dependencies given by both execution models. As example see guarded actions $x \Rightarrow o_1 := true$ and $x \wedge y \Rightarrow o_2 := true$, which would be normally considered dependent on x. But since x is only read during this state, it is can be shared due to the shared reading notion, such that both guarded actions can now be run in separate independent reactions. It must also be noted, that both execution functions might also have to cope with absent variables, that is guarded actions might be executed not only with boolean or integer values but also with the value absence itself. The responding behaviour is depicted for boolean operators in figure 4.1, while integer variable computations featuring absent values are always assumed to be evaluated to absent. Apart from the execution functions, modelling the particular execution model, the interpreter further allows to compute ahead. Though the 'Interpret' function still resides in state s, the interpreter can check if some successor states might actually compute some guarded actions, based on received buffer values and values from current computations. These computations might then be memorized and eventually submitted, if this successor state is indeed reached. Looking ahead of the current state is accomplished by setting flag 'computeAhead' and executed by function 'checkSuccessors'.

Algorithm 14 Basic Interpreter

```
function INTIALIZE
    partialReaction := false/true;
    computeAhead := false/true;
    sharedReadingFlag := false/true;
    for all x ∈ V^in do
        buffer[x] := new Array();
    end for
    for all s ∈ S do
        ϵ^s := ⊥;
        dϵ^s := ⊥;
        gA^s := ℒ(s);
    end for
end function

function INTERPRET(s)
    valid := false;
    repeat
        ReadInputs();                              ▷ env. writes to buffers
        buf := ReadFromBuffer(s);
        valid := ComputeReaction(s,buf);
    until (¬valid)
    DeleteFromBuffer(s);
    WriteOutputs(s);
    s' := ComputeSuccessor(s);
    interpret(s');
end function
```

Algorithm 15 Helper Functions Interpreter - Part I

function READFROMBUFFER(s)
  for all $v \in V^{in} \cap Var(\mathcal{L}(s)$ **do**
    buf[v] := buffer[v][0];
  end for
  return buf;
end function

function DELETEFROMBUFFER(s)
  if $gA^s = \mathcal{L}(s)$ **then**
    for all $v \in \{v \in V^{in} | \epsilon^s(v) \neq \bot\}$ **do**
      n := buffer[v].length;
      remove(buffer[v][n-1]);
      for i := 0 . . . (n-2) **do**
        buffer[v][i] := buffer[v][i+1];
      end for
    end for
  end if
end function

function RESET(s)
  for all $v \in V$ **do**
    $\epsilon^s(v) := \bot$;
    $d\epsilon^s(v) := \bot$;
  end for
  $gA^s := \mathcal{L}(s)$;
end function

function COMPUTESUCCESSOR
  s' = s;
  if $\forall g \in gA^s : [g]_{\epsilon^s} = true$ **then**
    s' = GetSuccessor(s);
    $d\epsilon^{temp} = d\epsilon^s$;
    reset(s); ▷ necessary, if s = s'
    $\epsilon^{s'} = d\epsilon^{temp}$;
  else
    $gA^s = gA^s \setminus \{g \in gA^s | [g]_{\epsilon^s} = true\}$;
  end if
  s_{temp} = s';
  if (computeAhead) **then**
    if s_{temp} = s **then**
      s_{temp} = GetSuccessor(s);
    end if
    if $s_{temp} \neq \bot \wedge s_{temp} \neq s$ **then**
      for all $v \in V^{in} \cap Var(\mathcal{L}(s) \setminus gA^s)$ **do**
        bufferIndex[v] = 1; ▷ over-approximation
      end for
      CheckSuccessors(s_{temp});
    end if
  end if
  return s';
end function

Algorithm 16 Helper Functions Interpreter - Part II

```
function GetSuccessor
    for all (s, c, s') ∈ T do
        if [c]_{ε^s} = true then
            return s';
        end if
    end for
end function

function CheckSuccessors(s)
    for all v ∈ Var: bufferIndex[v] = 0 do
        buf[v] := buffer[v][0];
    end for
    valid := ComputeReaction(s,buf);
    s' := GetSuccessor(s);
    if s' ≠ ⊥ ∧ s' ≠ s then
        for all v ∈ Var : ε^s(v) ≠ ⊥ do
            bufferIndex[v] := 1;
        end for
        dε^{s'} := dε^s;
        CheckSuccessors(s');
    end if
end function

function WriteOutputs(s)
    if ∀g ∈ gA^s : [g]_{ε^s} = true then
        write(ε^s);
    end if
end function
```

