Chapter 5
A New Property Language for the Specification of Hardware-Dependent Embedded System Software

Binghao Bao, Carlos Villarraga, Bernard Schmidt, Dominik Stoffel, and Wolfgang Kunz

Abstract This work introduces a new property language for describing the behaviour of low-level hardware-dependent software. The design of the language is motivated by the industrial success of property languages for hardware verification by simulation and formal techniques. The new language is constructed to concisely capture the timed behaviour of the interactions between software and hardware by means of sequences. In this chapter we present how the proposed verification language can be used to perform formal verification based on a computational model called *program netlist*. We show how the sequence model of the language is synthesised and combined with the program netlist so that a unified formula for a decision procedure, e.g., a SAT solver, can be constructed. Furthermore, a method for coverage analysis of property sets is introduced. The coverage criterion we propose determines whether or not the property set completely describes the input/output functional behaviour of a program. The work presents a case study showing how to use the proposed property language in order to specify an industrial implementation of a LIN (Local Interconnect Network) bus driver.

5.1 Introduction

Besides continuous advances in methods and algorithms for formal property checking of hardware designs, also the languages for formulating properties have played an important role for the adoption of formal verification techniques in industry in the last years. For instance, SystemVerilog Assertions (SVA) [15] and Property Specification Language (PSL) [1] allow to concisely specify the behaviour of the hardware, which is typically described at register transfer level (RTL).

B. Bao • C. Villarraga (✉) • B. Schmidt • D. Stoffel • W. Kunz
University of Kaiserslautern, Kaiserlautern, Germany
e-mail: bao@eit.uni-kl.de; villarraga@eit.uni-kl.de; schmidt@eit.uni-kl.de; stoffel@eit.uni-kl.de; kunz@eit.uni-kl.de

F. Oppenheimer, J.L. Medina Pasaje (eds.), *Languages, Design Methods, and Tools for Electronic System Design*, Lecture Notes in Electrical Engineering 361,
DOI 10.1007/978-3-319-24457-0_5

While being founded in a strictly defined mathematical framework, these property languages include various syntactic enhancements offering a natural and easy way to capture temporal behaviours of the design. Current commercial technology allows for checking assertions using simulation or formal verification engines.

On the other side, for the case of embedded software (SW) there is an increasing necessity of integrating formal verification also to the verification flows used in industry. In this work we focus on the verification of hardware-dependent software which is the part of the software in an embedded system that interacts directly with the surrounding hardware (HW). There are a number of reasons why we focus on this kind of software. Hardware-dependent software is a critical component in embedded systems since all other software layers (e.g., the operating system, application software, etc.) are built on top of it. Additionally, hardware-dependent software in embedded systems performs control-intensive tasks with complex interactions with the hardware and with other software layers, making development error prone and systems difficult to test. Because of the reactive behaviour of HW/SW interaction, specification languages and validation methods as they have been developed for application-level software are in many cases not suitable. This work proposes a new property language facilitating the specification of hardware-dependent software behaviour in embedded systems. Similar to property languages used for hardware, this new language allows to capture the reactive behaviour of the hardware-dependent software by using sequences describing series of input/output operations performed by the software at its interfaces.

The property language proposed here can be employed for simulation or verification purposes. However, in this work we present particularly how this language can be used in conjunction with a computational model called *program netlist* [14] in order to perform formal property checking. A program netlist is a combinational Boolean model representing compactly all possible execution paths of a software program. It is built using hardware models of the machine instructions executed in the program, and is therefore suitable for representing hardware-dependent software. For generating a program netlist, path-oriented techniques related to *symbolic execution* [5] are used. The actual flow of program control is modelled by additional logic added to the program netlist that enhances the efficiency of SAT reasoning on program segments and entire paths.

Unlike methods based on symbolic execution in which properties are proven by traversing explicitly the possible execution paths of a given program, in this work, we adopt the approach of [14] which employs a SAT solver in order to perform this traversal. A SAT proof benefits from the control logic in the program netlist by being able to focus on the execution paths being important for the particular problem instance and to prune at once entire execution paths that are not relevant. The effectiveness of this approach has been shown in [14].

In order to use a SAT solver for path traversal it is necessary to create a combined model containing the logic for the property and the program netlist such that the SAT solver has “the global view on the verification problem” (instead of having only the view of the problem for individual execution paths). For the global view, a model of the input/output sequences of the software is synthesised and integrated to the model.

