

System Description Aspects as Syntactic Sugar

Jens Brandt and Klaus Schneider
Reactive Systems Group
Department of Computer Science
University of Kaiserslautern
<http://rsg.informatik.uni-kl.de>

Abstract

Many different system description and specification languages are used in modern design flows to emphasize different aspects like modular architecture, multi-threaded behavior, abstract action-oriented behavior, and the desired temporal properties. However, the use of many specialized languages complicates the development of seamless and robust design flows. In this paper, we show that synchronous languages are powerful enough to capture the mentioned aspects of system descriptions as simple syntactic sugar. In particular, we show how hardware structures, multi-threaded and action-oriented programs as well as property specification languages can be incorporated in a synchronous programming language so that a single core language with a powerful compiler can handle all design descriptions in a consistent way.

1 Introduction

Many embedded systems are heterogeneous in the sense that they consist of parts that are currently developed by totally different design flows and languages. The most relevant distinction is thereby the partition into application-specific hardware and software. Different hardware architectures like DSPs or standard microprocessors also require different ways for the implementation of the software part, which often implies the use of specialized languages and compilers. Various levels of abstraction make use of different modeling paradigms like action-oriented or multi-threaded system behavior. Moreover, specialized languages are used to describe the desired functional and temporal properties for the simulation and verification phases. The set of languages used in modern system design therefore includes languages like system description languages as UML [30], System-C [28], and SystemVerilog [2], hardware description languages like VHDL [41] and Verilog [27], property specification languages like PSL [1], and of course, traditional programming languages like C.

However, the plethora of languages currently used in many design flows is simply unmanageable. Depending on many tools and licenses requires to update designs almost all the time, which makes the overall design process inefficient. Moreover, it is a big disadvantage that specialized languages force the designer to already think about a later realization (in software or hardware), which makes late design changes concerning the hardware-software partitioning or even weaker modifications of the architecture practically impossible. Specialized languages like Accellera's property specification language PSL [2] became moreover very complicated languages. However, the most difficult problem that results from the use of so many languages is that the underlying models of the languages do not always match: For example, hardware description languages mostly rely on an event based simulation, while software programs mostly rely on a sequential uniprocessor execution, and property specifications consider formal models like transition systems.

For this reason, we propose in this paper a unified approach that focuses on a single programming paradigm. This consolidation is the result of many bad experiences made with complex design flows. To this end, we propose the synchronous programming paradigm [3], since synchronous languages have already proved to be able to generate both application-specific hardware and software from the same synchronous program. Moreover, the underlying semantics match with the models used in formal verification like model checking.

However, synchronous languages currently do not offer much support for the specification of temporal specifications. Usually, observers are written as synchronous programs that are then run in parallel with the system and report whether something bad has happened. Compilers for synchronous languages are able to traverse the state space and check whether such a situation can happen, so that checking safety properties is already comfortably embedded in synchronous programming environments. However, checking more complex specifications like liveness, different kinds of fairness, or even assume-guarantee reasoning is not yet available in commercial tools.

Based on the Esterel language, we developed our own experimental language called Quartz and implemented an entire tool framework called AVerest for this language. The current version of AVerest that is available under www.averest.org can be used to translate synchronous programs to hardware (netlists given in VHDL or Verilog) or software (programs written in C) [38, 39], and to verify given specifications written in temporal logics like LTL, CTL, and even in the full μ -calculus [37]. In this paper, we present the new version of AVerest that is currently in an experimental status, and therefore not yet publically available. During the development of this version, we also discussed additional special syntactic means to describe the modular architecture of a system, and also more powerful specifications to support development processes based

on refinement techniques. However, having considered many languages and approaches, we came to the conclusion that most of these aspects can be implemented without much effort as simple syntactic sugar on top of our existing language.

This paper therefore demonstrates the expressive power of the synchronous programming model in that we show how different aspects like the description of the modular architecture, complex properties as given by ω -regular properties similar to PSL and assume-guarantee reasoning can be incorporated in a synchronous language. In particular, we show that regular expressions and temporal logics can be easily translated to even more readable synchronous programs. Besides the better readability of the specification, the main advantage is the possibility to use existing tools for synchronous languages to simulate, verify, and debug the specified properties as well as the program. Moreover, all translations to hardware and software already offered by compilers for synchronous languages can still be used for hardware-software codesign.

The outline of the paper is as follows: In the next section, we briefly consider the core of our synchronous language Quartz, which is the input language of our Averest system. In the following sections, we then show how structural descriptions, action-oriented descriptions, and property specifications are easily obtained in Quartz, so that these aspects can all be handled in a unique framework. For this reason, it is very simple to check their equivalence by means of verification or simulation.