Algorithm 17 Synchronous Computation of Reaction

```
function ComputeReactionSynchronously(s,buf)
    valid := true;
    ϵ^cur := ϵ^s ⊔ buf;
    dϵ^cur := dϵ^s;

    repeat
        ϵ' := ϵ^cur;

        for all < γ ⇒ x := σ >∈ gA^s do
            if ([γ]_ϵ^cur = true) ∧ ([σ]_ϵ^cur ∉ {⊥, ⊤}) then
                ϵ^cur(x) := ϵ^cur(x) ⊔ [σ]_ϵ^cur;
            end if
        end for
        for all x ∈ V do
            if ∀ < γ ⇒ x := σ >∈ gA^s: [γ]_ϵ^cur = false then
                ϵ^cur(x) := ϵ^cur(x) ⊔ (type(x) == evt ? default(x) : ϵ^prv(x));
            end if
        end for
    until (ϵ^cur ≠ ϵ')

    for all < γ ⇒ next(x) := σ >∈ gA^s do
        if ([γ]_ϵ^cur = true) ∧ ([σ]_ϵ^cur ∉ {⊥, ⊤}) then
            dϵ^cur(x) := dϵ^cur(x) ⊔ [σ]_ϵ^cur;
        end if
    end for

    if partial ∧ ¬sharedReadingFlag then
        if ∃g ∈ gA^s : [grd(g)]_ϵ^cur = ⊥ ∧ ∃v ∈ rdVar(g) : ϵ^cur(v) ∉ {⊥, ⊤} then
            valid := false;
        end if
    end if

    if ¬partial ∧ ¬sharedReadingFlag then
        if ∃g ∈ gA^s : [g]_ϵ^cur = ⊥ then
            valid := false;
        end if
    end if

    if partial ∧ sharedReadingFlag then
        if ∃v ∈ V^loc.∀g ∈ gA^s: v ∈ usedVar(g) ∧ ϵ^cur(v) ∉ {⊥, ⊤} ⇒ [g]_ϵ^cur = ⊥ then
            valid := false;
        end if
    end if

    if valid then
        ϵ^s := ϵ^cur;
        dϵ^s := dϵ^cur;
    end if

    return valid;
end function
```

Algorithm 18 Clocked Synchronous Execution of Reaction

function COMPUTEREACTIONCLOCKEDSYNCHRONOUSLY(s,buf)
  valid := true;
  $\epsilon^{cur} := \epsilon^{s} \sqcup buf;$
  $d\epsilon^{cur} := d\epsilon^{s};$
  Clk^{cur}: = deriveInputClocks(ϵ^{s});

  repeat
    $\epsilon' := \epsilon^{cur};$
    Clk' = Clk^{cur};

    for all $< \gamma \Rightarrow x := \sigma > \in gA^{s}$ **do**
      if $([\gamma]_{\epsilon^{cur},Clk^{cur}} = true) \wedge ([\sigma]_{\epsilon^{cur},Clk^{cur}} \notin \{\bot, \top\})$ **then**
        $\epsilon^{cur}(x) := \epsilon^{cur}(x) \sqcup [\sigma]_{\epsilon^{cur},Clk^{cur}};$
        $Clk^{cur}(x) := true \sqcup Clk^{cur}(x);$
      end if
    end for
    for all $x \in V$ **do**
      if $\forall < \gamma \Rightarrow x := \sigma > \in gA^{s}$: $[\gamma]_{\epsilon^{cur},Clk^{cur}} = false \wedge Clk^{cur}(x) = true \wedge \epsilon^{cur}(x) = \bot$ **then**
        $\epsilon^{cur}(x) := \epsilon^{cur}(x) \sqcup$ (type(x) == evt ? default(x) : $\epsilon^{prv}(x)$);
      end if
    end for
  until $(\epsilon^{cur} \neq \epsilon' \wedge Clk' \neq Clk^{cur})$

  for all $< \gamma \Rightarrow next(x) := \sigma > \in gA^{s}$ **do**
    if $([\gamma]_{\epsilon^{cur},Clk^{cur}} = true) \wedge ([\sigma]_{\epsilon^{cur},Clk^{cur}} \notin \{\bot, \top\})$ **then**
      $\mathrm{d}\epsilon^{cur}(x) := \mathrm{d}\epsilon^{cur}(x) \sqcup [\sigma]_{\epsilon^{cur},Clk^{cur}};$
    end if
  end for

  if $partial \wedge \neg sharedReadingFlag$ **then**
    if $\exists g \in gA^{s} : [grd(g)]_{\epsilon^{cur},Clk^{cur}} = \bot \wedge \exists v \in rdVar(g) : \epsilon^{cur}(v) \notin \{\bot, \top\}$ **then**
      valid := false;
    end if
  end if

  if $\neg partial \wedge \neg sharedReadingFlag$ **then**
    if $\exists g \in gA^{s} : [g]_{\epsilon^{cur},Clk^{cur}} = \bot$ **then**
      valid := false;
    end if
  end if

