Using Sequence Diagrams to Specify and to Generate RTL Assertions

Martin Schweikert
Computer Systems Group
Technische Universität Darmstadt
Merckstr. 25
64283 Darmstadt
Germany
www.rs.tu-darmstadt.de
schweikert@rs.tu-darmstadt.de

Tobias Dornes
Computer Systems Group
Technische Universität Darmstadt
Merckstr. 25
64283 Darmstadt
Germany
www.rs.tu-darmstadt.de
dornes@rs.tu-darmstadt.de

Hans Eveking
Computer Systems Group
Technische Universität Darmstadt
Merckstr. 25
64283 Darmstadt
Germany
www.rs.tu-darmstadt.de
eveking@rs.tu-darmstadt.de

In the field of hardware development, it is essential to prove the correctness of a new design. In order to check a design, verification engineers often use assertions written in a property or hardware specification language. Sequence diagrams are well established in the field of software engineering and allow easy and compact specification of protocols. We propose to use sequence diagrams to specify register transfer level (RTL) behaviour and present an approach to automatically generate temporal properties out of these diagrams. The validity of the approach is illustrated by verifying a Wishbone system-on-a-chip (SoC) local interconnect bus.

Sequence Diagrams, Formal Verification, Model-Checking, RTL, Assertions, Properties

1. INTRODUCTION

Product development usually starts with a written specification. Today's semiconductors are partitioned into subsystems that communicate via special SoC networks. For the development of the subsystems, the specification has to be read and understood by several engineers. This may lead to misinterpretations and thus to errors when interconnecting the subsystems. Instead, we propose to specify the inter-object behaviour using sequence diagrams. By means of the approach presented in the following, it is possible to automatically generate properties out of these diagrams. The generated properties are used for model checking.

Sequence diagrams have originally been created to specify scenarios. These scenarios offer a good human readable representation of system behaviour but are on the other hand, a loosely coupled and rather informal resource for the further design process. The problem is to transfer these "weak" system models into properties with a well-defined syntax and semantics.

We propose to use sequence diagrams on register transfer level (RTL) and to automatically extract assertions to verify RTL designs. In contrast to statecharts, sequence diagrams are considered to be easily understandable even for non-technical people.

The properties are individually generated for each subsystem. Thus, only one sequence diagram that specifies the behaviour of several subsystems has to be created. Because the properties are generated using the same sequence diagram, the resulting substystem specifications are automatically consistent. Errors due to different interpretations of subsystem interfaces are impossible.

This paper is structured as follows: Section 2 summarises the background of formal verification and sequence diagrams. Section 3 describes the basics of our proposed algorithm, used for the case study in section 4. Related work is discussed in section 5. The paper concludes with section 6.

2. PRELIMINARIES

2.1. Formal Verification

In this paper, we present an approach to automatically generate safety properties from sequence diagrams. These safety properties are used for formal verification to check against a given implementation.

We use *interval property checking* as model checking technique, a modification of *bounded model checking* (see Nguyen et al. 2008).

A property consists of an assumption and a commitment. When the assumption is matched by the current signal assignment, the commitment must be satisfied. Both blocks, the assumption block and the commitment block, are specified as temporal Boolean expressions that are combined together using the Boolean *implication*. The value of a signal is sampled at its specified time point. The time variable *t* that specifies an arbitrary time point, is used to specify the temporal relation. Other time points are derived from *t*, *t+1* specifies the subsequent time point and *t-1* specifies the preceding one. The following example property specifies that whenever the signal *a* is zero, signal *b* must be deasserted in the same clock cycle and then must be asserted in the following one:

$$a^t =' 0' \rightarrow b^t =' 0' \wedge b^{t+1} =' 1'$$

In the following, we introduce an abbreviation for properties. We use the literal itself as an assignment of '1' and the literal with a preceding Boolean negation ($\neg$) as assignment of '0'. The property above in shortform is:

$$\neg a^t \rightarrow \neg b^t \wedge b^{t+1}$$

The resulting properties are checked against an implementation using a formal model checking tool. Instead of our general notation, model checking tools usually use at least one of the existing property description languages, e.g. PSL[1] or SVA[2]. Hardware descripting languages (HDL), such as VHDL or Verilog, are used to specify the implementation.

2.2. Sequence Diagrams

There are several standards defining sequence diagrams. The most common ones are the *message sequence charts* (MSC), the UML *sequence diagrams* (on which we focus in this paper) and the *live sequence charts* (LSC). MSCs have been standardised by the *International Telecommunication Union* (ITU)[3] in 1992 and unified the several existing dialects for sequence diagrams. LSCs, introduced by Damm and Harel (1998), are an enhancement of the MSCs; we will discuss them in section 5.1.