2 The Averest System

Averest is a set of tools for specification, verification, and implementation of reactive systems. It includes a compiler for synchronous programs, a symbolic model checker, and a code generator for hardware and/or software synthesis. Further tools for event-based simulation as well as for supervisor synthesis are currently under development. Averest can be used for modeling and verifying finite as well as infinite state systems at various levels of abstraction and with different styles of descriptions. In particular, Averest is not only well-suited for the design of integrated circuits, but also for developing communication protocols, concurrent programs, software in embedded systems, etc.

Systems descriptions for Averest are given in the imperative synchronous programming language Quartz [34–36, 38]. Quartz is an imperative synchronous language that was derived from the Esterel language [5, 6]. The common paradigm of synchronous languages is the perfect synchrony [3, 17], which means that most of the statements are executed as micro steps in zero time. Consumption of time is explicitly programmed by special statements which partition the micro steps into macro steps. In the programmer’s view, all macro steps take the same amount of logical time. Thus, concurrent threads run in lockstep and automatically syn-

chronize at the end of their macro steps. The introduction of micro- and macrosteps is not only a convenient programming model, it is also the key to generate *deterministic* single-threaded code from multi-threaded synchronous programs. Thus, synchronous programs can be executed on ordinary microcontrollers without complex operating systems. As another advantage, the translation of synchronous programs to hardware circuits is straightforward [4, 33]. Moreover, the formal semantics of synchronous languages makes them particularly attractive for reasoning about program semantics and correctness. Therefore, synchronous languages are well-suited for the design of safety-critical embedded systems that consist of application-specific hardware and software.

In this paper, we mainly focus on the core of the Quartz language that is powerful enough to define many other statements as simple syntactic sugar. This core language consists of the following basic statements:

Definition 1 [Basic Statements of Quartz] *The set of basic statements of Quartz is the smallest set that satisfies the following rules, provided that S , S_1 , and S_2 are also basic statements of Quartz, ℓ is a location variable, x is an event variable, y is a state variable, σ is a Boolean expression, and α is a type:*

- nothing (empty statement)
- $y = \tau$ and $\text{next}(y) = \tau$ (assignments)
- ℓ : pause (consumption of time)
- if (σ) S_1 else S_2 (conditional)
- $S_1; S_2$ (sequential composition)
- $S_1 \parallel S_2$ (synchronous concurrency)
- do S while(σ) (iteration)
- [weak] suspend S when [immediate](σ) (suspension)
- [weak] abort S when [immediate](σ) (abortion)
- $\{\alpha \ y; \ S\}$ (local variable y with type α)
- choose S_1 else S_2 (nondeterministic choice)
- assume(φ) (inline assumption)
- assert(φ) (inline specification)
- name(τ_1, \dots, τ_n) (module instance)

Many other statements can be defined as macro statements, as e.g. the following always statement:

$$\text{always } S := \text{do } S; \text{pause}; \text{while true}$$

In addition to many Esterel statements that can be defined in the above spirit as syntactic sugar, the Quartz language features moreover generic programs (compile time parameters), different forms of concurrency (synchronous, asynchronous, interleaved), explicit nondeterministic choice, fixed bitwidth integers with a complete set of arithmetic operations, arrays, infinite integers and temporal logic specifications.

In Quartz, there are two kinds of (local and output) variables, namely *event* and *state variables*. State variables y are persistent, i.e., they store their current

value until an assignment changes it. Executing a delayed assignment $next(y) = \tau$ means to evaluate τ in the current macro step (environment) and to assign the obtained value to y in the following macro step. Immediate assignments update y in the current macro step and are therefore rather equations than assignments.

Event variables do not store their values, hence, an assignment $y = \tau$ to an event variable just gives y the value τ for this moment of time (unless there is another assignment in the next instant with the same value). If no assignment takes place, the value is not stored, instead, a default value is taken. As most events are of Boolean type, we use the statements *emit x* and *emit next(x)* as macros for $y = \text{true}$ and $next(y) = \text{true}$, respectively.

There is only one basic statement that defines a control flow location, namely the *pause* statement¹. For this reason, we optionally² endow *pause* statements with unique Boolean valued *location variables* ℓ that are true iff the control is currently at location ℓ : *pause*.

The semantics of the other statements is essentially the same as in Esterel. Due to lack of space, we do not describe their semantics in detail, and refer instead to [34–36] and, in particular, to the Esterel primer [5], which is an excellent introduction to synchronous programming.