  if $partial \wedge sharedReadingFlag$ **then**
    if $\exists v \in V^{loc}.\forall g \in gA^{s}$: $v \in usedVar(g) \wedge \epsilon^{cur}(v) \notin \{\bot, \top\} \Rightarrow [g]_{\epsilon^{cur},Clk^{cur}} = \bot$ **then**
      valid := false;
    end if
  end if

  if valid **then**
    for all $v \in V^{in} \cap V^{c}$: $\epsilon^{cur}(v) \notin \{\bot, \top\} \wedge \neg\exists g \in gA^{s}$:
$([grd(g)]_{\epsilon^{cur},Clk^{cur}} = false \wedge v \in grd(g))$
$\vee([grd(g)]_{\epsilon^{cur},Clk^{cur}} = true \wedge v \notin usedVar(g))$ **do**
      $\epsilon^{cur} = \bot;$
    end for
    $\epsilon^{s} := \epsilon^{cur};$
    $d\epsilon^{s} := d\epsilon^{cur};$
  end if

  return valid;
end function

Given the following guarded actions

$$
\begin{array}{r|c}
\hline
gA_1: & x \Rightarrow x := \text{true} \\
gA_2: & x \wedge y \Rightarrow o_1 := \text{true} \\
gA_3: & x \wedge z \Rightarrow o_2 := \text{true}
\end{array}
\quad \text{with bhv}(\{gA_1, gA_2, gA_3\}) :=
\begin{array}{c|c}
\text{x} & 1 \\
\text{y} & 1 \\
\text{z} & 1 \\
o_1 & 1 \\
o_2 & 1
\end{array}
$$

then shared reading also allows behaviours p_1, p_2:

$$
p_1 :=
\begin{array}{c|cc}
\text{x} & 1 & 1 \\
\text{y} & 1 & \perp \\
\text{z} & \perp & 1 \\
o_1 & 1 & \perp \\
o_2 & \perp & 1
\end{array}
\qquad \wedge \qquad
p_2 :=
\begin{array}{c|cc}
\text{x} & 1 & 1 \\
\text{y} & \perp & 1 \\
\text{z} & 1 & \perp \\
o_1 & \perp & 1 \\
o_2 & 1 & \perp
\end{array}
$$

Figure 4.2: Example for shared Reading

4.2 Shared Reading

Already put to work in the previous section, shared reading provides for a different independence notion than the one applied by weak endochrony. Yet it is missing any formal definition along with a thorough analysis of its implications towards behavioural properties of programs. These two issues will be addressed in the following. Given EFSM M over state set S = $\{S_1, \ldots, \S_n\}$, each behaviour p $\in$ Bhv(M) generated by some execution is assumed to be extended by a signal s: $\mathbb{N} \rightarrow \{1, \ldots, n\}$ indicating for each p_i, which state s $\in$ S produced p_i. This defines the overall behaviour as p_m := p $\sqcup$ s. Extending EFSM behaviours that way allows to think about each reaction of some behaviour of being indexed, by the unique state, which generated the corresponding reaction. As it is, this extension becomes particularly helpful, when talking about an interpreter applying shared reading on a synchronous/clocked program. Since shared reading leads to a set of reactions, all sharing the variables only read during the execution of some state[2],it is not compatible to the definition of weak endochrony and its notion of independence, which is based on the disjunction of variables sets being present. To elude those issues, the

[2]As already mentioned this is comparable to removing all input dependencies, which constrain the set of guarded actions of this state to be fired synchronously

concept of state-based weak endochrony is introduced, which is relies extensively on the definition of weak endochrony but is grounded on the notion of independence implied by the paradigm of shared reading. For a start, this new notion of independence is defined the following way: Two reactions r_s, r'_s generated by a state s are called independent, that is
$r_s \not\bowtie r'_s$, iff $\forall v \in V^{loc} \cup V^{out}$: $\neg(r_s(v) \neq \bot \wedge r'_s(v) \neq \bot) \wedge \exists r_s \sqcup r'_s$
and reactions $r_s, r'_{s'}$ are always dependent, as they are generated by a different states, s $\not=$ s'. The definition of joinability is kept, such that two reactions r_s, r'_s are joinable iff $\exists r_s \sqcup r'_s$. Along with the new definition of independency it is mandatory to redefine the operators for upper lower bound ($\sqcup$,$\sqcap$) and difference ($\backslash$), yielding the new operators $\sqcup'$, $\sqcap'$ and $\backslash'$. They are now defined as follows:
$\forall v \in V$: $r_s(v) \sqcup' r'_s(v) = r_s(v) \sqcup r'_s(v)$, provided r_s, r'_s are joinable
$$\forall v \in V\colon\ r_s(v)\sqcap' r'_s(v) = \begin{cases} r_s(v) & v \in Var(sga(r_s) \cap sga(r'_s)) \cap present(r_s \sqcap r'_s) \\ \bot & otherwise \end{cases}$$
$$\forall v \in V\colon\ r_s(v)\backslash' r'_s(v) = \begin{cases} r_s(v) & v \in Var(sga(r_s) \backslash sga(r'_s)) \cap present(r_s) \\ \bot & otherwise \end{cases}$$
with $sga(r_s) = \{g \in \mathcal{L}(s) | [g]_{r_s} = true\}$
An example of shared reading and the independency it allows, see 4.2. Going on, those redefinitions serve to introduce the definition of state-based weak endochrony:

