A Visual Approach to Validating System Level Designs [*]

Jochen Klose
University of Oldenburg
Klose@Informatik.Uni-Oldenburg.de

Thomas Kropf
University of Tübingen
Kropf@Informatik.Uni-Tuebingen.de

Jürgen Ruf
University of Tübingen
Ruf@Informatik.Uni-Tuebingen.de

ABSTRACT

This paper proposes a simulation-based methodology for validation of a system under design in an early phase of development. The key element of this approach is the visual specification, as Live Sequence Charts (LSCs), of the properties to be checked. The LSCs are automatically translated into the input format for the SystemC-based checker engine, which indicates during simulation, if the property is fulfilled or produces a counter-example, if the property is violated. The entire process from the visual property specification to the checking is largely automated, which makes our approach accessible even for users which have not been trained in formal methods.

Categories and Subject Descriptors

I.6.4 [**Simulation and Modeling**]: Model Validation and Analysis

General Terms

Design, Verification

Keywords

Validation Methodology, Simulation, Sequence Charts

1. INTRODUCTION

The advent of languages like SystemC [13] allows to raise the level of formalized design to higher levels of abstraction. Up to now, system level design often relied upon informal, even paper-and-pencil-based methods. Now designs can be captured and simulated at a much earlier stage. However, a simulation-based system level design requires a suitable methodology to systematically explore the design space. During this phase key properties of the system must be identified and captured in a suitable way. These properties must then be validated to gain sufficient confidence before the result is used to create an implementation on lower abstraction levels, e.g. on RTL and gate level.

To achieve this goal we propose a design and validation environment targeted at system level designers without knowledge of formal or semi-formal methods. Properties are specified by Live Sequence Charts [2], a graphical notation for specifying temporal relationships between signals and events. These are automatically translated into AR-automata, which are well suited for a simulation-guided property checking. If properties are violated, counterexamples are automatically generated which can often be easily visualized with acceptable programming effort. We have deliberately chosen simulation instead of formal verification as in the early phase of the design no fixed and complete specification is available. This specification is rather the result of this phase as a growing set of key properties of the design are identified by manual variation and exploration of the function space.
The simulation-based approach entails that not all LSC features are exploited, in particular liveness requirements are disregarded, since they are meaningless when considering finite traces.

The problem of easing the capturing of formal specification by graphical methods has been tackled before, e.g. by Schlör et al. [12], Fishler [3] or by Amla et al. [1]. Also simulation-based property checking has been presented before [11]. However, we believe that Live Sequence Charts are a formalism which – due to its simplistic and intuitive semantics – is especially suited for system level design. Moreover, to our knowledge this paper presents for the first time a simulation-guided property checking approach for SystemC based on a graphical specification technique. Our first experiences show that the ease of our approach leads to a high acceptance of designers, even if they have not been exposed to formal methods.

The paper is structured as follows: In section 2 we first describe the general approach. Afterwards, we introduce AR-automata as the underlying formalism in section 3. After Live Sequence Charts are presented in section 4 we explain how they can be translated into AR-automata. After having given some details of our implementation in section 6 we present experimental results which show the usefulness of our approach. The paper ends with conclusions.

[*]This work has been funded by the German Research Council (DFG) within the priority program *Integration of Specification Techniques with Engineering Applications* under grants number DA 206/7-2 and KR 1869/3-2.

2. OVERVIEW OF THE APPROACH

Our system level design method is based on a design flow as given in Figure 1. Basically, we use the SystemC-based property-checker described in [11] and extend it by Live Sequence Charts (LSCs), a more expressive variant of standard Message Sequence Charts and Sequence Diagrams, for the visual specification of the properties to be checked. LSCs have already been successfully used to this end in [5].

Given a design implemented in SystemC the designer specifies the properties he wants to check by simulation as LSCs. The LSCs are then translated into automata defining the semantics of the charts. In a second step each automaton is translated into an AR-automata, which is better suited for property checking. The checker then activates and monitors the selected LSCs during simulation of the SystemC design by running the correspondent AR-automata in parallel to the actual design.