In addition to the above statements, there are many more statements that are, however, simply reduced to the above core language. For example, using oracle inputs with nondeterministic choice, different kinds of concurrency like asynchronous and interleaved concurrency can be easily reduced to synchronous concurrency.

Moreover, Quartz supports *generic programming* in that sequences, and parallel statements can be described via compile-time parameters. For example, it is therefore possible to implement a system with n similar threads or with n -bit wide variables, or with arrays of length n . Moreover, statements can be given by primitive recursion so that n if-then-else statements can be nested in each other.

```

module BehaveDETECT110(event i,&o)
implements SpecDETECT110(i,o) {
  event prv_i, prv_prv_i;
  loop {
    if(i) emit next(prv_i);
    if(prv_i) emit next(prv_prv_i);
    if(!i&prv_i&prv_prv_i) emit o;
    pause;
  }
}

```

Figure 1: Behavioral Description of a 110-Detector

¹To be precise, immediate forms of suspend also have this ability.

²In case the programmer does not provide a location variable, the compiler will automatically generate one.

Quartz programs are simply a list of modules, where only the first (main) module is considered by the compiler. The other modules are instantiated in the first module, or they are simply not used. A Quartz module consists of a header that contains the name of the module, the declaration of the inputs and outputs³, as well as the body statement. A very simple example module is shown in Figure 1 that describes a module that checks whether the boolean-valued input stream i contains 110 as a subsequence. In case this subsequence has been read, the output o is made true.

3 Structural Descriptions

Structural descriptions emphasize the hierarchy that is obtained by the composition of modules to new modules at the next level of the hierarchy. Structural descriptions are particularly popular in classic hardware design. In Quartz, module instances can be used as ordinary statements, so that instances can run in sequence or in parallel, and can interact with each other.

```

module AND(event a,b,&out) {
  always
    if(a&b) emit out;
}

module OR(event a,b,&out) {
  always
    if(a|b) emit out;
}

module NEG(event a,&out) {
  always
    if(!a) emit out;
}

module DFF(event a,&out) {
  always
    if(a) emit next(out);
}

```

Figure 2: Behaviors of Basic Hardware Gates

It is therefore straightforward to describe modular hardware structures by Quartz modules. To this end, no special syntax is necessary, all that is required is a restriction to certain statements: *Basic structural modules*, which form the leaves of the hierarchy, are constructed by a behavioral description, i.e. their modules' bodies contain a Quartz statement. Figure 2 shows some simple definitions of basic hardware gates.

Composed structural modules simply consist of a local declaration of the internal variables (wires in the case of hardware circuits), and a parallel execution of the instantiated hardware modules. It might seem to

³Outputs are distinguished from inputs by prefixing their name with a $\&$ symbol.

be wasteful to run a single thread for each hardware gate, but this may not be the case in the generated code: The compiler is able to merge all the loops that are obtained by expansion of the modules, so that a single thread can simulate the gate netlist in software (if wanted). A similar argument holds for hardware code generation, so that the number of location variables is reduced to only one pause statement (all the location variables are easily detected to have the same transition relations, so that the compiler can unify them all).

```

module DETECT110_Structure(event i,&o)
implements DETECT110_Spec(i,o){
  event w1, w2, w3, w4, w5, w6, w7;
  { NEG(i,w1);
  || AND(i,w7,w2);
  || DFF(w2,w3);
  || NEG(w7,w4);
  || AND(i,w4,w5);
  || DFF(w5,w6);
  || OR(w3,w6,w7);
  }
}

```

Figure 3: Structural Implementation of a 110-Detector

An example is shown in Figure 3, where we implemented a hardware circuit for detecting a 110 subsequence in the input stream. As can be seen, structural descriptions are naturally obtained in a synchronous language like Quartz. Moreover, no special treatment is required for the compiler: it is obviously still possible to generate hardware and software from these modules. In case of hardware design, this offers the benefit of a highly efficient simulation at the synchronous level by compiling the obtained C programs individually for the particular design. For hardware synthesis, the compiler essentially generates the gates that have been used in the description. Hence, there is no loss of the hardware structure and neither a loss of efficiency due to the compilation.

4 Action Languages

Action languages are frequently used for modeling software or hardware systems at an abstract level. Examples of action languages are Unity [11], DisCo [22], TLA [23], sublanguages used in UML⁴ as well as languages used in model checkers like Murphi [14] or ALF [9].

The general model of computation of these languages is thereby that a program consists of a set of actions that are executed whenever an associated condition holds. In this sense, this computation model is related to the ‘guarded commands’ that were already considered by Dijkstra [12].