Since formal verification examines every possible input scenario, the usual coverage criteria evaluating the quality of test cases for software are not suitable for formal property checking. In case of software property checking, verification engineers face the same problem as engineers in hardware verification: "Does my property set cover every aspect of the design?" A number of methods for coverage analysis attempting to prove that a property set is "complete" have been successfully applied to hardware designs [4, 6, 9, 11]. In these approaches a set of safety properties is called a *complete specification* or simply *complete* if it uniquely describes the behaviour of a design. More precisely, these methods prove the completeness of a property set by means of checking to what extent the property set *uniquely* specifies the input and output behaviour of a hardware design. Based on these results, in this work we develop a method for proving the completeness of software properties specified in the software property language presented in this chapter. Such a completeness check is of particular importance for software property sets because a complete set of properties can, at least in principle, fully replace classical software tests. A typical source of error when writing software is that the programmer simply forgets to treat certain input sequences in his program, causing undefined behaviour when these inputs occur at runtime. Also, a verification engineer may forget to specify tests (or, in our case, properties) for such missing input sequences so that the bug can escape verification. The completeness check presented in this work removes such verification gaps. Because of similarity between hardware and low-level software, our method for proving the completeness of software property sets checks whether every input/output sequence is specified by a property in the set. The checking algorithm leverages the property language presented in this chapter, which allows for referring to the interfaces of the software program in order to describe its reactive and sequential behaviour.

As of today, there are a number of different approaches for formalising properties of embedded software. *Run-time assertions* are being used widely for testing [16] and formal verification [7, 19], of embedded software. For example, high-level programming languages like C provide the *assert()* statement for specifying predicates over the values of program variables. The main use of run-time assertions is to describe properties that are valid locally. More specifically, this kind of property is evaluated only when a program run reaches the location where the involved *assert()* statement has been placed. While run-time assertions have the advantage that the user is not required to learn a new language in order to specify properties, their main limitation is in what can be expressed. For the case of application software or for simple transformational code run-time assertions may be sufficient; however, for hardware-dependent software it is necessary to be able to describe reactive behaviour, relating to the inputs, outputs and states of the software and hardware *at different points in time*. For specifying temporal behaviour, temporal logics such as CTL and LTL [8, 12] can be used. There exist verification tools such as [10, 13] that accept temporal formulas directly as PSL. However, although CTL and LTL are powerful in formulating temporal relationships, they are hard to understand and use in practice.

Other tools such as [2, 3], in a similar way, employ automata in order to temporal specify properties. The use of automata can be convenient in many cases since they are easier to understand by a designer or verification engineer than temporal formulas in CTL or LTL. However, except for simple cases, the process of modelling a property using an automaton is cumbersome and error prone.

Different to the approaches mentioned previously, in this work we present a new verification language for hardware-dependent embedded software that allows to specify the temporal behaviour of interactions between software and hardware. The proposed language is intuitive and easy to use for the verification engineer. It allows to refer to the interfaces of the software and to describe explicitly the sequences of input/output operations at these interfaces. It adopts many syntactic elements from the C language, which makes learning the new language easy for software engineers. To the best of our knowledge, this is the first work on a property language for hardware-dependent embedded software with the characteristics mentioned above.

The remainder of the chapter is organised as follows. Section 5.2 reviews the computational model used in this work. The software PSL is described in Sect. 5.3, and it is applied to specifying properties for a LIN driver implemented in software, as presented in Sect. 5.5. In addition, a method for evaluating the completeness of property sets is described in Sect. 5.4. A conclusion and outline of future work are given in Sect. 5.6.

5.2 Low-Level Software Model

In this work we show how the language to be presented in Sect. 5.3 can be used together with a model for hardware-dependent low-level software in embedded systems, called *program netlist*, in order to perform formal property checking. We first review basic characteristics of the program netlist. A complete description of this model and its generation can be found in [14].

A program netlist is a combinational circuit that compactly represents the software that is executed on the underlying hardware. In order to generate a program netlist, the control flow graph (CFG) is extracted from a low-level description of the software program, like assembly or machine code. Every node in the CFG represents an instruction of the program and the associated *program state* (PS). The PS includes the contents of data memory associated with the variables used in the program, and the *architecture state* (AS), defining the state of the processor's registers that are visible to the programmer. An edge between two CFG nodes indicates a possible execution from one instruction to another one. An additional Boolean signal called *active* is attached to PS in order to model the control flow of the program. This signal is propagated alongside the nodes in the program netlist and helps the SAT solver to efficiently explore the possible execution paths of the program. The *active* signal, when set to 1, indicates that a given node (instruction) belongs to the active execution path. In the case that a node has more than one successor (e.g., nodes related to jump/branch instructions), exactly one branch is active at any time.

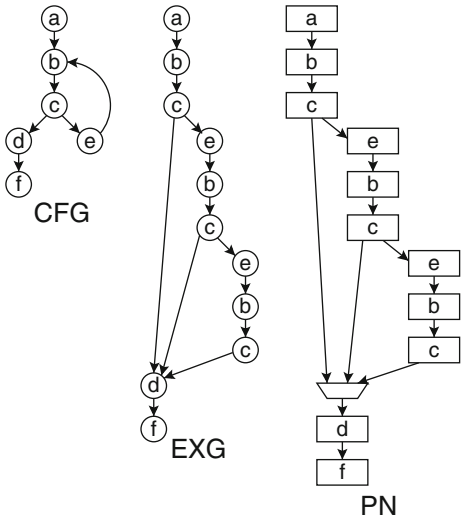

Fig. 5.1 Generating the program netlist (PN)