UML consists of a wide variety of different diagram types. Two classes are being distinguished:

[1]Property Specification Language, see IEEE Standard 1850-2005
[2]System Verilog Assertions, see IEEE Standard 1800-2009
[3]http://www.itu.int/

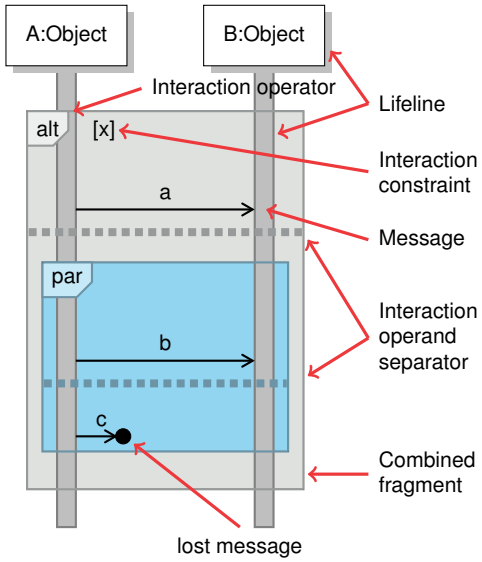

Figure 1: *Combined fragments*

Structure diagrams and *behaviour diagrams*. The first ones define static, structural constructs like classes of object oriented software or subsystems in system modelling, the latter ones define the dynamic behaviour of the system. In this paper, we focus on sequence diagrams, belonging to the group of behaviour diagrams. Although sequence diagrams were already present in the earliest versions of the UML, they were not considered to be usable until Version 2.0 of the UML was released in 2005, which was inspired by the MSCs and introduced a more powerful syntax.[4]

Sequence diagrams focus on the message exchange between systems. The usage of the basic diagram elements is demonstrated in Fig. 1. The related systems are represented as actor by vertical lifelines. Time flows from top to bottom, and exchanged messages are drawn as horizontal arrows between the lifelines.

In the diagram, combined fragments are depicted as a frame with an interaction operator in the upper left corner. Interaction operands are contained in combined fragments and may include messages and combined fragments. A combined fragment is always filled with at least one interaction operand. If there is more than one operand, they are separated by a horizontal dashed line.

A combined Fragment defines a legal sequence of messages by combining the content of the covered interaction operands and the used interaction operator. The following interaction operators are relevant to understand our approach:

[4]A detailed definition of the sequence diagram model elements is specified at http://www.omg.org/spec/UML/

- **par** Interaction operands contained in a combined fragment labelled with “par” are executed in parallel.
- **alt** “alt” specifies at least two possible alternatives of behaviour.
- **opt** An interaction operand with “opt” operator is optional.
- **loop** A “loop” operand may be repeated several times.

Two combined fragments are shown in the sequence diagram in Fig. 1. The *par*-fragment is contained in an *alt*-fragment. Both Fragments consist of two interaction operands.

An interaction operand may be guarded with a constraint, depicted as an expression enclosed in squared brackets. By means of these guards, it is e.g. possible to specify which of the interaction operands in a combined fragment with “alt” operands is used.

The message order defined by a sequence diagram defines a valid behaviour of the specified system. The example sequence diagram, given in Fig. 6, starts with messages *e1* and *e2*, followed by the two parallel messages *a1* and *c1*. The loop contains an *alt* combined fragment that is followed by an optional fragment and two messages.

3. GENERATING ASSERTIONS

In the following, we describe in general, how to read and how to translate sequence diagrams that are used to specify behaviour on register transfer layer (RTL). An example is given in the last part of the section.

3.1. Interpretation of sequence diagrams on RT-layer

Sequence diagrams have their origin in software engineering. Thus, messages exchanged between subsystems are usually complex and carry a lot of information. Since there are no generally accepted rules on how to specify an RTL design using sequence diagrams, we introduce the following conventions:

- Each message in the sequence diagram is interpreted as an output or internal signal value valid for exactly one clock cycle.
- There is no implicit retention property. As common in the verification domain, signals are undefined if not explicitly specified. On the other hand, hardware description languages drive signals with a constant value as long the signal is not redefined.

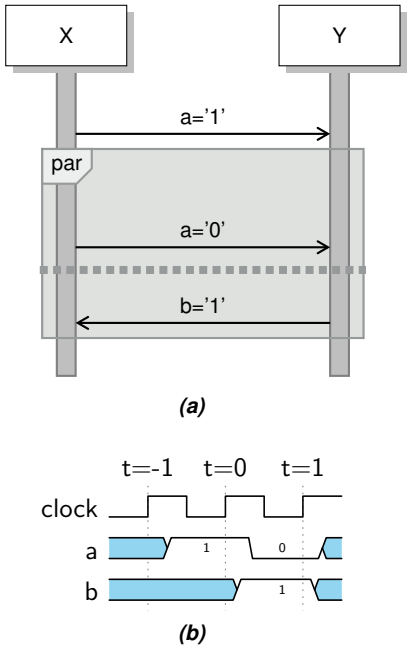

Figure 2: *Intrepretation of sequence diagrams as timing-diagram.*

There is an additional convention for the interpretation of sequence diagrams that has not been introduced for our special purposes, but is a generally accepted agreement of most of the related approaches (see section 5):

- Every actor in the sequence diagram has to try to satisfy the behaviour specified by the sequence diagram by driving its signals with the sepecified value. An actor is not responsible for violations of the sequence diagrams due to illegally received signals.

According to these conventions and the other related approaches, a message is treated as an action for the sending actor. The same message is treated as an event from the receiver’s point of view.