The translation of Quartz programs consists of computing also a set of guarded commands [39]: The com-

piler computes for each assignment $x = \tau$ the guard condition, i.e. the condition that holds iff the assignment is executed. Then, for each variable x , the hardware code generation will generate a case construct that checks the different guards and assigns the corresponding right hand sides.

It is therefore straightforward to directly integrate action languages in the Quartz language. The compiler is thereby even simplified, since the conditional actions can be directly used for the intermediate data structures of the compiler. Moreover, it is very simple to implement a given set of conditional actions as a Quartz module: given actions $(\gamma_1, \alpha_1), \dots, (\gamma_n, \alpha_n)$, we simply use the following module:

```

module ModuleName(...){
  loop{
    if( $\gamma_1$ )  $\alpha_1$ ;
    :
    if( $\gamma_n$ )  $\alpha_n$ ;
    pause;
  }
}

```

The above scheme is very simple, but nevertheless very powerful. In particular, the combination of other Quartz statements like parallel execution, abortion, suspension, etc. allows us to capture also complex module transformations like merging of action modules and many others.

5 Property Specification

In this section, we consider the main ingredients of property specifications available in Quartz. Quartz offers additional constructs to specify complex temporal properties that we have to omit due to lack of space. In particular, temporal logics like CTL, LTL, special fragments of CTL*, as well as the full μ -calculus can be used for this purpose. In particular, past temporal operators are often very convenient as can be seen even with the simple example given in Figure 4: **s1** means that at all points of time (**G**) on all computation paths (**A**) of the system, **o** holds iff currently **i** is false, and **i** was true at the preceding two points of time. **s2** is another temporal specification stating the same property with only future operators. Finally, **s3** considers only one implication, and can therefore make use of the CTL logic and its more efficient model checking algorithms.

```

spec SpecDETECT110(event i,&o) {
  s1 : A G ( o <-> !i & PSX ( i & PSX i ) );
  s2 : A G ( i & X i & X X !i <-> X X o );
  s3 : A G ( i -> A X ( i -> A X ( !i -> o ) ) );
}

```

Figure 4: Temporal Specification of a 110-Detector

⁴see www.omg.org

Besides the possibility to simply list temporal logic specifications as shown in the specification module given in Figure 4, Quartz offers also inline specifications that refer to the corresponding control flow location of the program. These inline specifications are similar to those developed for VHDL in [32]. Moreover, it is possible to make use of regular and ω -regular expressions similar to Accellera’s industry standard property specification language PSL. In contrast to PSL, we make use of Quartz statements to replace regular expressions by more readable program statements.

In the remainder of this section, we first present the possibilities of inline specifications and assumptions to support readable specifications as well as assume-guarantee reasoning. Then, we show how regular expressions can be translated to Quartz statements, and finally we briefly discuss similar translations for temporal logics.

5.1 Inline Assertions and Assumptions

Verifying a system aims at ensuring that the system has exactly the specified behavior. Traditionally, two kinds of specifications can be distinguished: *White box specifications* may refer to internal states or local variables of a module, while *black box specifications* only describe the external behavior of a module without referring to internals. In general, black box specifications should be preferred to support a hierarchical reasoning that is independent of a particular implementation. However, a specification may not hold all the time, but only after reaching some point of the computation which is conveniently defined with inline (white box) specifications.

In traditional programming languages, which lack a built-in verification support, programmers are used to employ assertions for this task: They specify a condition that should hold at a particular control flow location by means of *assertion statements*. Whenever the control flow hits such a statement during the execution of the program, the given condition is checked and the program is aborted if the condition is violated.

Quartz supports both black box and white box specifications by means of *assume* and *assert* statements. Like any statement, they can be used at an arbitrary point of a Quartz module and both statements take a list of labelled conditions, where the conditions can be given in the temporal logic LTL. However, their semantics is completely different: *Assume statements list conditions that the programmer assumes to hold at the corresponding control flow location, whereas assert statements list conditions that have to be checked at that location.* The verification tool trusts the programmer and uses the assumptions to check the other conditions. The task of assume statements is to give the verification tool additional information that can not be derived from the program alone. For example, knowledge about the environment and possible input values or complex mathematical relations can be added in this way. In many cases, it is possible to check the given assumptions if the final context is given by other mod-

ules, but some assumptions are directly given by the environment and can therefore not be checked unless a model of the environment is provided.

Of course, assume guarantee reasoning [19, 29, 42] is directly supported by the assume and assert statements. Moreover, statements in a module can be annotated with classical pre-/postconditions and loop invariants [13, 15, 20]. The task of the compiler is the extraction of the given assumptions and proof obligations and their hand-over to a model checker. Additionally, it creates specifications for common program errors: For example, for bitvectors, it is checked whether an overflow occurs due to arithmetic expressions, and for arrays, it is checked that the index expressions remain in the declared bounds.