The CFG is fully unrolled into an *execution graph* (EXG). An EXG is a directed acyclic graph containing all possible execution paths of the program. An execution path always begins at a start state of the program and ends at an end state. The CFG is unrolled by unwinding the loops of the program. Figure 5.1 illustrates an example of unrolling. In order to reduce the complexity of the model, only branches that are part of at least one possible execution path are processed. A SAT solver can be used to identify such branches. Unrolling ends when all *active* branches have been processed and the end of the program has been reached. In addition, in order to minimise the size of the model, nodes belonging to identical program locations are merged. In this manner an EXG is obtained in which a single node may be shared by different execution paths. Merging is only allowed if it does not insert loops in the EXG.

A program netlist is then obtained from the EXG by replacing every node by its corresponding *instruction cell*. An instruction cell is a piece of combinational logic circuitry describing the functional behaviour of an ISA instruction according to the specific CPU architecture at hand. Consecutive instruction cells are connected by buses representing the program state.

A kind of instruction that is especially relevant to this work are load/store instructions which are used to communicate with the program's environment, e.g., the hardware periphery or other software layers. Instruction cells corresponding to such kind of instructions are equipped with additional input and output ports as shown in Fig. 5.2. These ports are called *pdata*, *ploc* and *pact* and represent respectively, the data value, the accessed location and the *active* flag indicating the activeness of the related instruction cell. Depending on whether the instruction cell reads or writes, *pdata* is an input or output signal. These three signals of an instruction cell constitute an *access port* for I/O memory locations.

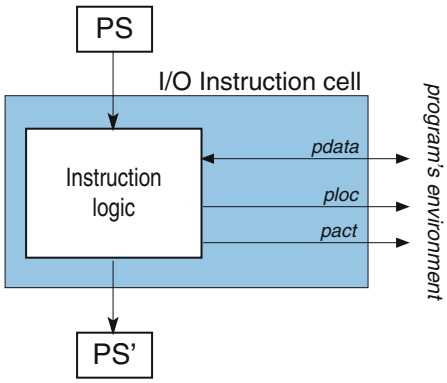

Fig. 5.2 Instruction cell with ports for accessing the environment

In the sequel, we use the term *program location* to indicate a memory location storing an instruction, and the term *memory location* to indicate an address corresponding to a location of the hardware periphery or a memory variable.

In the program netlist, instructions that access data memory require additional constraints so that the behaviour of the data memory is also modelled [18]. Therefore, for each instruction cell that reads from data memory there is a multiplexer structure that selects the last valid value written to the memory location being read by the instruction. In the case that a program depends on external events, e.g., by means of shared variables/channels, additional access ports of the respective instruction cell are left open or unconstrained as shown in Fig. 5.2. These access ports serve as the interfaces of the program, as will be further explained in the next section.

5.3 Software Property Language

This chapter presents how the interaction between hardware and software can be described in terms of I/O sequences and how the model of the sequences can be synthesised and combined with the underlying model of the software in order to perform formal property checking. As explained earlier, this is necessary in order to capture the reactive behaviour exhibited by hardware-dependent software. An additional advantage is that a model of the sequences allows to map the elements of the language to the elements of the underlying software model in a straightforward way. Subsequently, we show how a property language can be developed in terms of such sequences. The current working name for this language is RSPL (Reactive Software Property Language).

In the following, we introduce the main syntactic elements of RSPL. Since the programming language C is widely used for embedded software, our property language adopts many operators and syntax elements from C. For example, RSPL inherits from C the standard arithmetic, Boolean and comparison operators.

```
⟨Name of variable⟩'read
⟨Name of variable⟩'write
```

Fig. 5.3 Read/write attributes

5.3.1 Interfaces of a Hardware-Dependent Program

A property language for hardware-dependent software needs to provide a means for referring to the interfaces of a given program. In contrast to hardware description languages, software programs in high-level languages such as C do not explicitly capture their interfaces in a separate entity. For hardware-dependent software the elements of the interface correspond a set of addresses identifying, for example, registers inside hardware peripherals or shared memory locations used for communication with the operating system or with the application code. In view of the program netlist, such interface elements are modelled by means of access ports belonging to input/output instruction cells as explained in Sect. 5.2. In RSPL each of these addresses is assigned a name. In case of compiled machine code these names can be automatically obtained from the symbol table. Otherwise variable names can be defined manually by the user to enhance readability of the verification code. In order to distinguish the action of reading a variable (as input) from the action of writing a variable (as output), two variable attributes are introduced to the property language, namely *read* and *write*, as depicted in Fig. 5.3. This is the basis for referring to all input/output operations. Note that an address can be read and written several times. How this can be handled by the property language and how the verification engineer can refer to the different read and write instances is described below in Sect. 5.3.2.