Fig. 2 contains an example demonstrating the coherence between a sequence diagram and the corresponding timing diagram. The sequence diagram contains messages *a* and *b* in sequential order. In the derived timing-diagram in Fig. 2b, signal *a* is asserted at time point *t=0* for one clock cycle and is deasserted at time point *t=1*. The signal is undefined at any other time points. Signal *b* is asserted at time point *t=1*. It is undefined at the subsequent time point and the preceding ones.

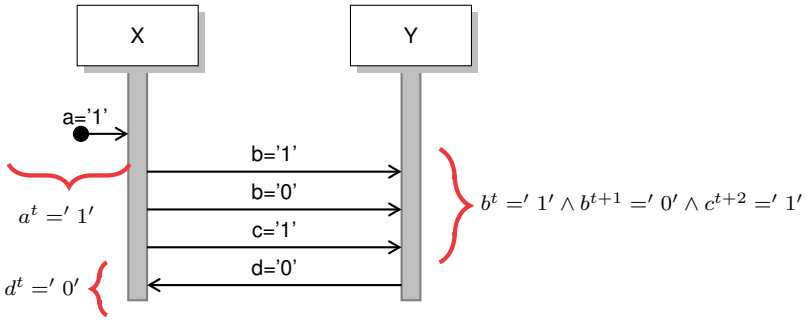

Figure 3: *Synthesizing properties*

3.2. Assertion Synthesis Approach

Properties are generated by composing temporal-boolean expressions that are built from one or more diagram elements. The simplest possible temporal-boolean expression solely contains one message, i.e. the message label is directly copied to the resulting expression. This is depicted in Fig. 3, where message *a='1'* is equal to the message label.

A message sequence is translated into one temporal-boolean expression as pointed out in Fig. 3. The referenced time point for each message depends on its position in the message sequence.

Properties are individually generated for each actor by composing the temporal-boolean expressions. The expression for the first message *a* is used as assumption for the first property of actor *X*. Since the three following ones are actions from the actor's point of view, the corresponding expression is used as commitment. The assumption (message *a*) has a duration of one clock cycle; hence the commitment must also be shifted by one clock cycle to match the sequence diagram's temporal specification.

The first property is generated for actor *X*:

$$a^t \rightarrow b^{t+1} \wedge \neg b^{t+2} \wedge c^{t+3}$$

The second property describes the behaviour of actor *Y*:

$$b^t \wedge \neg b^{t+1} \wedge c^{t+2} \rightarrow \neg d^{t+3}$$

The translation of combined fragments into expressions depends on the content and the interaction operator of the fragment. The temporal-boolean expression of a fragment with *par*-Operator is generated by conjoining the contained elements using the *and*-operator and the identical temporal reference for each element.

A fragment with an *alt*-Operator may or may not have a constraint. For alternative execution without constraint, the approach is similar to the approach for parallel fragments, but the *or*-operator is used. If a constraint is present, the following Boolean expression is used:

$$(< \textit{constraint} > \wedge < \textit{if-part} >)$$
$$\vee(\neg < \textit{constraint} > \wedge < \textit{else-part} >)$$

The constraint is directly copied from the sequence diagram, thus any temporal-boolean expression may be used. Temporal references must be specified with respect to the surrounding combined fragment, therefor x^{-1} is used to refer to signal x one clock cycle earlier as the combined fragment, x^{+1} refers to the preceding clock cycle etc.

Each diagram element covers a specific time frame. A message has a duration of one clock cycle and a sequence of messages has a duration that is equal to the number of messages. A combined fragment with *par*-operator or *alt*-operator that solely contains elements with known duration also has a known duration. The duration of such fragments is equal to the longest contained element.

The duration of combined fragments with *loop*-operator or *opt*-operator is unknown. These fragments cannot be translated into one single expression but must be translated into a property.

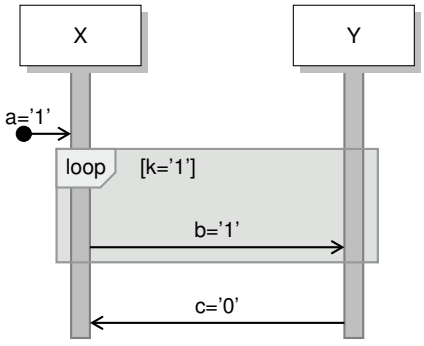

Figure 4: *Sequence diagram containing a loop*

Fig. 4 depicts a sequence diagram containing a loop. Two properties have to be generated for describing a loop: The first property specifies the loop entry; the second one specifies the continuation and the abortion of the property.

The first property is build using the expression describing the content of the *loop* combined fragment as commitment and the expression describing the preceding element as assumption. The following property is generated for actor *X* from the example in Fig. 4:

$$a^t \rightarrow b^{t+1}$$

The constraint is used in the same way as the fragment with *alt*-Operator to conjoin the temporal-boolean expression of the loop with the expression for the subsequent element of the loop. The resulting expression is used as commitment of the second property. The expression of the loop is used as assumption. The resulting second property for actor *X* is:

$$b^t \rightarrow (k^{t+1} \wedge b^{t+1}) \vee (\neg k^{t+1} \wedge \neg c^{t+1})$$

Note that our interpretation of loops differs from the official standard. The properties we generate from the sequence diagrams ensure that the loop is executed at least once. This is a violation of the standard, which says that the constraint must be checked before the first execution. The advantage of our solution is that it enables the user to simply select between two opportunities. If the behaviour according to the standard is required then the user must cover the loop with a combined fragment with *opt*-Operator that has the same constraint as the loop.