Assume/assert statements implement a white-box specification. Specification of the black box behavior is achieved by defining an *abstract module* (an example is shown in Figure 4). The body of an abstract module does not consist of a statement; instead, a list of temporal specification is given that have to be fulfilled by any concrete module that *implements the abstract module*. The *implements* clause in the header of a concrete module tells the compiler to set up also proof obligations to check the temporal properties of the referred abstract module. These mechanisms allow us to establish a refinement relation between two modules similar to the B-method [24, 25].

Abstract modules are again only syntactic sugar: In principle, they consist of a sequence of assert statements that list the temporal specifications. For code generation, assume and assert statements are implemented in an observer that will set an error flag in case the property is violated during runtime.

In this way, Quartz allows the programmer the step-wise refinement of a given system. Starting with a pure temporal specification, proceeding with an intermediate action-oriented description, and finally concluding with a behavioral Quartz statement, a programmer can refine a given model of a system so that each refinement step can be proved to be correct. Moreover, as Averest is capable to handle unbounded integers, one can first start with unbounded integers, then one could check how large value may grow to finally determine sufficient bitwidths. Hence, also the data types can be refined during this process.

5.2 Regular Expressions and Finite Automata

In the previous section, we have shown where specifications can be placed in a module, but we left open, which logic is used to write down the formal specifications. Quartz allows one to use temporal logic specifications, in particular, the linear temporal logic LTL and the branching time temporal logic CTL. Moreover, certain fragments of the more powerful logic CTL* are considered, and even the full μ -calculus is supported,

which is currently one of the most expressive specification logics [37].

We discussed also the use of Accellera's property specification language PSL for future versions of *Averest*, but came to the conclusion that synchronous languages can be used to make such specification much more readable: While PSL also provides LTL and CTL, it additionally considers regular expressions and finite as well as infinite paths in order to support both simulation and verification, respectively.

Regular expressions can also be added to *Quartz* as syntactic sugar: It is well-known that regular expressions, right- and left-linear grammars, and finite automata in different variants are equivalent to each other [7, 8, 10, 18, 21, 31]. Regular expressions are a very simple, yet powerful formalism to describe languages that can be accepted by finite state automata. Note that regular expressions describe languages of finite words over a fixed alphabet Σ . For the specification of reactive systems, however, also infinite computations have to be taken into account, which can be done by ω -regular expressions [40].

In the following, we demonstrate that, using our language *Quartz*, we are able to write powerful specifications that are as readable as programs. We believe that a programming approach to specification is more appreciated by programmers and engineers. To this end, we will consider a finite fixed set of variables \mathcal{V} . The alphabet $\Sigma_{\mathcal{V}}$ of \mathcal{V} is then simply the powerset of \mathcal{V} , hence, $\Sigma_{\mathcal{V}} := 2^{\mathcal{V}}$. Then, regular expressions are defined as follows:

Definition 2 (Regular Expressions) *The set of general regular expressions $\text{RegExp}(\Sigma)$ over the alphabet Σ is defined as the smallest set that satisfies the following properties (where $\alpha, \beta \in \text{RegExp}(\Sigma)$):*

- $\emptyset \in \text{RegExp}(\Sigma)$ and $1 \in \text{RegExp}(\Sigma)$
- $\vartheta \in \text{RegExp}(\Sigma)$ with $\vartheta \subseteq \Sigma$ (letters)
- $\bar{\alpha} \in \text{RegExp}(\Sigma)$ (complement)
- $\alpha + \beta \in \text{RegExp}(\Sigma)$ (union)
- $\alpha \& \beta \in \text{RegExp}(\Sigma)$ (intersection)
- $\alpha\beta \in \text{RegExp}(\Sigma)$ (concatenation)
- $\alpha^* \in \text{RegExp}(\Sigma)$ (finite iteration)

The above definition of regular expression contains already some syntactic sugar. The semantics of a regular expression r is a set of finite words $\text{Lang}(r) \subseteq \Sigma_{\mathcal{V}}^*$ that is recursively defined as follows (ε is the empty word):