The result can be a violation or a fulfillment of the property. An inconclusive result is also possible, i.e. the LSC has neither been traversed completely nor has it been violated. The point of activation, i.e. when an LSC becomes active, is part of the chart and given by a boolean condition. Two variants of checking are possible: global and local checking. Global checking means that the LSC is activated whenever its activation condition becomes true. In case of a local checking the LSC is only activated when the simulation enters a particular area of the SystemC design.

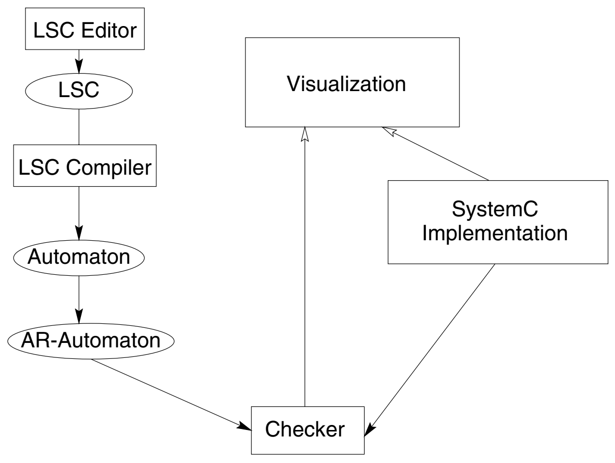

Figure 1: Overview

3. SIMULATION-GUIDED PROPERTY CHECKING

One approach bringing together simulation and formal verification is presented in [11]. The authors present a simulation-based validation technique of formal specifications. These specifications are given in a temporal logic. The formulas are translated into finite state machines called AR-automata. These AR-automata are then driven by the remaining simulation trace. If an AR-automaton reaches an accept state (A) then the formula is valid on the given simulation trace, and if the AR-automaton reaches a reject state (R), the trace is a counter example for the given specification.

The advantage of this approach is the small runtime overhead caused by checking the AR-automaton during simulation and the expressiveness of the underlying system description language (SystemC [13]). Furthermore the approach allows the user to specify formulas anywhere in the design description (white-box validation). This means, that formulas are only checked if the simulation activates them.

We will now extend the notion of AR-automata in order to cope with events instead of signal values. These AR-automata can then be used to check LSCs.

A trace consists of signals and their value changes over the time. For the remainder of this paper, assume $V = \{a, b, c, \ldots\}$ is a finite set of distinct symbols, called the *variable domain*. These symbols either represent boolean signals of the design or complex boolean conditions over arbitrary design entities. A simulation trace is then formally defined by:

Definition 1. *A trace $T[n..m]$ $(m \geq n)$ is a mapping $T : \{n, \ldots, m\} \to 2^V$. If n and m are clear from the context, we often simply write T instead of $T[n..m]$. The set of all traces is denoted by $\mathcal{T}$.*

If the variable $a \in V$ is contained in $T[i]$ then we interprete this variable to be $true$ otherwise the variable is $false$. A variable a changing from $false$ to $true$ induces a posedge event $\uparrow a$. A variable a changing from $true$ to $false$ induces a negedge event $\downarrow a$. The set of all possible events induced by a set of variables V is defined by E_V

Definition 2. *Let $T[n..m]$ be a trace, the induced event trace $T^{\updownarrow}[k..l]$ with $n \leq k \leq l \leq m$ is a mapping $T^{\updownarrow} : \{k, \ldots, l\} \to 2^{E_V}$ defined through:*

- *$T^{\updownarrow}[k] = \emptyset$ if $n = k$*
- *for all i with $k \leq i \leq l$ and $n < i$ it holds*
  $\uparrow a \in T^{\updownarrow}[i] \Leftrightarrow a \notin T[i-1] \wedge a \in T[i]$
- *for all i with $k \leq i \leq l$ and $n < i$ it holds*
  $\downarrow a \in T^{\updownarrow}[i] \Leftrightarrow a \in T[i-1] \wedge a \notin T[i]$

For $k > n$ or $l < m$ we call the event trace an induced event subtrace. In contrast to induced event traces, induced event subtraces may have events in the first time period.

Now we are able to extend the notion of AR-automata as it was presented in [11].