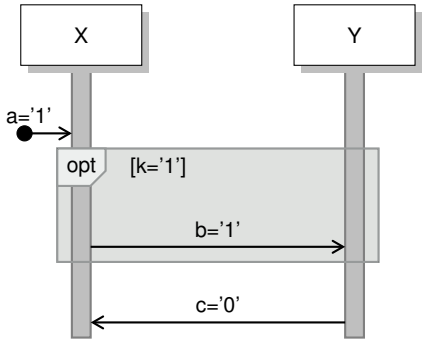

Figure 5: *Sequence diagram containing an optional fragment*

The translation of combined fragments with *opt*-Operator is quite similar to the translation of loops. Again, two properties are generated. The first one conjoins the expression that describes the commitment of the *optional* fragment with the expression for the subsequent element. The second property uses the expression for the *optional* fragment as assumption and the expression describing the subsequent element as commitment. The generated properties for actor X in Fig. 5 are:

$$a^t \rightarrow (k^{t+1} \wedge b^{t+1}) \vee (\neg k^{t+1} \wedge \neg c^{t+1})$$
$$b^t \rightarrow \neg c^{t+1}$$

Loops and *optional* fragment may or may not be guarded with a constraint. Fragments without constraint are resulting in non-deterministic properties and are translated in the same way as the deterministic ones above. As in *alternatives* without constraint, the *or*-operator is used.

Since we generate safety properties it is not possible to build a temporal-boolean expression describing a fragment with unknown duration at once, those fragments have an *entering expression* and an *exit expression*. The *entering expression* is the largest temporal-boolean expression that can be built with the elements at the beginning of the fragment. Accordingly, the *exit expression* is the largest temporal-boolean expression that can be built with the elements at the end of the fragment. An example is given in the following section.

The overall synthesis is summarised as follows:

1. Try to build a temporal-boolean expression for each diagram element. Messages and sequences of messages can be described as single expression. This is also possible for sequences of elements with known duration and for fragments that solely contain elements with known duration and have an operator of type *par* or *alt*. *Entering expressions* and *exit expressions* are built for all other diagram elements. The generation of temporal-boolean expression works with a *greedy algorithm*, i.e. it always tries to build the largest possible expression.
2. Build properties using the temporal-boolean expression. Starting with the first message as assumption for the first property, the conjoined expression for all allowed succeeding diagram elements has to be built and is used as commitment. The expressions for those succeeding elements are then used as assumptions for the following properties and so on.

The approach is used recursively, i.e. the generation of properties of a diagram works in the same way as

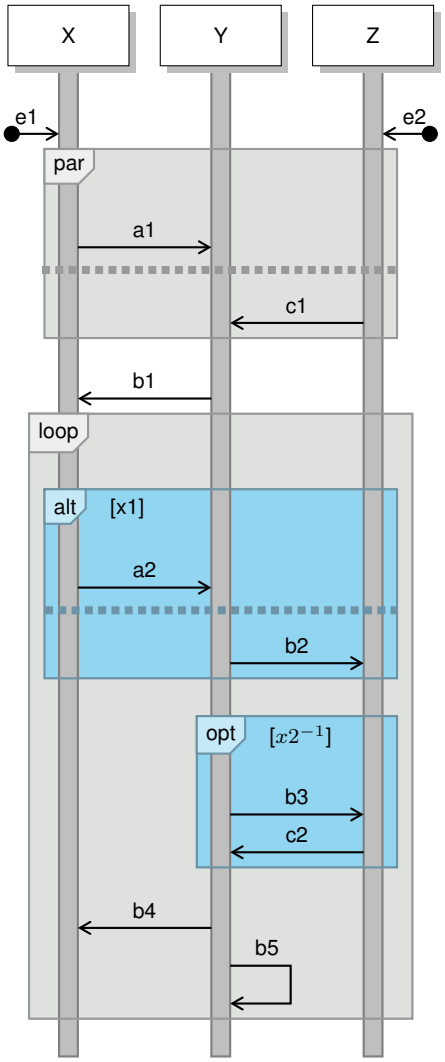

Figure 6: *Example*

properties are generated within a diagram element of unknown duration.

Interaction uses may be used to structure diagrams. An *interaction use* is depicted as a frame (as a fragment) with the keyword *ref* in the upper left corner. The name of the referenced diagram is written in the middle of the frame. *Interaction uses* are transparent in the sense of our translation approach since the *interaction use* is simply replaced by the content of the referenced sequence diagram.

3.3. Example

In the following, we demonstrate our approach with the sequence diagram depicted in Fig. 6. The properties are generated individually for each actor. Although we describe the generation process in the following at once, in fact our approach has to be executed three times. The synthesis result is a total of nine properties (see Fig. 7).