There are also cases in which it is necessary to refer to the programmer-visible registers, for example when separately verifying a subroutine of a software driver. In such cases the content of a register can be expressed using the following syntax.

```
$⟨Name of register⟩'start
$⟨Name of register⟩'end
```

The attributes "*start*" and "*end*" indicate the start and the end of the program, respectively.

5.3.2 Sequences of Variables

The *sequence* is the key concept of our language; it is inspired by sequences in hardware property languages like SVA [15]. A sequence in SVA is constructed using the delay operator # which specifies the relative clock cycles (delay) between two events. However, we cannot directly import the semantics of sequences from SVA, since a sequence in SVA is defined over cycles which are relative to a global time reference such as a hardware clock. Models used for software (and in particular the program netlist) are not accurate with respect to hardware clock cycles, but

```
⟨Name of variable⟩'read #⟨n⟩
⟨Name of variable⟩'write #⟨n⟩
```

Fig. 5.4 Element accessor

rather instruction-accurate. Therefore, sequences are defined relative to the ordering of instruction executions. As illustrated in Fig. 5.4 we provide users with a way to define the individual elements of a sequence. Several such elements may be combined using Boolean operators in order to form sequences. We call the symbol # the *element accessor* for sequences and the natural number n represents the n-th element of a sequence. Since not every instruction accesses the interface of the program, the element n is the n-th occurrence of the associated interface variable along an execution path of a program (as opposed to the n-th instruction along that path). A software tool evaluating properties written in our language needs to map the elements of a sequence to the respective access ports in the program netlist. Because of the merging mechanism used to generate a program netlist, an input/output instruction cell can belong to several different execution paths. In other words, along an execution path an access port might be the i-th sequence element of a variable, whereas along another path, it may correspond to the j-th ($j \neq i$) sequence element of the same variable.

In the following we present an algorithm to map elements of sequences to the corresponding access ports in the program netlist. This algorithm is the basis for building the property logic for a SAT-based proof engine. To simplify the presentation, in the sequel, we only consider the memory locations of input/output variables, not their symbol names, since this relationship can be established easily through the symbol table. We use the term *memory location* to refer to both, an input or output variable as defined in Sect. 5.3.1, if the use is clear from the context.

The first step in the algorithm is a topological sorting of the nodes in the execution graph. We assign every node a unique index m, with $m \in \mathbb{N}$, so that along every execution path the (instruction) node indexed with i is executed earlier than the node indexed with j, if $i < j$. In the sequel, we refer to a node by its index in the topological order. Each memory location Loc_k is associated with a set of nodes accessing this location, i.e., $W = \{i_1, i_2, \ldots, i_{|W|}\}$ with $i_j < i_{j+1}$ and $1 \leq j < |W|$. The access port AP_{i_j} for a node i_j is composed of $pdata_{i_j}$, $pact_{i_j}$, and $ploc_{i_j} = Loc_k$, corresponding to the signal names in Fig. 5.2.

Given a memory location Loc_k, to map the n-th element ($1 \leq n \leq |W|$) of Loc_k's sequence to an access port AP_{i_j}, the challenging task is to find the index i_j of the node related to this element. The function *comp_index(n)*, depicted in pseudo-code notation in Algorithm 2, performs this task. It is generated for every memory location Loc_k. Function *port_mapping()* of Algorithm 3 is based on *comp_index()*. It connects the n-th element of the sequence to an access port in the program netlist.

The formulation of function *comp_index()* is based on the fact that at any time exactly one execution path of a program is active. An active path is characterised by the nodes in the program netlist whose *active* flags are asserted. In summary,

Algorithm 2 Compute index of the node associated with n-th sequence element

```
1: function COMP_INDEX(n)
2:     if n = 1 then
3:         if pact_{i_1} = true then
4:             return i_1
5:         else if pact_{i_2} = true then
6:             return i_2
7:             ...

8:         else if pact_{i_|W|} = true then
9:             return i_|W|
10:        else
11:            return 0
12:        end if
13:    else
14:        if pact_{i_n} = true ∧ compindex(n − 1) < i_n then
15:            return i_n
16:        else if pact_{i_n+1} = true ∧ compindex(n − 1) < i_n+1 then
17:            return i_n+1
18:            ...

19:        else if pact_{i_|W|} = true ∧ compindex(n − 1) < i_|W| then
20:            return i_|W|
21:        else
22:            return 0
23:        end if
24:    end if
25: end function
```

Algorithm 3 Map sequence to access port

```
1: function PORT_MAPPING(n)
2:     if compindex(n) = i_1 then
3:         return pdata_{i_1}
4:     else if compindex(n) = i_2 then
5:         return pdata_{i_2}
6:         ...

7:     else if compindex(n) = i_|W| then
8:         return pdata_{i_|W|}
9:     else
10:        return UNDEFINED
11:    end if
12: end function
```

comp_index() works as follows: To determine the n-th element of a sequence along any execution path, we first check the *active* flag of the node with index i_n, this is the very first node that could be the n-th element of a sequence. We also examine whether the $(n-1)$-th element along that path exists already, by checking whether the index of the node associated with $(n-1)$-th element is smaller than the index of the current node. If both conditions are met, then the n-th element of the sequence is

known to exist and the respective index can be returned. Otherwise we move on to the next candidate until a node related to the element we search is found or does not exist. With the *comp_index* function we can test whether an element of a sequence exists on a given path (by testing whether the result of *comp_index* is zero). The function is also used for verifying the execution order (cf. Sect. 5.3.3) of sequence elements that are related to different memory locations (variables).