A state-based system P over input variables V^{in} is considered state-based weakly endochronous, iff forall s $\in$ S it holds that:
W1: $\forall p \in$ Bhv(P) and $\alpha_s, \beta_s \in$ React(s) s.t. $p \cdot \alpha_s, p \cdot \beta_s \in$ Bhv(P), then:

1. $\alpha_s|^{V^{in}} = \beta|^{V^{in}} \Rightarrow \alpha_s = \beta_s$

W2: $\forall \alpha_s, \alpha'_s$ being independent and $p \cdot \alpha_s, p \cdot \alpha'_s \in$ Bhv(P) it holds:

1. $p \cdot \alpha_s \in$ Bhv(P) $\wedge$ $p \cdot \alpha'_s \in$ Bhv(P) $\Rightarrow p \cdot (\alpha \sqcup' \alpha'_s) \in$ Bhv(P)

2. $p \cdot \alpha_s \cdot \alpha'_s \in$ Bhv(P) $\Rightarrow p \cdot \alpha'_s \in$ Bhv(P)

W3: For joinable reactions α_s, α'_s with $p \cdot \alpha_s, p \cdot \alpha'_s \in$ Bhv(P) it holds that:

1. $p \cdot (\alpha_s \sqcap' \alpha'_s) \cdot (\alpha_s \backslash' \alpha'_s) \in$ Bhv(P)

2. $p \cdot (\alpha_s \sqcap' \alpha_s') \cdot (\alpha_s' \backslash' \alpha_s) \in \text{Bhv(P)}$

For every state of state-based weakly endochronous system, it can be shown that every state fufills the small diamond lemma[27], namely:
Given a state-based weakly endochronous system P over a state set S, then forall s ∈ S and independent reactions r_s, r_s' with $p \cdot r_s, p \cdot r_s' \in \text{Bhv(P)}$, the following holds:

- $p \cdot r_s \cdot r_s' \in \text{Bhv(P)}$
- $p \cdot r_s' \cdot r_s' \in \text{Bhv(P)}$
- $p \cdot (r_s \sqcup' r_s') \in \text{Bhv(P)}$

Proof. Assume r_s, r_s' independent actions for a given state s with $p \cdot r_s, p \cdot r_s' \in \text{Bhv(P)}$, for a given state-based weakly endochronous system P, then the implication W2.1 holds and from that $p \cdot (r_s \sqcup' r_s') \in \text{Bhv(P)}$ can be concluded. Since $r_s \sqcup' r_s'$ and r_s' are joinable, from W3 it follows that also $p \cdot ((r_s \sqcup' r_s') \sqcap' r_s') \cdot ((r_s \sqcup' r_s') \backslash' r_s')$ is a valid behaviour. For analysing the reaction $((r_s \sqcup' r_s') \backslash' r_s')$ it is best to look at the definition of the upper bound $\sqcap'$, which is defined for $\forall v \in Var(sga(r_s \sqcup' r_s') \cap sga(r_s')) \cap present((r_s \sqcup' r_s') \sqcap r_s')$ for this particular reaction. For simplifying this term, it is necessary to proof that:
$sga(r_s \sqcup' r_s') = sga(r_s \sqcup r_s') = sga(r_s) \cup sga(r_s')$
The first equivalence holds since operators $\sqcup'$ and $\sqcup$ are equivalent based on the definition of $\sqcup'$. Hence it is left to show that $sga(r_s \sqcup r_s') = sga(r_s) \cup sga(r_s')$. For that purpose assume $sga(r_s \sqcup r_s') \supset sga(r_s) \cup sga(r_s')$. Then $\exists g \in sga(r_s) \cup sga(r_s') \exists v \in Var(g)$: $([g]_{r_s} = \bot \wedge r_s(v) \neq \bot) \vee ([g]_{r_s'} = \bot \wedge r_s'(v) \neq \bot)$. This is not possible, since the execution model prevents such actions from being executed. Hence it assumed that $sga(r_s \sqcup r_s') \subset sga(r_s) \cup sga(r_s')$ holds. If that is so, there $\exists g \in sga(r_s) \cup sga(r_s')$: $([g]_{r_s} = true \vee [g]_{r_s'} = true) \wedge [g]_{r_s \cup r_s' = \bot}$. Since this leads to a contradiction, this subsumption does also not hold, which leaves the sole possibility that $sga(r_s \sqcup r_s') = sga(r_s) \cup sga(r_s')$. Having proven this equivalence, the term $Var(sga(r_s \cup' r_s') \cap sga(r_s'))$ simplifies to $Var((sga(r_s) \cup sga(r_s')) \cap sga(r_s')) = Var(sga(r_s'))$. The term $present((r_s \sqcup r_s') \sqcap r_s')$ on the other side simplifies to $present((r_s \sqcup r_s') \sqcap r_s') = present(r_s')$, which intersected with $Var(sga(r_s'))$ yields r_s'. Now for