(1.1)	$e1^t$	$\rightarrow$	$a1^{t+1}$
(1.2)	$b1^t$	$\rightarrow$	$\neg x1^{t+1} \vee a2^{t+1}$
(1.3)	$b4^t$	$\rightarrow$	$\neg x1^{t+2} \vee a2^{t+2}$

(a) *Generated properties for actor X*

(2.1)	$a1^t \wedge c1^t$	$\rightarrow$	$b1^{t+1}$
(2.2)	$b1^t$	$\rightarrow$	$x1^{t+1} \vee b2^{t+1}$
(2.3)	$(x1^t \wedge a2^t)$ $\vee(\neg x1^t \wedge b2^t)$	$\rightarrow$	$(x2^t \wedge b3^{t+1})$ $\vee(\neg x2^t \wedge b4^{t+1}$ $\wedge b5^{t+2})$
(2.4)	$b3^t \wedge c2^{t+1}$	$\rightarrow$	$b4^{t+2} \wedge b5^{t+3}$

(b) *Generated properties for actor Y*

(3.1)	$e2^t$	$\rightarrow$	$c1^{t+1}$
(3.2)	$a2^t \vee b2^t$	$\rightarrow$	$\neg x2^t \vee c2^{t+2}$

(c) *Generated properties for actor Z*

Figure 7: *Generated properties for example sequence diagram in fig. 6*

The sequence diagram starts with signals *e1* and *e2*, followed by parallel signals *a1* and *c1*. The expression describing the whole parallel fragment is: $a1^t \wedge c1^t$. Since actor *X* is not responsible for signal *c1* and actor *Z* is not responsible for signal *a1*, those signals are omitted from the commitment in property 1.1 respectively property 3.1.

The parallel fragment is followed by a single message *b1*. This time, the expression generated for the first parallel fragment is entirely used as assumption, resulting in property 2.1.

Since loops cannot be described in one expression, the subsequent combined fragment with *loop* operator has an *entering expression* and a *exit expression*. The *loop*-fragment contains three elements: an *alt*-fragment, an *opt*-fragment and a sequence of two messages. For the *entering expression*, the largest possible expression at the beginning of the loop is build. Since the *opt*-fragment cannot be included in this expression, the *entering expression* solely contains the expression build for the *alt*-fragment: $x1^t \wedge a2^t \vee \neg x1^t \wedge b2^t$. Signal *b1* is used as assumption for properties 1.2 and 2.2. Both messages use the *entering expression* of the loop as commitment, respectively the part of the expression that contains the signals of the actor's responsibility.

Since there is no expression describing the whole loop at once, the generation of properties is recursively continued within the loop. Property 2.3 uses the *alt*-fragment as assumption. The subsequent diagram element is either the *opt*-fragment or the sequence of messages *b4* and

b5, depending on the constraint *x2*. Property 3.2 describes the *opt*-fragment from actor *Z*'s point of view. Property 2.4 conjoins the optional fragment with the last message sequence.

The *exit expression* of the loop is equal to the sequence of messages *b4* and *b5*, since it is the largest possible expression that can be built at the end of the *loop*-fragment. The loop does neither have a constraint nor a subsequent element, thus it can only be followed by itself. The *exit expression* is conjoined with the *entering expression* in property 1.3. Note, that in the property the signals *x* and *a2* are shifted for two clock cycles. The reason is that one clock cycle is blocked for signal *b5*, even though it is not visible to actor *X*.

4. CASE STUDY: VERIFYING A WISHBONE BUS

The Wishbone bus is an open source bus used to interconnect IP-cores contained in a system-on-chip. In our example, a single master is connected to a slave using a point-to-point connection. The connection is synchronous and both actors react to incoming signal changes with a delay of one cycle.

The sequence diagram shown in Fig. 8 models a simple data transfer including burst cycles according to the *advanced synchronous cycle termination* defined in the Wishbone specification[5]. Properties are generated automatically from the diagrams, using our algorithm.

Message #1 indicates the idle status. The acknowledge signal *ack* has to be set to '0' until the master starts a bus cycle by setting *cyc* (cycle output) to '1' (guard in the loop around message #1). The message *cyc* itself is not modelled here since the master is allowed to set it equal to '0' or equal to '1'. The following loop models the bus cycle. The master is allowed, but not required, to set the *stb* (strobe) signal in the same cycle as the *cyc* signal. The *stb* signal indicates valid read or write data on the bus. Message #2 and #3 model the situation that *stb* is not yet set equal to '1'. In this case, the master must assert *cyc* since it is not allowed to abort the transfer here, and the slave has to deassert *ack*.

Messages #4 and #5 indicate that *cyc* and *stb* must be asserted for at least one clock cycle. The master asserts both signals until the slave sets *ack* equal to '1' (messages #6 and #7). After that, the master has the possibility to end the cycle by deasserting *cyc*, to add waiting cycles by setting *stb* equal to '0', or to continue with the next data by driving both signals with '1'.