With the ability of obtaining the index i_j of the node related to the n-th element of a sequence, it is straightforward to map the n-th element of the sequence to the access port AP_{i_j}. Function *port_mapping* depicted in Algorithm 3 performs this task.

In the remainder of this paper, for simplicity, we use the term *variable* for both, an element of an input/output sequence, or the state of a register at the start node/end node of the program.

5.3.3 Execution Order

Besides being able to relate software accesses to the same location at different points in time, in many cases it is also important to specify a temporal order of accesses to different memory locations. For instance, in order to issue a new transaction, a peripheral device may require that its driver first write the configuration/data register of the device at memory location Loc_1, and then set the start flag at memory location Loc_2; not maintaining this order could result in undefined behaviour of the device. Obviously, the property specification "**Loc_1'write#1 == Config_Data && Loc_2'write#1 == Start**" is insufficient for this requirement, since this statement does not define which of the two accesses, "**Loc_1'write#1**" and "**Loc_2'write#1**" is to be executed first by the software driver.

In RSPL, temporal ordering of accesses to different locations can be specified using the *execution order* section in a property. A user may specify an execution order between an input and an output, two inputs related to two different memory locations and two outputs related to two different memory locations. Checking the execution order is implemented by comparing the results of the functions *comp_index* for the respective sequence elements. Taking the example from above, if the returned value of the function *comp_index* for "**Loc_1'write#1**" is smaller than the returned value of this function for "**Loc_2'write#1**", then "**Loc_1'write#1**" is executed first. The syntax definition and an example of an *execution_order* specification are given in Fig. 5.5.

5.3.4 Safety and Liveness Properties

So far, we introduced the basic concepts and building blocks of the property language. In this section, we will present how to use them to build a property, and we will discuss what kinds of properties we can specify.

```
// execution order definition
execution_order:
⟨sequence element⟩ > ⟨sequence element⟩;

// examples
execution_order:
config'write#1 > Start'write#1;
config'write#2 > Start'write#1;
config'read#1 > config'write#1;
```

Fig. 5.5 Execution order

```
// Example of property

property example1;

// assumption part
assume:
$R5'start == 2;

// prove part
prove:
data_out'write#1 == data_in'read#1 + 4;
$R4'end == 0;

// execution order part
execution_order:
data_out'write#1 > data_in'read#1;
endproperty

// safety property
inst1: always example1;
// liveness property
inst2: eventually example1;
```

Fig. 5.6 Safety- /liveness-property

A property begins with the keyword *property*, followed by a valid identifier. The general structure of the property body follows an assumption/guarantee style. The body consists of two optional sections, *execution_order* and *assume*, and one mandatory section, *prove*. The *assume* part specifies the circumstances under which the assertion as specified in the *execution_order* and *prove* parts is to be checked. If we denote the assumption part by a predicate a, the prove part by c and the execution order part by o. Then a property p is translated to a Boolean formula $p := a \rightarrow (c \wedge o)$.

Given a property p, we can instruct the property checker to check it as a safety property or as a liveness property. As illustrated in Fig. 5.6, a safety property is indicated by the keyword "*always*", and a liveness property is indicated by the keyword "*eventually*" .

The semantic of "safety/liveness" is defined by evaluating the execution paths of a program. In contrast to Kripke models used in LTL or CTL model checking, the program netlist contains a finite number of paths of finite length. This greatly simplifies the evaluation of safety and liveness properties. A safety property "*always p*" means that on every execution path (from a start state to an end state of the program), the property p holds. This is similar to the LTL property $G\,p$, however, applied to a finite-length path. A liveness property "*eventually p*" means that there exists at least one execution path on which the property p holds. This is similar to the meaning of the CTL property EGp. It is straightforward to check the safety property using a SAT solver. In order to check a liveness property, we check the safety property "*always* $\neg p$". In case this property holds, we may conclude that the corresponding liveness property fails.

5.3.5 Syntax Extensions

With the language elements presented so far, we are able to capture the reactive behaviour of the software programs considered by our technique. We now present a number of extensions to the syntax that do not increase expressiveness but make property notation easier and more compact.

In the following, a variable *var* represents either an input (with attribute *read*) or an output (with attribute *write*). The symbol $\bowtie$ represents an arbitrary comparison operator, and *expr* represents any valid expression at either side of a comparison operator. The accessors depicted in Fig. 5.7 can be used to access a range of sequence elements related to a variable *var*. Every element is compared with the expression *expr*; if all comparison operations evaluate to true, then the result of this statement is true, otherwise it evaluates to false.

