Modeling and Formal Verification of Production Automation Systems*

Jürgen Ruf, Roland J. Weiss, Thomas Kropf, and Wolfgang Rosenstiel

Wilhelm-Schickard-Institut für Informatik, Universität Tübingen
Sand 13, 72076 Tübingen, Germany
{ruf,weissr,kropf,rosenstiel}@informatik.uni-tuebingen.de

Abstract. This paper presents the real-time model checker RAVEN and related theoretical background. RAVEN augments the efficiency of traditional symbolic model checking with possibilities to describe real-time systems. These extensions rely on multi-terminal binary decision diagrams to represent time delays and time intervals. The temporal logic CCTL is used to specify properties with time constraints. Another noteworthy feature of our model checker is its ability to compose a system description out of communicating modules, so called I/O-interval structures. This modular approach to system description alleviates the omnipresent state explosion problem common to all model checking tools.

The case study of a holonic[1] material transport system demonstrates how such a production automation system can be modeled in our system. We devise a detailed model of all components present in the described system. This model serves as basis for checking real-time properties of the system as well as for computing key properties like system latencies and minimal response times. A translation of the original model also allows application of another time bounded property checker for verification of the holonic production system. Finally, we present an approach combining simulation and formal verification that operates on the same system model. It enables verification of larger designs at the cost of reduced coverage. Only critical states detected during simulation runs are further subjected to exhaustive model checking. We contrast the runtimes and results of our different approaches.

1 Introduction

Real-time systems pervade almost every aspect of our daily life. Accelerating a contemporary vehicle initiates a plethora of processes involving micro controllers: fuel injection should be optimized for economical fuel usage and for smooth engine operation, wheelspinning should be avoided based on data provided by special sensors, and so on. The same applies to aerospace industry,

* The results described in this article have been achieved in the course of the DFG project GRASP within the DFG Priority Programme 1064.

[1] A holon is an autonomous and cooperative unit of a manufacturing system for transforming, transporting, storing and/or validating information and physical objects.

medical systems, home entertainment, telecommunication, large-scale industrial manufacturing, and of course the classical computer industry. Even in outdoor activities we rely on integrated circuits in equipment like avalanche transceivers and GPS receivers.

Establishing the correctness of these systems poses an increasingly difficult and time-consuming challenge in the design process. Two aspects primarily motivate the significance of the verification process:

Safety-critical systems. These systems mandate error-free operation because malfunctioning could endanger human life or the environment. Obviously, an erroneous circuit in a car controller unit constitutes a major threat for passengers and road users.

Economic risks. It is of primary importance to detect design errors in early stages of the development process. Once a hardware chip is shipped, it becomes extremely expensive to fix an overlooked fault. The Pentium floating point bug has cost Intel millions of dollars because of an insufficient verification process.

The ever increasing system complexity and shorter development cycles make verification the bottleneck in the design process. Nowadays, verification consumes up to 80% of the development time.

Simulation of the design under verification is the predominating validation technique. A testbench provides an executable model with stimuli and monitors the resulting outputs. However, for large systems simulation cannot provide a complete coverage of the system. Simulation time is becoming a prohibiting factor.

This has ignited interest in formal methods that can provide better coverage and run more efficiently in certain scenarios. Equivalence checking has become state of the art in verifying evolutionary changes in designs even for very large hardware circuits.

However, if no *golden design* exists, this technique is not applicable. Formal property checking is bridging this gap. Property checking tools try to automatically prove properties extracted from the design description against a system model, or to generate a counter example trace if the property is violated.

In this paper we present a toolset that offers the verification engineer various options for verifying real-time systems. Special emphasis is put on realizing solutions that deal with the timing aspect of these systems. We exemplify our approaches with a holonic material transport system, because production automation systems make up an important application area for real-time systems.

The rest of this paper is structured as follows. Next we present our formal model of timed systems, followed by an introduction to temporal logics for specifying real-time properties and algorithms for model checking such properties. Thereafter, the outline of our toolchain is sketched and we explain our tools in more detail. Afterward, we describe the model of a material transportation system and how our tools can help to gain confidence in the system model. We give some experimental results and conclude.