[5]http://www.opencores.org/projects.cgi/web/wishbone/

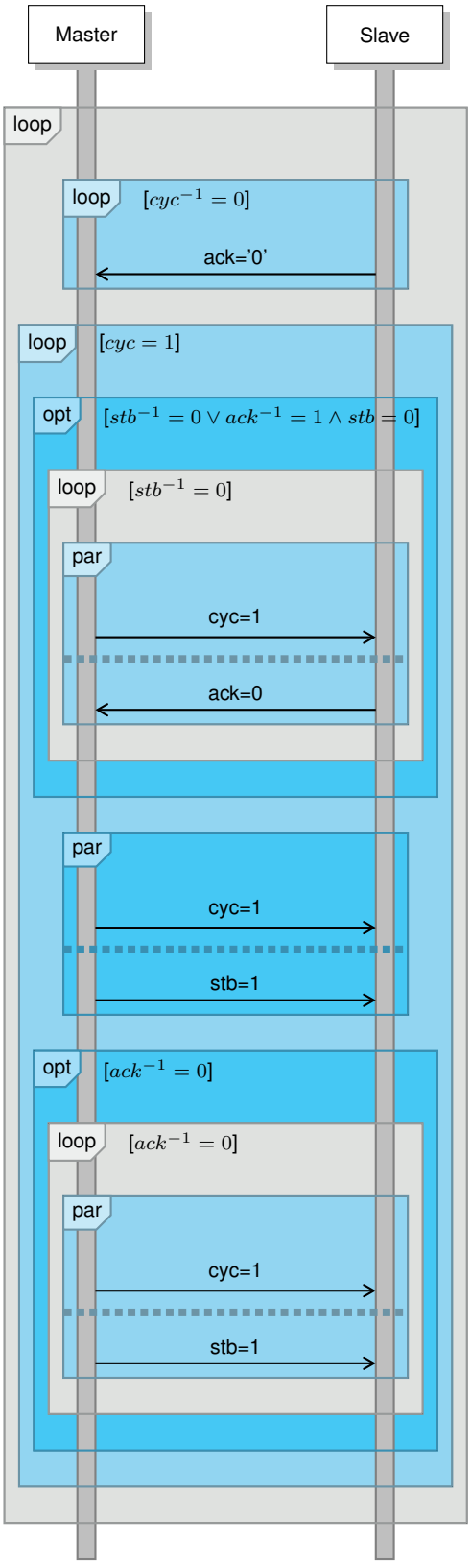

Figure 8: *Wishbone transfer*

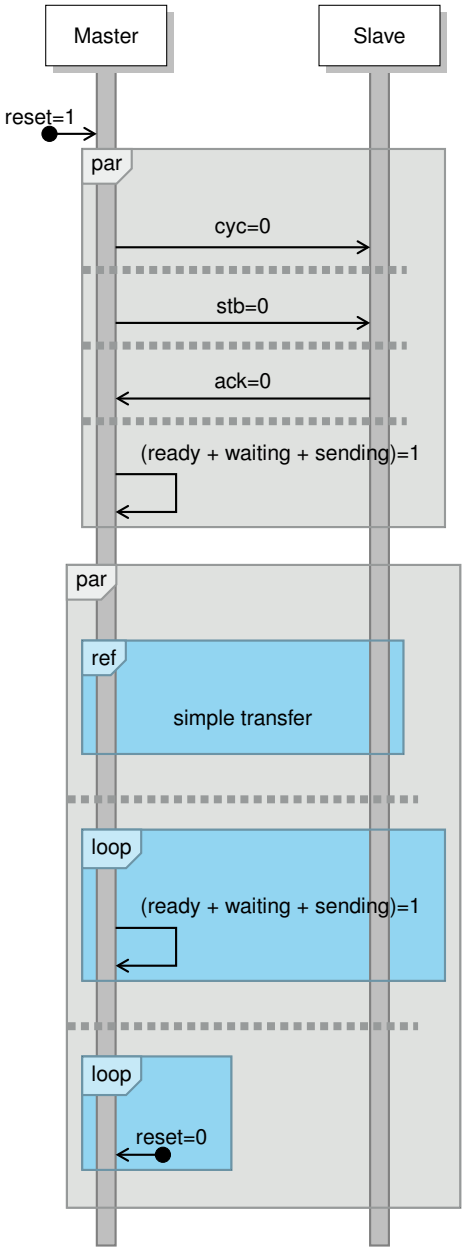

Figure 9: *Wishbone toplevel diagram*

The outer loop in the used sequence diagram does not have any subsequent diagram elements. This infinite loop must not be guarded by a condition since this would lead to undefined behaviour.

The sequence diagram shown in Fig. 8 is embedded into another one, depicted in Fig. 9. This additional diagram is required to model the reset-behaviour and to offer the designer a possibility to model invariant properties. UML's *interaction use* diagram element, labeled by *ref* is used to define a reference to another interaction, i.e. sequence diagram. The content of the referenced diagram is copied and replaces the interaction use.

The reset signal (message #1) is modelled as a message of unknown origin (a so called *found message*). The following parallel fragment models the signal values after the reset. Message #5 is an invariant. The HDL implementation of the master uses three one-hot coded states (ready, waiting and sending), only one must be active at each time point.