The dual case is handled by the accessor depicted in Fig. 5.8: it evaluates to true if the comparison operations return true at least N times in sequence.

```
var#[m:n] ⋈ expr :=
(var#m ⋈ expr) && (var#(m+1) ⋈ expr) &&
... && (var#n ⋈ expr)

with  m ≤ n and m,n ∈ ℕ
```

Fig. 5.7 Access a range of elements (universal)

```
var#EN[m:n] ⋈ expr :=

(var#m ⋈ expr) || (var#(m+1) ⋈ expr) ||
... || (var#n ⋈ expr)

with  m ≤ n, 0 < N < m − n, K ≥ N, and m,n,N ∈ ℕ,
where  K represents the number of terms evaluated true
```

Fig. 5.8 Access a range of elements (existential)

The function **exists** tests whether a sequence element exists on an execution path. In a safety property **exists** (**var#1**) checks whether **var#1** exists for every execution path, whereas in a liveness property it checks whether this element can be generated/consumed at least once during the execution of the program. This function can also be used to check whether the *assume* part of a property always evaluates to false due to non-existing sequence elements. Again, a SAT-based property checker can implement these checks using *comp_index()*.

5.4 Completeness of Property Sets

The completeness of a set of properties for a hardware design can be proven by using design-independent approaches such as [4, 6]. Our approach strongly relates to [4], which is explained in more detail in Sect. III-D of [17]. This method proves that two models of a design satisfying a set of properties $\{p_i\}$ are sequentially equivalent in terms of input/output sequences. What input and output values are considered and at what time points (clock cycles) is specified by the user in terms of so-called *determination conditions*. For example, a "data" signal needs to be uniquely determined by the design whenever a corresponding "valid" signal is asserted. A complete property set fulfils this determination condition if every property specifies the expected "data" value at the time points when the "valid" signal becomes asserted. Note that a property set can be checked for completeness independently of any design implementation for which this set of properties holds, because only relationships between signal names in the properties are checked. This idea can also be transferred to software properties written in RSPL and results in the following definition for completeness of a set of properties:

Definition 5.1. A set of RSPL properties is called complete, iff

1. there exists a property with a matching *assume* part for every possible input sequence applied to the program, and
2. every property uniquely specifies every output sequence produced by the program under the input sequences specified in the *assume* part.

Testing the two conditions of Definition 5.1 can be directly implemented in two checks called *Determination Test* and *Case Split Test*.

5.4.1 Determination Test

Unlike hardware that generates output sequences for every time point (clock cycle), the low-level software may produce output sequences of varying length, depending on the input sequences applied to the program. For instance, depending on configuration data given by the program's environment, a software driver may

perform burst write operations with 2 or 4 beats of data transfer, causing sequences with 2 or 4 elements, respectively. If we use design-independent methods, the completeness checker needs to know how many elements of sequences should every property at least specify. Denoting these values for every output in every property is tedious and error-prone. Therefore, for checking completeness of RSPL property sets, we give up on the design independence of a completeness criterion. Instead, we make use of the software model to determine how long the checked sequences are on each program execution.

In order to ensure that every output signal is uniquely determined by the property set, we perform two steps. First, we ensure that every property describes every element of all output sequences that are produced under all matching input sequences specified in the assumption part of the property. We solve this problem with the help of the design under verification, M, and the *comp_index* function. For simplicity, in the following we assume that the properties are written in a *causal* form, expressed as an implication between a cause (property assumption) and an effect (property commitment). This causal form is given if the input sequences are specified in the *assume* part of the property and the expected output behaviour is specified in the *prove* and *execution_order* parts. Given a property $p := a \rightarrow (c \wedge o)$, by syntactic analysis, we can identify the maximum sequence length k of any output sequence specified in the property. Then we check whether k is the maximum element generated by the software model M under the assumption a. For this purpose, we resort to the *comp_index(k+1)* function with respect to the assumption a: If this function returns a non-zero value, it means that the program can output a sequence with at least $k+1$ elements. The property under consideration does not specify this output sequence element, hence, we have detected a verification gap. Similarly, if a property does not at all mention some output produced by the program, we can detect this by checking for non-zero return of *comp_index(1)* for that output. A list of all possible outputs of a program can be easily obtained when synthesising the model of the program.