- $\text{Lang}(\emptyset) := \{\}$ and $\text{Lang}(1) = \{\varepsilon\}$
- $\text{Lang}(\vartheta) := \{\vartheta\}$ for every letter $\vartheta \subseteq \Sigma$
- $\text{Lang}(\bar{\alpha}) := \Sigma^* \setminus \text{Lang}(\alpha)$
- $\text{Lang}(\alpha + \beta) := \text{Lang}(\alpha) \cup \text{Lang}(\beta)$
- $\text{Lang}(\alpha \& \beta) = \text{Lang}(\alpha) \cap \text{Lang}(\beta)$
- $\text{Lang}(\alpha\beta) := \{ab \mid a \in \text{Lang}(\alpha), b \in \text{Lang}(\beta)\}$
- $\text{Lang}(\alpha^*) := \bigcup_{i=0}^{\infty} \text{Lang}(\alpha)^i$

It is well-known how to translate a given regular expression r to a finite state automaton \mathcal{A}_r that accepts the language $\text{Lang}(r)$ [8, 10, 18, 21, 31]. It is even possible to compute a symbolic representation of such an automaton in time $O(|r|)$ of length $O(|r|)$ [37]. As *Quartz* programs are also a way to symbolically describe automata, we can directly translate regular expressions to equivalent *Quartz* statements, which is done by the following function \mathbb{Q} :

- $\mathbb{Q}(\emptyset) := \text{assert}(\text{false})$
- $\mathbb{Q}(1) := \text{assert}(\text{true})$
- $\mathbb{Q}(\vartheta) := \text{assert } \varphi_{\vartheta}; \text{pause};$
with $\varphi_{\vartheta} := (\bigwedge_{a \in \vartheta} a) \wedge (\bigwedge_{a \notin \vartheta} \neg a)$ for $\vartheta \subseteq \mathcal{V}$
- $\mathbb{Q}(\alpha \& \beta) := \mathbb{Q}(\alpha) \parallel \mathbb{Q}(\beta)$
- $\mathbb{Q}(\alpha + \beta) := \text{choose } \mathbb{Q}(\alpha) \text{ else } \mathbb{Q}(\beta)$
- $\mathbb{Q}(\alpha\beta) := \mathbb{Q}(\alpha); \mathbb{Q}(\beta)$
- $\mathbb{Q}(\alpha^*) := \text{finloop } \mathbb{Q}(\alpha)$
- $\mathbb{Q}(\alpha^{\omega}) := \text{loop } \mathbb{Q}(\alpha)$

finloop is thereby a loop that only finitely often iterates its body statement, but the number of iterations is nondeterministic. Using nondeterministic *Quartz* statements, it is possible to implement such a loop together with a temporal assertion statement:

```

finloop  $S$  ::= {event  $d$ ;
                choose emit  $d$  else nothing;
                while( $d$ ){
                     $S$ ;
                    assert  $F\ell$ ;
                    choose emit  $d$  else nothing;
                }}
 $\ell$  : pause;

```

Special care has to be taken for instantaneous statements, i.e. for the empty word and the empty language. Moreover, as it is well-known that complement operations are redundant, i.e., every regular expression can be rewritten to eliminate complementation, it follows that we can translate every regular expression to *Quartz*.

As an example, consider the regular expression $(\{a\} + \{a\}\{b\})^{\omega}$ that describes all infinite words over the alphabet $\{\{ \}, \{a\}, \{b\}, \{a, b\}\}$ that does not contain two succeeding occurrences of b . It is translated to the *Quartz* program given in Figure 5.

5.3 Observers for Temporal Logic

Although *Quartz* allows one to use full temporal logic formulas⁵ as specifications, it is sometimes more convenient to write observers with some reachability or fairness constraints. This is the classic approach that is used in many synchronous programming environments. Moreover, it is straightforward to compute for temporal past properties an equivalent deterministic (!) finite

⁵Full temporal logic includes also past time operators, which makes the logic not more powerful, but exponentially more succinct and in general more readable [16, 26, 37].

```

loop
  choose {
    assert(a&!b)
    pause;
  } else {
    assert(a&!b);
    pause;
    assert(!a&b);
    pause;
  }

```

Figure 5: Quartz program for $(\{a\} + \{a\}\{b\})^\omega$

state automaton that can be used as an observer [37]. It is even straightforward to implement temporal past operators by means of equivalent Quartz modules that can then be simply called to ‘execute’ the specification.

```

module PastAlways(event phi, &failure) {
  while(phi) {
    pause;
  }
  emit failure;
}

module PastUntil(event phi,psi, &failure) {
  bool q;
  q = false;
  loop {
    next(q) = psi | phi & q;
    pause;
  }
}

```

It is well-known how to translate LTL formulas φ to equivalent ω -automata \mathcal{A}_φ (see [37] for further references), and even translations to symbolic automata exist that work in linear time (thus producing linear-sized output). We can also use these algorithms to directly generate nondeterministic Quartz statements with assert statements to capture the fairness requirements that are generated by these translations.