the reaction $((r_s \sqcup r'_s)\backslash' r'_s)$ it is again helpful to have a look at the definition of difference operator $\backslash'$. For the reaction at hand the operator $\backslash'$ is defined for $\forall v \in Var(sga(r_s \sqcup' r'_s)\backslash sga(r'_s)) \cap present(r_s \sqcup r'_s)$. The term $Var(sga(r_s \sqcup' r'_s)\backslash sga(r'_s))$ can be simplified to $Var(sga(r_s \cup r'_s)\backslash sga(r'_s)) = Var(sga(r_s))$. Intersecting $Var(sga(r_s))$ with $present(r_s \sqcup r'_s)$ yields r_s, as reactions r_s and r'_s only share read-only variables for this particular behaviour. Using all those simplification $p \cdot ((r_s \sqcup' r'_s) \sqcap' r'_s) \cdot ((r_s \sqcup' r'_s)\backslash' r'_s)$ is equivalent to $p \cdot r'_s \cdot r_s$. With behaviours $p \cdot (r_s \sqcup' r'_s)$ and $p \cdot r_s$ as starting point, the behaviour $p \cdot r_s \cdot r'_s$ can be proven the very same way. $\square$

5 Summary

5.1 Conclusion

In the end it can be concluded, that most of the thesis dealt with synchronous Quartz programs and its relations to the term dependency. While alone the term dependency implies independency by language, it was shown that both concepts offer many interpretations as well as subtleties, which should be regarded in any serious work. In reference to that, the terms syntactic and behavioural dependency were established, both in meaning as well in mathematical definition. Equipped with the two execution models of Quartz, the synchronous and clocked synchronous model, and the knowledge of different representations for each program, a host of methods was introduced to answer one particular question. How to find every form of dependency inside a given representation, when the program utilizes not only boolean but also integer as well as indexed variables. In response to this question, methods and algorithms like the non-linear solver or the data-flow-equations for handling syntactic dependencies were introduced. They are accompanied by a decomposition criterion for state machines, an abstract interpreter for asynchronous environments and a new independence notion, called state-dependent weak endochrony. All of those methods relate to the primary goal of analysing dependencies and constructing concurrency for synchronous programs. Future elaborations on these topics might hopefully come to similar results or even include running examples of aforementioned algorithms, as many algorithms still leave room for optimization. Personally I like to hope that anyone reading this thesis will finish with some insights gained, since many hours of personal thinking have taught me one lesson: Every day, you learn something new.

Appendix

Min-/**Max**-definitions allow to maximize or minimize a respective expression:

	Expression		Rewrite	Condition
min	min[c]	$\Rightarrow$	c	
	min[v]	$\Rightarrow$	v	
	min[-x]	$\Rightarrow$	-min[x]	
	min[x + y]	$\Rightarrow$	min[x] + min[y]	
	min[x - y]	$\Rightarrow$	min[x] - max[y]	
	min[x * y]	$\Rightarrow$	min[x] * min[y]	$S_{min}(x) \geq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			min[x] * max[y]	$S_{max}(x) \leq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			max[x] * min[y]	$S_{min}(x) \geq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			max[x] * max[y]	$S_{max}(x) \leq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			Min(min[x]*min[y],...,max[x]*max[y])	*otherwise*
	min$[\frac{x}{y}]$	$\Rightarrow$	$\frac{min[x]}{min[y]}$	$S_{min}(x) \geq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			$\frac{min[x]}{max[y]}$	$S_{min}(x) \geq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			$\frac{max[x]}{min[y]}$	$S_{max}(x) \leq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			$\frac{max[x]}{max[y]}$	$S_{max}(x) \leq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			Min$(\frac{min[x]}{min[y]}, \ldots, \frac{max[x]}{max[y]})$	*otherwise*
	min$[x^n]$	$\Rightarrow$	min$[x]^n$	$S_{min}(x) \geq_{sig} 0 \wedge n \geq 0$
			min$[x]^n$	$S_{max} \leq_{sig} 0 \wedge ((n \geq 0 \wedge$ n mod $2 = 1) \vee (n \leq 0 \wedge$ n mod 2
			max$[x]^n$	$S_{min}(x) \geq_{sig} 0 \wedge n \leq 0$
			max$[x]^n$	$S_{max} \leq_{sig} 0 \wedge ((n \geq 0 \wedge$ n mod $2 = 0) \vee (n \leq 0 \wedge$ n mod
			Min$(min[x]^n, max[x]^n)$	*otherwise*
max	max[c]	$\Rightarrow$	c	
	max[v]	$\Rightarrow$	v	
	max[-x]	$\Rightarrow$	-max[x]	
	max[x + y]	$\Rightarrow$	max[x] + max[y]	
	max[x - y]	$\Rightarrow$	max[x] - min[y]	
	max[x * y]	$\Rightarrow$	min[x] * min[y]	$S_{max}(x) \leq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			min[x] * max[y]	$S_{max}(x) \leq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			max[x] * min[y]	$S_{min}(x) \geq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			max[x] * max[y]	$S_{min}(x) \geq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			Max(min[x]*min[y],...,max[x]*max[y])	*otherwise*
	max$[\frac{x}{y}]$	$\Rightarrow$	$\frac{min[x]}{min[y]}$	$S_{max}(x) \leq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			$\frac{min[x]}{max[y]}$	$S_{min}(x) \geq_{sig} 0 \wedge S_{max}(y) \leq_{sig} 0$
			$\frac{max[x]}{min[y]}$	$S_{min}(x) \geq_{sig} 0 \wedge S_{min}(y) \geq_{sig} 0$
			$\frac{max[x]}{max[y]}$	$S_{max}(x) \leq_{sig} 0 \wedge S_{min}(y) \leq_{sig} 0$
			Max$(\frac{min[x]}{min[y]}, \ldots, \frac{max[x]}{max[y]})$	*otherwise*
	max$[x^n]$	$\Rightarrow$	min$[x]^n$	$S_{min}(x) \geq_{sig} 0 \wedge n \leq 0$
			min$[x]^n$	$S_{max} \leq_{sig} 0 \wedge ((n \geq 0 \wedge$ n mod $2 = 0) \vee (n \leq 0 \wedge$ n mod
			max$[x]^n$	$S_{min}(x) \geq_{sig} 0 \wedge n \geq 0$
			max$[x]^n$	$S_{max} \leq_{sig} 0 \wedge ((n \geq 0 \wedge$ n mod $2 = 1) \vee (n \leq 0 \wedge$ n mod
			Max$(min[x]^n, max[x]^n)$	*otherwise*

Function '**sig**' allows to check the monotonic properties of some expression. It is therefore defined over the set set - , 0 , + with partial order: $- <_{sig} 0 <_{sig} +$
and rewrite rules:

<table>
<tr><th></th><th>Expression</th><th></th><th>Rewrite</th></tr>
<tr><td rowspan="9">sig</td><td>sig[c]</td><td>⇒</td><td>0</td></tr>
<tr><td>sig[v]</td><td>⇒</td><td>0</td></tr>
<tr><td>sig[-x]</td><td>⇒</td><td><table><tr><th>x</th><th>-x</th></tr><tr><td>-</td><td>+</td></tr><tr><td>0</td><td>0</td></tr><tr><td>+</td><td>-</td></tr></table></td></tr>
<tr><td>sig[x + y]</td><td>⇒</td><td><table><tr><th>$+_{op}$</th><th>-</th><th>0</th><th>+</th></tr><tr><td>-</td><td>-</td><td>-</td><td>⊥</td></tr><tr><td>0</td><td>-</td><td>0</td><td>+</td></tr><tr><td>+</td><td>⊥</td><td>+</td><td>+</td></tr></table></td></tr>
<tr><td>sig[x - y]</td><td>⇒</td><td><table><tr><th>$-_{op}$</th><th>-</th><th>0</th><th>+</th></tr><tr><td>-</td><td>⊥</td><td>-</td><td>-</td></tr><tr><td>0</td><td>+</td><td>0</td><td>-</td></tr><tr><td>+</td><td>+</td><td>+</td><td>⊥</td></tr></table></td></tr>
<tr><td>sig[x * y]</td><td>⇒</td><td><table><tr><th>$*_{op}$</th><th>-</th><th>0</th><th>+</th></tr><tr><td>-</td><td>+</td><td>0</td><td>-</td></tr><tr><td>0</td><td>0</td><td>0</td><td>0</td></tr><tr><td>+</td><td>-</td><td>0</td><td>+</td></tr></table></td></tr>
<tr><td>sig[x / y]</td><td>⇒</td><td><table><tr><th>$/_{op}$</th><th>-</th><th>0</th><th>+</th></tr><tr><td>-</td><td>+</td><td>⊥</td><td>-</td></tr><tr><td>0</td><td>0</td><td>⊥</td><td>0</td></tr><tr><td>+</td><td>-</td><td>⊥</td><td>+</td></tr></table></td></tr>
<tr><td>sig[x^n]</td><td>⇒</td><td><table><tr><th>$.^{op}$</th><th>-</th><th>0</th><th>+</th></tr><tr><td>-</td><td>⊥</td><td>+</td><td>+</td></tr><tr><td>0</td><td>+</td><td>+</td><td>+</td></tr><tr><td>+</td><td>+</td><td>+</td><td>+</td></tr></table></td></tr>
</table>