Once we have certified that every property describes every possible element of all output sequences, we check whether these output sequences are determined *uniquely*, i.e., whether, along any execution path, the property specifies exactly one value for every element of the output sequence. This can be done for every property independently of the software model. Let p be a property containing a set of signals $\{v_i\}$ (composed of a set of inputs $\{x_j : j \in \mathbb{N}, j \leq m\}$ and a set of outputs $\{o_n : n \in \mathbb{N}, n \leq l\}$) and corresponding sequence elements $\{v_i^{k_{v_i}} : k_{v_i} \in \mathbb{N}_{\neq 0}, k_{v_i} \leq t_{v_i}\}$. We create a copy p' of p by considering a copied set $\{v_i'\}$ of the variables appearing in the property and imposing the property on the copied variable set. The property p determines the outputs $\{o_n\}$ uniquely, iff the following formula is a tautology

$$(p \wedge p' \wedge \bigwedge_{j=0}^{m} \bigwedge_{k_{x_j}=1}^{t_{x_j}} (x_j^{k_{x_j}} = x_j'^{k_{x_j}})) \rightarrow \bigwedge_{n=0}^{l} \bigwedge_{k_{o_n}=1}^{t_{o_n}} (o_n^{k_{o_n}} = o_n'^{k_{o_n}})$$

5.4.2 Case Split Test

The case split test checks whether the property set covers every possible input sequence. Given a set of properties p_i with their respective assumption parts a_i, the case split test is conducted by proving that the formula $\bigvee_i a_i$ is a tautology.

5.4.3 Completeness Criterion

Theorem 5.1. *If and only if a set of RSPL properties $\{p_i\}$ passes both the Determination Test and the Case Split Test, then the property set is complete according to Definition 5.1.*

Proof. The theorem is true by Definition 5.1 because the Case Split Test checks for fulfilment of condition 1 of Definition 5.1 and the Determination Test checks for fulfilment of condition 2. □

5.5 Case Study

The property language developed in this work has been successfully applied to specifying properties for an industrial software driver for a *LIN* master node. The software was developed by Infineon Technology AG. Note that the focus of the work is on the challenges of specifying complete sets of properties for this type of software, not on the proving techniques. The properties shown in this case study have already been proven earlier [14], based on a manual construction of checker automata which were added to a program netlist model of the software. In this work we present the formulation of these properties in RSPL.

The hardware peripheral controlled by the software driver is a UART (Universal Asynchronous Receiver/Transmitter), connected to the physical LIN bus lines. A LIN bus is composed of one master node and several slave nodes. Data is transmitted on the LIN bus in so-called *frames*. A frame is composed of several fields: a header, up to 8 bytes of data, and a checksum. The master node is responsible for sending the header field which is composed of a *break* field indicating the start of new frame, a *sync byte* field used for synchronisation, and an *identifier* (ID) field. Slave nodes evaluate the identifier field and, if there is a match, then the corresponding slave node either sends or receives data. The LIN driver code under consideration implements a master node. It supports six fixed-valued IDs. It can send or receive 2, 4 or 8 bytes of data for each of the six IDs. Data is communicated with the application software through shared memory locations that serve as the interface of the LIN driver.

We now consider a first property in Fig. 5.9 specifying the transmission of a frame, according to the protocol specification for the LIN bus. For reasons of space,

```
property lin_master_transmits_2_bytes;

assume:
s_id'read#1 == C_ID0;

prove:
// master task
uart'write#1 == C_BREAK;
uart'write#2 == C_SYNC;
uart'write#3 == C_ID0;

// slave task

// an example for undetermined uart'write#4 would be
// uart'write#4 == 1 || uart'write#4 == 0;
uart'write#4 == data1'read#1;
uart'write#5 == data2'read#1;
uart'write#6 == CHECKSUM;

execution_order:
data1'read#1 > uart'write#4;
data2'read#1 > uart'write#5;
endproperty;

// safety property
inst1: always lin_master_transmits_2_bytes;
```

Fig. 5.9 LIN_TX_Frame_2_Bytes

we show only the case for data length of 2 bytes. Furthermore, for readability, we use the names of variables and constants instead of their memory addresses. Variables *data1* and *data2* store the payload data provided by the application software. The *s_id* are shared variables storing the ID that needs to be transferred to the slave task. The symbol *uart* refers to the Tx/Rx buffer of the UART.

In the following, the prefix "C_" indicates a constant value. *C_ID0* identifies a 2-byte transmission. *CHECKSUM* abstracts the "checksum" computation.

We also need to define an *execution_order* section in the property in order to specify that the data must be available before it is transmitted.

Note that a program that does not support *C_ID0* at all may nevertheless fulfil the property in Fig. 5.9. We therefore need to check the liveness property in Fig. 5.10 in order to make sure that at some point in time a *C_ID* = *frame* is indeed sent to the UART.

Figure 5.11 shows the use of the *exists* function in a property that checks whether the driver is capable of transmitting 8-byte data frames.

Obviously, the safety property in Fig. 5.9 does not completely specify the entire program. It specifies only the case that the LIN master transmits 2 bytes of data to a slave. The case split test presented in Sect. 5.4 identifies the missing cases by checking whether there exists a corresponding property for every value of *s_id*.

```
property lin_test_C_ID0;

prove:
uart'write#3 == C_ID0;
endproperty;

// liveness property
inst2: eventually lin_test_C_ID0;
```

Fig. 5.10 LIN liveness

```
property lin_test_support_8_bytes;

prove:
exists(uart'write#11);
endproperty;

inst3: eventually lin_test_support_8_bytes;
```

Fig. 5.11 LIN liveness 2

The comments section of this property shows an example of a failing determination test for a sequence element, where this statement states that the value of "**uart'write#4**" could be either 0 or 1. Thus, the value of this variable is not *unique*: Suppose that "**uart'write#4**" is a Boolean variable, then the expression in the statement evaluates to *true*. Hence, the property proves nothing about this variable.