Definition 3. *An AR-automaton is a 5-tuple $\mathcal{A} = (S, \to, A, R, s_0)$ where $S = \{s_1, \ldots, s_n\}$ is a finite set of states, $\to \subseteq S \times 2^{E_V} \times S$ is the transition relation, $A \subset S$ is the set of* accepting states, *$R \subset S$ is the set of* rejecting states *and $s_0 \in S$ is the start state of A*

We write $s_i \xrightarrow{in} s_j$ to express that there is a transition from s_i to s_j labeled with $in \subseteq 2^{E_V}$.

In the remainder of this paper we will only operate with deterministic AR-automaton. If we apply a simulation trace to a deterministic AR-automaton, then we obtain a sequence of states called a run.

Definition 4. *Let $\mathcal{A}$ be an AR-automaton and $T[0..m]$ be a trace. A* run *of T with respect to $\mathcal{A}$ is a sequence of states $s_0, s_1, \ldots, s_n$ such that $s_i \overset{T^{\updownarrow}[i]}{\to} s_{i+1}$ holds for $0 \leq i < m$.*

We can also compute runs with respect to induced event subtraces. Now we can define the acceptance or rejection of traces applied to an AR-automaton.

Definition 5. *Let $\mathcal{A} = (S, \rightarrow, A, R, s_0)$ be an AR-automaton and $T[0..m]$ be a trace.*

- *T is called an* accepted trace *if for the run $s_0, s_1, \ldots, s_{m+1}$ induced by T, there is a j with $0 \leq j \leq m+1$ with $s_j \in A$ and for all $k < j$ holds $s_k \notin R$. Accordingly, this particular run is called an* accepted run.
- *T is called a* rejected trace *if for the run $s_0, s_1, \ldots, s_{m+1}$ induced by T, there is a j with $0 \leq j \leq m+1$ with $s_j \in R$ and for all $k < j$ holds $s_k \notin A$. Accordingly, this particular run is called a* rejected run.

An analog definition can be given for induced event subtraces.

4. LIVE SEQUENCE CHARTS

4.1 LSC Overview

Live Sequence Charts (LSCs) have been developed in [2] to overcome several shortcomings – with respect to a formal usage – of normal Message Sequence Charts [4] and Sequence Diagrams (SDs) of UML (Unified Modeling Language, [10]. The major points of criticism are

1. Only an existential or scenario view is supported by MSCs and SDs, i.e. they only describe a sample behavior of a system, one possible communication sequence.
2. It remains unclear when the communication described in a chart should be observed, i.e. when the MSC/SD should be activated.
3. It is impossible to specify if progress is enforced or not, e.g. if a certain point within the chart must be reached or a message must be received.
4. There is no formal semantics for Sequence Diagrams.

The basic idea of LSCs is the distinction of *mandatory* and *possible* behavior, where the former constitutes the added expressiveness of LSCs and the latter corresponds to classical MSCs and SDs. On the chart level the existential view represents the possible variant and a new *universal* interpretation is added for the mandatory variant. A universal LSC has to be satisfied by *all* system runs, not just by a sample one. In our approach we do not distinguish between universal and existential LSCs, since we in general cannot observe all system runs by simulation.

The second point of criticism raised above is answered by adding a boolean *activation condition* to each LSC, which activates the chart whenever it is evaluated to true. It is furthermore complemented by an *activation mode* which can be either *initial, invariant* or *iterative.* The initial mode means that the behavior described in the LSC has to be observed at system start. The activation condition is typically simply true for this mode, since the activation point is already sufficiently characterized. Specifying a non-trivial activation condition for this mode thus amounts to a restriction of the system start state. If such an initial activation condition does not hold, the entire LSC is considered violated. The invariant mode indicates that the chart is activated whenever the activation condition holds. The iterative mode is similar to the invariant mode, but it additionally requires that only one incarnation of each LSC may be active at any point in time, i.e. an LSC cannot be re-activated while it is still active.

We will not elaborate on the third point above, since liveness is irrelevant when only considering finite system runs. A sketch of the semantics is presented below.