Before function 'sig' might be used, the following intialization must take place:

for all $x \in V$ **do**
  $Env_{min} := s_{min}$;
  $Env_{max} := s_{max}$;
end for
tS_{min} := TopologicalSort(s_{min};
tS_{max} := TopologicalSort(s_{max};

for $i_{min} := 1$ to $|tS_{min}|$ **do**
  $var_{i_{min}} := tS_{min}|^{i_{min}}$;
  ($var_{i_{min}}$, t) := Substitute($(var_{i_{min}}, s_{min}(var_{i_{min}}))$, s_{min}, FV($s_{min}(var_{i_{min}})$));
  $Env_{min}(var_{i_{min}})$:= sig(t);

end for

for $i_{max} := 1$ to $|tS_{max}|$ **do**
$var_{i_{max}} := tS_{min}|^{i_{max}}$;
($var_{i_{max}}$, t) := Substitute($(var_{i_{max}}, s_{max}(var_{i_{max}}))$, s_{max}, FV($s_{max}(var_{i_{min}})$));
$Env_{max}(var_{i_{max}})$:= sig(t);
end for
function S_{min}(x)
for all $v \in FV(x)$ **do**
x := $[x]_v^{Env_{min}(v)}$;
end for
return sig(x);
end function
function S_{max}(x)
for all $v \in FV(x)$ **do**
x := $[x]_v^{Env_{max}(v)}$;
end for
return sig(x);
end function

Boundary Analysis allows to estimate loop iterations
for boolean expressions consisting of integer dependent predicates:

	Expression		Rewrite
BA	BA[α]	$\Rightarrow$	$\mathbb{N}$
	BA[$\neg\alpha$]	$\Rightarrow$	BA[α]
	BA[$\alpha \wedge \beta$]	$\Rightarrow$	BA[α] $\cap$ BA[β]
	BA[$\alpha \vee \beta$]	$\Rightarrow$	BA[α] $\cup$ BA[β]
	BA[x $\leq$ y]	$\Rightarrow$	max[y] - min[x]
	BA[x $<$ y]	$\Rightarrow$	BA[x $\leq$ y] - 1
	BA[x $\geq$ y]	$\Rightarrow$	max[x] - min[y]
	BA[x $>$ y]	$\Rightarrow$	BA[x $\geq$ y] - 1

Bibliography

[1] B. Caillaud A. Beneviste and P. Le Guernic. Compositionality in dataflow synchronous languages: specification and distributed code generation 1,2,3. *Inf. Comput.*, 163(1):125–171, November 2000.

[2] David F. Bacon, Susan L. Graham, and Oliver J. Sharp. Compiler transformations for high-performance computing. *ACM Comput. Surv.*, 26(4):345–420, December 1994.

[3] Y. Bai, J. Brandt, and K. Schneider. Data-flow analysis of extended finite state machines. In B. Caillaud, J. Carmona, and K. Hiraishi, editors, *Application of Concurrency to System Design (ACSD)*, pages 163–172, Newcastle Upon Tyne, England, UK, 2011. IEEE Computer Society.

[4] Y. Bai and K. Schneider. Isochronous networks by construction. In *Design, Automation and Test in Europe (DATE)*, Dresden, Germany, 2014. IEEE Computer Society.

[5] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proceedings of the IEEE*, 91(1):64–83, 2003.

[6] J. Brandt, M. Gemünde, K. Schneider, S.K. Shukla, and J.-P. Talpin. Embedding polychrony into synchrony. *IEEE Transactions on Software Engineering (TSE)*, 39(7):917–929, July 2013.

[7] J. Brandt and K. Schneider. Static data-flow analysis of synchronous programs. In *Proceedings of the 7th IEEE/ACM international conference on Formal Methods and Models for Codesign*, MEMOCODE'09, pages 161–170, Piscataway, NJ, USA, 2009. IEEE Press.