5.6 Conclusion

In this chapter we presented the concept and the basic framework of a software property language for reactive low-level embedded software. The language is based on a computational model for this type of software, called *program netlist*. The language allows to easily express the I/O behaviour of the software by using temporal sequences. Furthermore, properties can be synthesised and combined with the program netlist into a single model allowing to perform formal verification. Taking advantage of the temporal description we defined a completeness criterion for a set of properties. We showed how to use the elements of the language to formally and completely specify a LIN master node.

The future development of the proposed property language will include extensions in order to support compositional verification for cases where the overall verification of a program needs to be partitioned to improve scalability. Additionally, the concept of *functions* or *macros* will be introduced for structuring and re-using verification code.

References

1. Accellera Organization Inc. Property Specification Language - Reference Manual, Version 1.1. http://www.eda.org/vfv/docs/PSL-v1.1.pdf, June 2004
2. Ball, T., Rajamani, S.K.: Slic: A specification language for interface checking (of c). Technical Report MSR-TR-2001-21, Microsoft Research, Jan (2002)
3. Beyer, D., Chlipala, A., Henzinger, T., Jhala, R., Majumdar, R.: The blast query language for software verification. In: Giacobazzi, R. (ed.) Static Analysis. Lecture Notes in Computer Science, vol. 3148, pp. 2–18. Springer, Berlin (2004)
4. Bormann, J., Busch, H.: Verfahren zur Bestimmung der Güte einer Menge von Eigenschaften (Method for determining the quality of a set of properties). European Patent Application, Publication Number EP1764715, 09 (2005)
5. Cadar, C., Sen, K.: Symbolic execution for software testing: three decades later. Commun. ACM **56**(2), 82–90 (2013)
6. Claessen, K.: A coverage analysis for safety property lists. In: Proceedings of International Conference on Formal Methods in Computer-Aided Design (FMCAD), pp. 139–145. IEEE Computer Society, Washington, DC, (2007)
7. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi c programs. In: Tools and Algorithms for the Construction and Analysis of Systems, pp. 168–176. Springer, Berlin (2004)
8. Clarke, E.M., Emerson, E.: Synthesis of synchronization skeletons for branching time temporal logic. In: Logics of Programs. Lecture Notes in Computer Science, vol. 131. Springer, Berlin (1981)
9. Haedicke, F., Große, D., Drechsler, R.: A guiding coverage metric for formal verification. In: DATE, pp. 617–622, 2012
10. Holzmann, G.J.: The SPIN model checker. IEEE Trans. Softw. Eng. **23**, 279–295 (1997)
11. Katz, S., Grumberg, O., Geist, D.: "have i written enough properties?" - a method of comparison between specification and implementation. In: Proceedings of Advanced Research Working Conference on Correct Hardware Design and Verification Methods (CHARME), pp. 280–297. Springer, London (1999)
12. Kurshan, R.P.: Computer-Aided Verification of Coordinating Processes – The Automata-Theoretic Approach. Princeton University Press, Princeton, NJ (1994)
13. Schlich, B.: Model checking of software for microcontrollers. ACM Trans. Embed. Comput. Syst. **9**(4), 36:1–36:27 (2010)
14. Schmidt, B., Villarraga, C., Fehmel, T., Bormann, J., Wedler, M., Nguyen, M., Stoffel, D., Kunz, W.: A new formal verification approach for hardware-dependent embedded system software. IPSJ Trans. Syst. LSI Des. Methodol. **6**, 135–145 (2013)
15. Spear, C.: SystemVerilog for Verification: A Guide to Learning the Testbench Language Features. Springer, Berlin (2008)
16. The MathWorks, Inc. USA: Polyspace - Static Analysis Tools (2014). http://www.mathworks.com/products/polyspace/ (2014)
17. Urdahl, J., Stoffel, D., Kunz, W.: Path predicate abstraction for sound system-level models of rt-level circuit designs. IEEE Trans. Comput. Aided Des. Integr. Circuits Syst. **33**(2), 291–304 (2014)
18. Villarraga, C., Schmidt, B., Bartsch, C., Bormann, J., Stoffel, D., Kunz, W.: An equivalence checker for hardware-dependent software. In: 11. ACM-IEEE International Conference on Formal Methods and Models for Codesign, pp. 119–128 (2013)
19. Yang, F.Z., Ganai, M., Gupta, A., Shlyakhter, I., Ashar, P.: FSoft software verification platform. In: Proceedings of International Conference Computer Aided Verification (CAV), pp. 301–306. Springer, Berlin (2005)