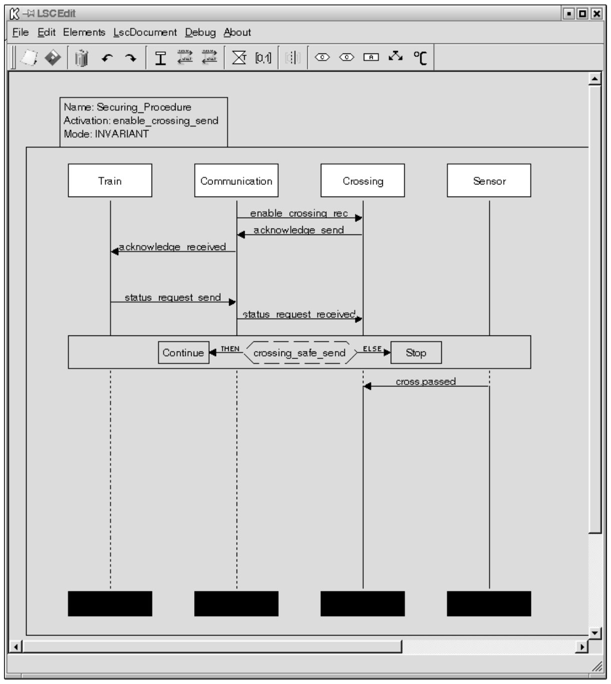
Name: Securing_Procedure
Activation: enable_crossing_send
Mode: INVARIANT

Figure 2: LSC describing the securing protocol

Figure 2 shows an example LSC for the train system application, which is described in more detail in section 6. It shows the message exchange necessary for the securing of a crossing. The participating components are the train, the crossing, the communication channel which transports the messages between them, and the pass sensor, which notifies the crossing when the train has passed. The protocol starts when the train requests the securing of the crossing by sending the message `enable_crossing` which therefore appears as the activation condition of the LSC. This message is relayed to the crossing, which immediately acknowledges its receipt and tries to secure the crossing. The train meanwhile waits as long as the crossing normally needs to be secured, and then requests the current status of the crossing.

At this point the LSC contains a branching sub-chart, since the protocol may evolve in two different directions: Either the crossing has been secured or some error has occurred. In the first case the crossing reports its status (safe) to the train, which can then safely pass. In the second case the crossing does not respond to the status request and the train must stop in front of the crossing. The driver then has to ensure that the crossing can be passed before continuing. These alternatives are captured by the sub-chart construct, which switches on the presence of the reply of the crossing (`crossing_safe`) and then enters the corresponding sub-LSC. In the case of a reply the sub-LSC `Continue` (figure 3)is entered, which contains the behavior for this case, otherwise the sub-LSC `Stop` (figure 4) is entered, which describes the second alternative. Regardless of the branch taken, the train will pass the crossing and activate the pass sensor.

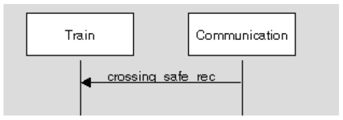

Figure 3: Sub-LSC for the good case

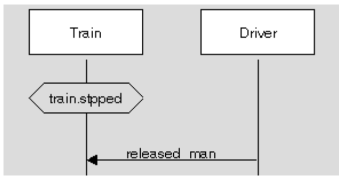

Figure 4: Sub-LSC for the error case

4.2 Construction of the AR-Automaton

A key point of LSCs with respect to our property-checking approach is that their formal semantics is defined by automata, which accept all those communication sequences satisfying the LSC. The basic strategy of the algorithm for the construction of an automaton from an LSC is to walk through the LSC from top to bottom while respecting the partial order defined by the LSC. The algorithm uses *cuts* to step through the LSC, where a cut contains exactly one location of each instance line. A cut shows how far the algorithm has already progressed through the LSC. Each cut becomes a state in the automaton generated by the algorithm and transitions in the automaton represent the successor relation among the cuts. Figure 5 shows a sample cut through the LSC of figure 2 and figure 6 shows the complete automaton for this LSC, with the state corresponding to the cut of figure 5 being shaded.
The details of the algorithm can be found in [7].