The second parallel fragment includes the reference of the sequence diagram in Fig. 8. Message #6 is the invariant; message #7 deasserts the reset signal. Including the referenced sequence diagram, there are three infinite loops that are contained in a parallel fragment. The generation algorithm deals with this by generating the properties for each operand separately. The pure property, generated from the diagram in Fig. 8 without the diagram from Fig. 9 for the second operand is:

$$(ready + waiting + sending)^t = 1 \rightarrow (ready + waiting + sending)^{t+1} = 1$$

This is an inductive property: If the invariant holds, it must also hold in the next cycle. In the second step of the generation algorithm, the assumptions are crosswise added to the other operands. Two assumptions are added to the property above:

$$\neg reset^t$$

$$((cyc \wedge stb) \vee \neg ack \vee (cyc \wedge \neg ack) \vee (cyc \wedge stb)^t$$

The first assumption is generated from the third operand; it deasserts the reset signal. The second one is the union of the assumptions of the properties generated from the first operand, respectively the referenced sequence diagram. In other words, the property above has to be fulfilled for every other property. The generated expression can be simplified to clarify it: $ack =' 0' \vee cyc =' 1' \wedge stb =' 1'$. This does not disturb our invariant since the slave is not allowed to drive *ack* equal to '1' if *cyc* or *stb* is equal to '0'.

The exchange of assumptions is also done in the reverse direction; every property generated from the first operand gains two extra assumptions: The reset assumption and the invariant state assumption. Since the third operand does not contain any actions, no property is generated.

The reset signal is only connected to the master since the HDL implementation of the slave does not

require any reset. If this actor would also need a reset, extra messages would have been needed.

The generation algorithm is implemented in a Java-program. Seven properties are generated for the master, and five properties are generated for the slave. The generation for each actor is done in less than a second on a Core2Duo with 1.86 GHz. The generated properties are formally verified against the VHDL reference implementation using the industrial verification suite *OneSpin 360 MV*[6].

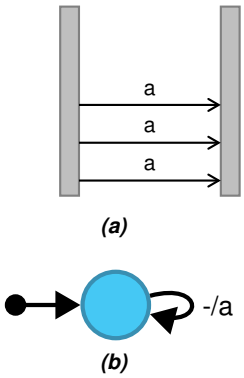

Figure 10: *Overgeneralisation of event based approaches (from Ziadi et al. 2004)*

5. RELATED WORK

5.1. Statechart Synthesis

The presented approach is closely related to the automatic generation of state automata from sequence diagrams. A number of approaches to this topic have been published; an overview is presented by Liang et al. (2006). In the following, we will discuss some work in more detail.

Koskimies and Mäkinen (1994) present a basic algorithm for the automatic synthesis of state machines from sequence diagrams. As in most of the other approaches, a “send” message is viewed as an event by the receiver, while being viewed as an action by the sender. It is assumed that the given traces of actions and events represent an unknown state machine that can be recovered using an algorithm. This algorithm aims to reuse states in the generated statechart as long as possible, resulting in the problem that the generated statechart may be more general than the source sequence diagram, i.e. it may allow undesired behaviour. The proposed solution to the problem is to add “negative“ cases to the set of traces. This allows the designer to explicitly forbid unwanted behaviour.

The approach presented by Maier and Zündorf (2003) is quite similar to the one presented by Koskimies and Mäkinen, but additionally introduces pseudo messages, offering the designer to define ”phases” being translated into inner states in a hierarchical state diagram. Other pseudo messages allow the modelling of transitions leaving and re-entering inner states.

The above approaches are characterised as *event-based* by Liang et al.. The algorithms use a set of scenarios and aim to find common states without user interaction.

Other approaches (called “condition-based”) require an explicit definition of states by the designer. Krüger et al. (1999) add named state labels to the sequence diagram. These state labels identify states, hence it is possible to enter the state in one sequence diagram and then continue in another sequence diagram at the position of the state label with the same name.

Ziadi et al. (2004) do not name the states like in the approach above but use the UML 2.0 notation, utilising the possibility to use fragments of different types. This allows the designer to implicitly model states. In Liang et al. (2006), these approaches are called “composition-based”.

Live sequence charts (LSC) are an enhancement of the MSCs and were introduced by Damm and Harel (1998). The liveness relates to the possibility to distinguish between *provisional* and *mandatory* behaviour. A live sequence chart consists of a prechart and mainchart. The prechart defines a scenario that, if matched by the current system run, activates the main chart, which then must be satisfied by the system.

Although it is possible to generate a statemachine from LSCs with the presented approaches, Harel and Marelly (2003) point out another solution. The so-called *Play-Engine* executes LSCs directly, using the current progress of the diagram as *state*.

Approaches that tend to overgeneralise the sequence diagram are inappropriate for our purposes. That affects especially the *event-based* approaches (Koskimies and Mäkinen 1994; Maier and Zündorf 2003) that are trying to reuse states. A sequence of three messages (see Fig. 10a) would be translated into a state automate consisting of a single state and an unconditional self-transition sending ’a’. It is impossible to distinguish how often ’a’ has been send (Ziadi et al. 2004).

According to Liang et al., the approach presented in this paper is a *hybrid approach*:

[6]http://www.onespin-solutions.com

- The commonalities with the *composition-based* approaches are obvious. As we do, those approaches make use of special sequence diagram syntax (e.g. UML 2.0).
- The labelling of system states is a characteristic of the *condition-based* approaches. In our approach, we use self-message for the labelling of system states.
- A *semantic-based* approach is characterized by using *cuts* as state. A *cut* is the set of all current execution locations of all lifelines in a LSC (see Harel and Marelly 2003). This is similar to the temporal-boolean expressions we use in this paper, both approaches are a more powerful and more expressive than the simple usage of messages as states (like in the *event-based* approaches).