6 Summary and Conclusions

In this paper, we have shown that different aspects of a system can be described with a unique programming paradigm. We use the synchronous programming paradigm to describe modular concurrent systems like hardware gate netlists, action-oriented modules, as well as temporal assertions (even extended by ω -regular expressions) in form of simple syntactic sugar. Hence, synchronous programs may not only serve as realization independent descriptions for either hardware or software; they can also be used as alternatives to complex property specification languages like PSL. In addition to increased readability, the approach offers also simulation and execution of these specifications on different architectures.

References

- [1] ACCELLERA. Property specification language reference manual, version 1.1. <http://www.eda.org>, June 2004.
- [2] ACCELLERA. SystemVerilog 3.1a language reference manual. Tech. rep., 2004. Accellera’s Extensions to Verilog.
- [3] BENVENISTE, A., CASPI, P., EDWARDS, S., HALBWACHS, N., LE GUERNIC, P., AND DE SIMONE, R. The synchronous languages twelve years later. *Proc. of the IEEE* 91, 1 (2003), 64–83.
- [4] BERRY, G. A hardware implementation of pure Esterel. In *Workshop on Formal Methods in VLSI Design* (Miami, Florida, January 1991).
- [5] BERRY, G. The Esterel v5_91 language primer, June 2000.
- [6] BERRY, G., AND GONTHIER, G. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming* 19, 2 (1992), 87–152.
- [7] BERRY, G., AND SETHI, R. From regular expressions to deterministic automata. *Theoretical Computer Science* 48, 1 (1986), 117–126.
- [8] BRÜGGEMANN-KLEIN, A. Regular expressions into finite automata. In *Latin American Symposium on Theoretical Informatics (LATIN)* (São Paulo, Brazil, 1992), I. Simon, Ed., vol. 583 of *LNCS*, Springer, pp. 87–98.
- [9] BULTAN, T. Action language: A specification language for model checking reactive systems. In *International Conference on Software Engineering (ICSE)* (University of Limerick, Ireland, 2000), IEEE Computer Society, pp. 335–344.
- [10] CHAMPARNAUD, J.-M. Implicit structures to implement NFA’s from regular expressions. In *Conference on Implementation and Application of Automata (CIAA)* (London, Ontario, Canada, 2001), S. Yu and A. Paun, Eds., vol. 2088 of *LNCS*, Springer, pp. 80–93.
- [11] CHANDY, K., AND MISRA, J. *Parallel Program Design*. Addison-Wesley, Austin, Texas, May 1989.
- [12] DIJKSTRA, E. Guarded commands, nondeterminacy and formal derivation of programs. *Communications of the ACM* 18, 8 (1975), 453–457.
- [13] DIJKSTRA, E. *A Discipline of Programming*. Prentice Hall, 1976.
- [14] DILL, D. The Murphi verification system. In *Conference on Computer Aided Verification (CAV)* (New Brunswick, NJ, USA, July/August 1996), R. Alur and T. Henzinger, Eds., vol. 1102 of *LNCS*, Springer, pp. 390–393.
- [15] FLOYD, R. Assigning meaning to programs. In *Symposia in Applied Mathematics: Mathematical Aspects of Computer Science* (1967), vol. 19, pp. 19–31.
- [16] GABBAY, D., PNUELI, A., SHELAH, S., AND