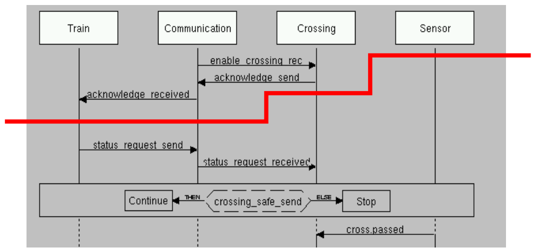

Figure 5: Cut example

Note that the automaton generated by the algorithm is not an AR-automaton; the transformation however is easy to accomplish. The main difference, as figure 6 illustrates, is that LSC automata only have an accept state, but no reject state. The only accept state is the state which corresponds to the complete traversal of the LSC (the last state in figure 6); all other states are pending, i.e. neither accepting nor

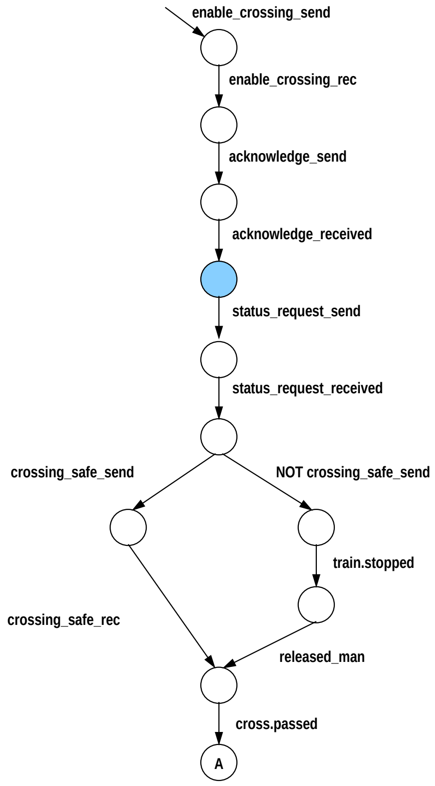

Figure 6: Automaton for LSC Securing_Procedure

rejecting. In order to transform an LSC automaton into an AR-automaton it suffices to introduce the reject state and add transitions from all pending states to it for all combinations of input signals, which are not already present on transitions in the original automaton. For the state corresponding to the cut of figure 5 the transition annotation would e.g. be $\neg \mathit{status\_request\_send} \wedge (\mathit{enable\_crossing\_send} \vee \cdots \vee \mathit{cross.passed})$.

This conversion of automata entails that we do not use all the features, which LSCs have to offer, such as liveness or interpretation (existential or universal) of the LSCs. This is due to the fact that we are monitoring a simulation and not formally verifying a design. The most valuable feature – apart from the fact that the semantics of LSCs is given as automaton – is the possibility to characterize the point of activation.

5. IMPLEMENTATION USING SYSTEMC

For our implementation of the simulation-based LSC checker we have chosen the SystemC language [13]. SystemC allows the specification of systems on various levels of abstraction and it provides fast simulation speed due to compiled executable specifications (simulation kernel + system description). Moreover, SystemC is open source and therefore easily extendable.

We have realized the LSC checker as a stand-alone library which is linked to the simulation kernel. The user sensitizes LSCs with the `sc_lsc_check` command. This command takes the file name of an AR-automaton description produced by the AR-automaton compilation. If an LSC is sensitized, it will be checked as soon as the activation con-

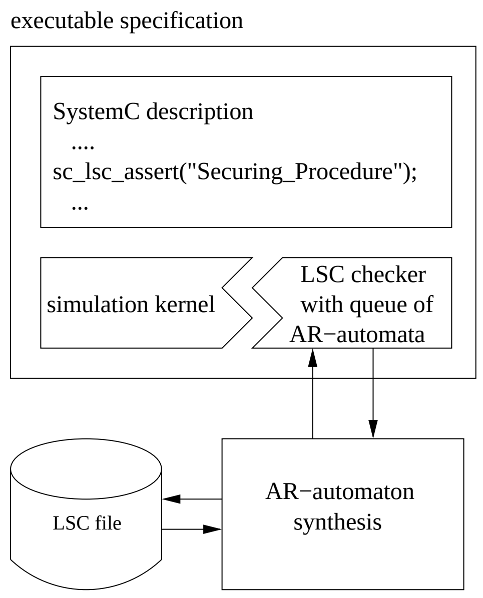

Figure 7: Architecture of our implementation

dition becomes true. The `sc_lsc_check` command may be placed anywhere in the system description. It is also possible to globally sensitize a LSC. In this case the LSC will be activated (i.e. checked) whenever the activation condition becomes true. More details are described in Section 4.