  Both approaches are able to handle the "three a" sequence diagram (Fig 10a): With our proposed approach, it is possible to translate that sequence diagram into an expression: $a^t \wedge a^{t+1} \wedge a^{t+2}$. With the *Play-Engine*, the diagram is executed directly.

There are a many similarities between the presented approaches and our proposal to generate properties from sequence diagrams. The differences are mainly caused by the differing intentions the approaches are proposed for. Our approach focuses on the specification of systems on RT-layer, the other publications are designed to work on higher levels of abstraction.

5.2. UML and System Design

The Unified Modeling Language (UML) is a standardised way of visualising and specifying software components and their composition to large software systems. Most approaches using UML to model systems focus on a higher abstraction level, e.g. Transaction Level Modeling (TLM) (see Martin and Mueller 2005; Vanderperren et al. 2008). The Systems Modeling Language (SysML)[7] - an extension of a subset of UML - has been developed especially for the requirements of system modelling. Sequence diagrams, which we focus on in this approach, are defined in UML as well as in SysML.

UML/SysML include diagrams with weak semantics like *use case diagrams*, that help to organise the specification, as well as diagrams with a strict semantic that can be used directly. In SysML, block diagrams (an extension of UML class diagrams) may be used to model the partitioning into subsystems. These block diagrams can be translated automatically into e.g. SystemC classes.

[7] http://www.omg.org/spec/SysML/

Laemmermann et al. (2006) generate PSL properties from sequence diagrams. The generated properties are mapped to SysML block diagrams and are used as assertions in a TLM specification of the system.

6. CONCLUSION

We have presented an approach to specify a system at RT-level by means of sequence diagrams. Sequence diagrams offer a good human readability and are well established in software engineering. The standardisation of UML and its continuously growing support by hardware designers simplifies the communication in an interdisciplinary working group of hardware and software experts.

We have outlined an algorithm to automatically generate properties from an existing diagram. The approach is especially suited for the development of communication systems. A SoC bus was viewed as an example application.

A further benefit of our proposed approach is the designer's ability to model the overall behaviour of the complete system at once. Properties of the individual components can be extracted automatically afterwards. In our case study, the same diagram is used to generate the properties of a master and a slave of a communication system. This saves time, since the system has to be described only once, and, additionally, this may reduce errors in writing properties, because both property sets automatically are consistent to each other.

Currently, no comfortable solution exists to close the semantic gap between TLM and RTL. It is not possible to automatically verify the equivalence of a TLM design and an RTL design. The presented work is a step towards a solution of this problem. A possible approach might be to replace a message in a TLM sequence diagram by a whole RTL sequence diagram that models the communication in detail.

7. REFERENCES

Damm, W. and Harel, D. (1998). LSCs: Breathing Life into Message Sequence Charts. Technical report, Jerusalem, Israel, Israel.

Harel, D. and Marelly, R. (2003). *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, USA.

Koskimies, K. and Mäkinen, E. (1994). Automatic synthesis of state machines from trace diagrams. *Softw. Pract. Exper.*, 24(7):643–658.

Krüger, I., Grosu, R., Scholz, P., and Broy, M. (1999). From MSCs to statecharts. In *DIPES '98: Proceedings of the IFIP WG10.3/WG10.5 international workshop on Distributed and parallel embedded systems*, pages 61–71, Norwell, MA, USA. Kluwer Academic Publishers.

Laemmermann, S., Weiss, R., Ruf, J., Kropf, T., and Rosenstiel, W. (2006). Automatic Generation of Verification Properties for SoC Design from SysML-Diagrams. In *3rd International DAC Workshop UML for SoC Design (UML-SOC)*.

Liang, H., Dingel, J., and Diskin, Z. (2006). A comparative survey of scenario-based to state-based model synthesis approaches. In *SCESM '06: Proceedings of the 2006 international workshop on Scenarios and state machines: models, algorithms, and tools*, pages 5–12, New York, NY, USA. ACM.

Maier, T. and Zündorf, A. (2003). The Fujaba Statechart Synthesis Approach. In *2nd International Workshop on Scenarios and State Machines: Models, Algorithms, and Tools, ICSE 2003*. Wiley-VCH.

Martin, G. and Mueller, W., editors (2005). *UML for SOC Design*. Springer.

Nguyen, M., Thalmaier, M., Wedler, M., Bormann, J., Stoffel, D., and Kunz, W. (2008). Unbounded protocol compliance verification using interval property checking with invariants. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 27(11):2068 –2082.

Vanderperren, Y., Mueller, W., and Dehaene, W. (2008). UML for electronic systems design: a comprehensive overview. *Design Automation for Embedded Systems*, 12(4):261–292.

Ziadi, T., Helouet, L., and Jezequel, J.-M. (2004). Revisiting statechart synthesis with an algebraic approach. *Software Engineering, 2004. ICSE 2004. Proceedings. 26th International Conference on*, pages 242–251.