[8] J. Brandt, K. Schneider, and Y. Bai. Passive code in synchronous programs. *Transactions on Embedded Computing Systems (TECS)*, 2014.

[9] D.M. Chapiro. *Globally-Asynchronous, Locally-Synchronous Systems*. PhD thesis, Department of Computer Science, Stanford, California, USA, 1984.

[10] Mohammad R. Haghighat and Constantine D. Polychronopoulos. Symbolic analysis for parallelizing compilers, 1994.

[11] N. Halbwachs. *Synchronous programming of reactive systems*. Kluwer, 1993.

[12] C. Karfa, K. Banerjee, D. Sarkar, and C. Mandal. Verification of loop and arithmetic transformations of array-intensive behaviors. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 32(11):1787–1800, Nov 2013.

[13] Ken Kennedy and John R. Allen. *Optimizing Compilers for Modern Architectures: A Dependence-based Approach*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2002.

[14] Amy W. Lim and Monica S. Lam. Maximizing parallelism and minimizing synchronization with affine transforms. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 201–214, New York, NY, USA, 1997. ACM.

[15] Dror E. Maydan, Saman P. Amarasinghe, and Monica S. Lam. Array-data flow analysis and its use in array privatization. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 2–15, New York, NY, USA, 1993. ACM.

[16] M. Nanjundappa, M. Kracht, J. Ouy, and S.K. Shukla. A new multi-threaded code synthesis methodology and tool for correct-by-construction synthesis from polychronous specifications. In *Application of Concurrency to System Design (ACSD), 2013 13th International Conference on*, pages 21–30, July 2013.

[17] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. In *Application of Concurrency to System Design (ACSD)*, pages 48–57, Saint-Malo, France, 2005. IEEE Computer Society.

[18] D. Potop-Butucaru and B. Caillaud. Correct-by-construction asynchronous implementation of modular synchronous specifications. IRISA Technical report PI 1670, Institut National de Recherche en Informatique et en Automatique (INRIA), 2005.

[19] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. In *Application of Concurrency to System Design (ACSD)*, pages 67–76, Hamilton, Ontario, Canada, 2004. IEEE Computer Society.

[20] D. Potop-Butucaru, B. Caillaud, and A. Benveniste. Concurrency in synchronous systems. *Formal Methods in System Design (FMSD)*, 28(2):111–130, 2006.

[21] D. Potop-Butucaru, R. de Simone, and Y. Sorel. From multi-clock constraints to multi-rate GALS executives. Research Report 6021, INRIA, Sophia Antipolis, France, November 2006.

[22] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. In C.M. Kirsch and R. Wilhelm, editors, *Embedded Software (EMSOFT)*, pages 124–133, Salzburg, Austria, 2007. ACM.

[23] D. Potop-Butucaru, R. de Simone, and Y. Sorel. Necessary and sufficient conditions for deterministic desynchronization. Research Report 6152, INRIA, Sophia Antipolis, France, March 2007.

[24] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. Clock-driven distributed real-time implementation of endochronous synchronous programs. In S. Chakraborty and N. Halbwachs, editors, *Embedded Software (EMSOFT)*, pages 147–156, Grenoble, France, 2009. ACM.

[25] D. Potop-Butucaru, R. de Simone, Y. Sorel, and J.-P. Talpin. From concurrent multi-clock programs to deterministic asynchronous implementations. In S. Edwards, R. Lorenz, and W. Vogler, editors, *Application of Concurrency to System Design (ACSD)*, pages 42–51, Augsburg, Germany, 2009. IEEE Computer Society.

[26] William Pugh. The omega test: A fast and practical integer programming algorithm for dependence analysis. In *Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, Supercomputing ’91, pages 4–13, New York, NY, USA, 1991. ACM.

[27] K. Schneider. The synchronous programming language quartz, 2009.

[28] Jean-Pierre Talpin, Julien Ouy, Loïc Besnard, and Paul Le Guernic. Compositional design of isochronous systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE ’08, pages 928–933, New York, NY, USA, 2008. ACM.

[29] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[30] Robert A. van Engelen, Johnnie Birch, Yixin Shou, Burt Walsh, and Kyle Gallivan. A unified framework for nonlinear dependence testing and symbolic analysis. In Paul Feautrier, James R. Goodman, and André Seznec, editors, *ICS*, pages 106–115. ACM, 2004.

[31] Peng Wu, Albert Cohen, and David Padua. Induction variable analysis without idiom recognition: Beyond monotonicity. In *In Proceedings of the 14th International Workshop on Languages and Compilers for Parallel Computing*, page 2624, 2001.