The checker is implemented as a standard SystemC process. It is running in parallel to the simulation. This implies that LSC violations can be prompted immediately to the user. Figure 7 gives an overview of our checker architecture.

Whenever the simulation runs over the `sc_lsc_check` statement, the corresponding file will be read and the automaton will be stored in a checker internal queue.

The first time the activation condition will be checked is in the current simulation cycle (the simulation cycle in which the `sc_lsc_check` statement is called). In case of the iterative or the invariant LSC-mode, the activation condition will be checked again in every further simulation cycle. In case of the initial LSC-mode the activation condition is only checked once. If this check fails, this indicates a violation of the LSC.

If the activation condition is true, the AR-automaton will be reset to its initial state. Then the AR-automaton will be simulated in parallel to the design simulation. This means that the AR-automaton performs a state transition each simulation cycle depending on the current signal values.

In order to handle multiple incarnations of one LSC in the invariant mode we provide a dynamic list of current states. For each new incarnation the list is appended by the initial state and in every simulation cycle all states of the list are updated.

In case of the iterative mode, the list may have not more than one entry. Otherwise the iterative-incarnation condition of LSCs is violated and this failure is also prompted to the user. If an LSC is accepted or rejected (i.e. the AR-automaton reaches an accept or a reject state) the user is informed and the state is deleted from the state list.

6. SAMPLE APPLICATIONS

6.1 Radio-based Signaling System

The first application we use to demonstrate our approach is part of the specification of a wireless train control system, which is currently developed by the German railroad company Deutsche Bahn. It has been made available as a trial case study within the DFG-funded program mentioned above and is focusing on the interaction between train and level crossings. The SystemC implementation we use is based on a STATEMATE model; see [6] for details.

The basic idea of wireless train control is to remove as much wayside equipment – like signals and the wiring connecting it to crossings, switches, etc. – as possible and use radio-based communication instead. The main components of our train system are: the train and the crossing, which communicate via radio. Thus in the protocol given by the LSC in figure 2 the communication channel represents the radio link which the train has established prior to contacting the crossing. The crossing consists of a communication unit, a traffic light, a barrier and a pass sensor. The communication between the radio unit and the other units of the crossing does not use a radio link, but are rather connected conventionally.
The typical communication between an approaching train and a crossing is described by the LSC in figure 2. The communication channel has been modeled explicitly as a separate component in order to be able to introduce transmission errors into the system.

The SystemC implementation not only covers the controllers for the peripheral units of the crossing, but also includes the physical elements, i.e. the actual traffic lights, barriers and pass sensor. These components can also be disrupted so as to model physical failures or malfunctions, such as a stuck barrier or a failure of the yellow light. The train additionally contains a component, which calculates the maximal speed of the train, controls that the train does not exceed this speed and computes the point in time when to contact the crossing.

In the SystemC implementation each component – train, crossing, and communication channel – is modeled by a process.

6.2 Holonic Material Transport System

The holonic material transport system consists of an input station, three machines, an output station and three autonomous transport vehicles (the holons). Two of the three machines are for work piece processing, one is for cleaning. All holons are identical. The task of the holons is to move work pieces to the three machines in consecutive order. The work pieces leave the plant at the output station. The holons are autonomous, i.e. they look for tasks and negotiate who is performing the next task. For instance if two holons are idle and a cleaned work piece appears on the cleaning machine, the holons negotiate and the winner starts moving to the cleaning machine. In our modeling, the winner of such a negotiation is the holon, which needs less time to move to the cleaned work piece.

We split the system in different processes so that each process models a physically unit. The modeling consists of three identical holons, two processing machines, one cleaning machine, the input station, the output station, one input and one output buffer for each machine.

7. EXPERIMENTAL RESULTS

In this section we will highlight some of the results we have achieved by applying our technology to the described real-world examples.