- STAVI, J. On the temporal analysis of fairness. In *Symposium on Principles of Programming Languages (POPL)* (New York, 1980), ACM, pp. 163–173.
- [17] HALBWACHS, N. *Synchronous programming of reactive systems*. Kluwer, 1993.
- [18] HALBWACHS, N., LAGNIER, F., AND RAYMOND, P. Synchronous observers and the verification of reactive systems. In *Conference on Algebraic Methodology and Software Technology (AMAST)* (Twente, June 1993), Workshops in Computing, Springer, pp. 83–96.
- [19] HENZINGER, T., QADEER, S., AND RAJAMANI, S. You assume, we guarantee: Methodology and case studies. In *Conference on Computer Aided Verification (CAV)* (Vancouver, BC, Canada, 1998), A. Hu and M. Vardi, Eds., vol. 1427 of *LNCS*, Springer, pp. 440–451.
- [20] HOARE, C. An axiomatic basis for computer programming. *Communications of the ACM* 12 (1969), 576–580.
- [21] HOPCROFT, J., AND ULLMAN, J. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.
- [22] JÄRVINEN, H., AND KURKI-SUONIO, R. The DisCo language and temporal logic of actions. Technical Report 11, Tampere University of Technology, Software Systems Laboratory, 1990.
- [23] LAMPORT, L. The temporal logic of actions. Tech. Rep. 79, Digital Equipment Cooperation, 1991.
- [24] LANO, K., AND ANDROUTSOPOULOS, K. Reactive system refinement of distributed systems in B. In *Integrated Formal Methods (IFM)* (York, UK, 1999), K. Araki, A. Galloway, and K. Taguchi, Eds., Springer, pp. 415–434.
- [25] LEUSCHEL, M., AND BUTLER, M. Automatic refinement checking for B. In *International Conference on Formal Engineering Methods (ICFEM)* (Manchester, UK, 2005), K.-K. Lau and R. Banach, Eds., vol. 3785 of *LNCS*, Springer, pp. 345–359.
- [26] MARKEY, N. Temporal logic with past is exponentially more succinct. *Bulletin of the European Association for Theoretical Computer Science* 79 (2003), 122–128.
- [27] MOORBY, P. History of Verilog. *IEEE Design and Test of Computers* (September 1992), 62–63.
- [28] OPEN SYSTEMC INITIATIVE. SystemC 2.0.1 language reference manual. <http://www.systemc.org>, 2003.
- [29] PASAREANU, C., DWYER, M., AND HUTH, M. Assume-guarantee model checking of software: A comparative case study. In *Model Checking Software (SPIN Workshop)* (Toulouse, France, 1999), D. Dams, R. Gerth, S. Leue, and M. Massink, Eds., vol. 1680 of *LNCS*, Springer, pp. 168–183.
- [30] RATIONAL SOFTWARE CORPORATION. UML notation guide - version 1.1, 1997.
- [31] RAYMOND, P., AND ROUX, Y. Describing non-deterministic reactive systems by means of regular expressions. *Electronic Notes in Theoretical Computer Science (ENTCS)* 65, 5 (2002). Workshop on Synchronous Languages, Applications, and Programming (SLAP).
- [32] REETZ, R., SCHNEIDER, K., AND KROPP, T. Formal specification in VHDL for formal hardware verification. In *Design, Automation and Test in Europe (DATE)* (February 1998), IEEE Computer Society.
- [33] ROCHETEAU, F., AND HALBWACHS, N. Pollux, a Lustre-based hardware design environment. In *Conference on Algorithms and Parallel VLSI Architectures II* (Chateau de Bonas, 1991), P. Quinton and Y. Robert, Eds.
- [34] SCHNEIDER, K. A verified hardware synthesis for Esterel. In *Workshop on Distributed and Parallel Embedded Systems (DIPES)* (Schloß Ehringerfeld, Germany, 2000), F. Rammig, Ed., Kluwer, pp. 205–214.
- [35] SCHNEIDER, K. Embedding imperative synchronous languages in interactive theorem provers. In *Conference on Application of Concurrency to System Design (ACSD)* (Newcastle upon Tyne, UK, June 2001), IEEE Computer Society, pp. 143–156.
- [36] SCHNEIDER, K. Proving the equivalence of microstep and macrostep semantics. In *Higher Order Logic Theorem Proving and its Applications (TPHOL)* (Hampton, VA, USA, 2002), V. Carreño, C. Muñoz, and S. Tahar, Eds., vol. 2410 of *LNCS*, Springer, pp. 314–331.
- [37] SCHNEIDER, K. *Verification of Reactive Systems – Formal Methods and Algorithms*. Texts in Theoretical Computer Science (EATCS Series). Springer, 2003.
- [38] SCHNEIDER, K. The synchronous programming language Quartz. Internal Report (to appear), Department of Computer Science, University of Kaiserslautern, 2006.
- [39] SCHNEIDER, K., BRANDT, J., AND SCHUELE, T. A verified compiler for synchronous programs with local declarations. *Electronic Notes in Theoretical Computer Science (ENTCS)* 153, 4 (2006), 71–97.
- [40] THOMAS, W. *Automata on Infinite Objects*, vol. B. Elsevier, 1990, ch. Automata on Infinite Objects, pp. 133–191.
- [41] *IEEE Standard VHDL Language Reference Manual*. New York, USA, June 1993. ANSI/IEEE Std 1076-1993.
- [42] ZULKERNINE, M., AND SEVIORA, R. Assume-guarantee algorithms for automatic detection of software failures. In *Integrated Formal Methods (IFM)* (Turku, Finland, 2002), M. Butler, L. Petre, and K. Sere, Eds., vol. 2335 of *LNCS*, Springer, pp. 89–108.