Radio-based Signaling System

We were able to find several design errors in our SystemC implementation of the train system by employing the LSC-Checker. We first checked, if the scenario depicted in figures 2 - 4 was valid for one pass of the train. The result was a failure, i.e. the scenario described by the LSCs was violated. The reason was that a signal was not reset fast enough, so that it was recognized by the automaton as an illegal signal occurrence. Whereas this was a mere oversight in the implementation, the second try revealed a more serious error of a more conceptual nature: After introducing a barrier error into the system, which kept the barriers from closing, the train stopped in front of the crossing and resumed its journey after confirmation by the driver, as prescribed by the LSC. But after the passing of the train, the crossing did not return to its idle state. The problem was that the controller for the barrier recognized the barrier failure, but immediately afterwards, i.e. too early, returned to its idle state, in which it does not react to opening requests. When the overall controller ordered it to open the barrier again once the train had passed, it consequently remained silent and the overall controller entered into a deadlock state.

Holonic Transport System

During simulation we were interested in a correct flow of the work pieces. Therefore we have checked some LSCs describing the communication between the holons and the machines managing the work piece flow.

We found that work pieces properly move between the machines. But we also found configurations where the holons are mutually blocked.

8. CONCLUSION

We have presented a simulation-based validation environment for system level designers, which does not require knowledge of formal specification languages. Properties to be validated are specified by Live Sequence Charts, a graphical notation for specifying temporal relationships between signals and events. These are automatically translated into AR-automata, which are well suited for a simulation-guided property checking. System implementations are given in SystemC. Our approach is deliberately based on simulation as typically no fixed specification is given at the beginning of system-level design but only certain key properties. If these properties are violated during simulation, counterexamples are automatically generated which can be often visualized with limited programming effort, e.g., in Tcl. In our experiments it has turned out that this approach leads to significantly shorter validation cycles: the specification effort is reduced due to the tool-based graphical specification input, which together with a suitable presentation of possible counterexamples also eases the communication between product designer and system architect.

9. REFERENCES

[1] N. Amla, E. Emerson, R. Kurshan, and K. Namjoshi. Model Checking Synchronous Timing Diagrams. In *Proceedings Formal Methods in Computer-Aided Design (FMCAD)*, LNCS, pages 283–298. Springer Verlag, 2000.

[2] W. Damm and D. Harel. LSCs: Breathing Life into Message Sequence Charts. *Formal Methods in System Design*, 19(1):45 – 80, July 2001.

[3] K. Fisler. Timing Diagrams: Formalization and Formal Verification. *Journal of Logic, Language and Information*, 8(3), 1999.

[4] ITU-T. *ITU-T Recommendation Z.120: Message Sequence Chart (MSC)*. ITU-T, Geneva, 1999.

[5] J. Klose and M. Lettrari. Scenario-based Monitoring and Testing of Real-time UML models. In M. Gogolla and C. Kobryn, editors, *UML 2001 - The Unified Modeling Language: Modeling Languages, Concepts, and Tools*, volume 2185 of *LNCS*. Springer Verlag, 2001.

[6] J. Klose and A. Thums. The Statemate Reference Model of the Reference Case Study 'Verkehrsleittechnik'. Technical report, University of Augsburg, 2000. http://www.Informatik.Uni-Augsburg.DE/swt/formosa/RefVL/bericht.ps.gz.

[7] J. Klose and H. Wittke. An Automata Based Representation of Live Sequence Charts. In T. Margaria and W. Yi, editors, *Proceedings of TACAS 2001*, number 2031 in LNCS. Springer Verlag, 2001.

[8] T. Kropf. *Formal Hardware Verification – Methods and Systems in Comparison*, volume 1287 of *Lecture Notes in Computer Science*. Springer Verlag, state of the art report edition, August 1997.

[9] K. McMillan. *Symbolic Model Checking*. Kluwer Academic Publishers, Norwell Massachusetts, 1993.

[10] OMG. *Unified Modeling Language Specification, Version 1.3*. OMG, 1999. http://www.rational.com/uml/resources/documentation.

[11] J. Ruf, D. W. Hoffmann, T. Kropf, and W. Rosenstiel. Simulation-guided property checking based on multi-valued AR-automata. In *Design Automation and Test in Europe (DATE)*. IEEE Conmputer Society Press, Los Alamitos, 2001.

[12] R. Schlör, B. Josko, and D. Werth. Using a visual formalism for design verification in industrial environments. In *VISUAL'98*, Lecture Notes in Computer Science 1385, pages 208–221. Springer Verlag, 1998.

[13] Synopsys, CoWare and Frontier Design, www.systemc.org. *SystemC Version 2.0 User's Guide*.