2 Formal Model of Timed Systems

A crucial property of production automation systems is the ability to express time constraints. Furthermore, it is desirable to compose a system's model out of multiple components. Our formal model of timed systems is an extension of traditional Kripke structures, which are commonly used to describe reactive systems [1].

2.1 Interval structures

Interval structures are based on Kripke structures. We use the notion of clocks to represent time. Every structure of our system contains exactly one clock representing time from a discrete domain. A clock is reset if a transition fires.

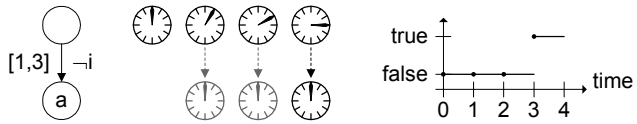

Fig. 1. Example interval structure. Delay times δ_i on timed tranistions are required to be positive integers, i.e. $\delta_i > 0$. Arrows at clocks indicate possible transitions, and the black arrow indicates the nondeterministically chosen one. The timing diagram reflects this behavior.

Fig. 1 shows a simple interval structure on the left-hand side. Clocks depict the progress of time for this interval structure, and where applicable the clocks are associated with possible state transitions symbolized by arrows. Finally, the figure contains the timing diagram for atomic proposition a when the last transition out of three possible transitions is taken.

Definition 1 An interval structure $\Im$ is a tuple $\Im = (P, S, S_0, T, L, I)$ with a set of atomic propositions P, a set of states S, a set of initial states S_0, a transition relation between the states $T \subseteq S \times S$ such that every state in S has a successor state, a state labeling function $L : S \mapsto \wp(P)$ and a transition labeling function $I : T \mapsto \wp(\mathbb{N})$.

The only difference to Kripke structures are the transitions which are labeled with delay times (not restricted to intervals). Every state of the interval structure must be left at the latest after the *maximal state time*.

Definition 2 The maximal state time of a state s is the maximal delay time $maxTime : S \mapsto \mathbb{N}$ of all outgoing transitions of s, i.e.

$$maxTime(s) := \max\{t \mid \exists s'.(s, s') \in T \wedge t = \max(I(s, s'))\}. \tag{1}$$

We also have to consider the elapsed time to determine the transition behavior of the system. Hence, the actual configuration of a system is given by pairs consisting of a state s and the actual clock value v.

Definition 3 A configuration $g = (s, v)$ is a state s associated with a clock value v. The set of all configurations of an interval structure $\Im = (P, S, S_0, T, L, I)$ is given by:

$$G = \{(s, v) \mid s \in S \wedge 0 \leq v < maxTime(s)\} \tag{2}$$

The semantics of interval structures is given by runs which are the counterparts of paths in Kripke structures.

Definition 4 Given the interval structure $\Im = (P, S, S_0, T, L, I)$ and a starting configuration $g_0 \in G$. A run is a sequence of configurations $r = (g_0, g_1, ...)$. For all $g_j = (s_j, v_j) \in G$ it holds that either

- $g_{j+1} = (s_j, v_j + 1)$ with $v_j + 1 < maxTime(s_j)$ or
- $g_{j+1} = (s_{j+1}, 0)$ with $(s_j, s_{j+1}) \in T \wedge v_j + 1 \in I(s_j, s_{j+1})$.

The set of all runs starting in g_0 is given by $\Pi(g_0)$. The i-th component of a run can be accessed by $g[i] := g_i$.

The semantics of an interval structure can also be given in terms of Kripke structures. The Kripke structure corresponding to an interval structure is obtained with a stutter state expansion operation transforming an interval structure into a Kripke structure. For every timed transition, this operation introduces new stutter states and connects them in a chain like fashion. These chains are connected with the original states of the interval structure. In the Kripke structure we call these states main states (in contrast to stutter states). Stutter state exapansion is visualized in Fig. 2.

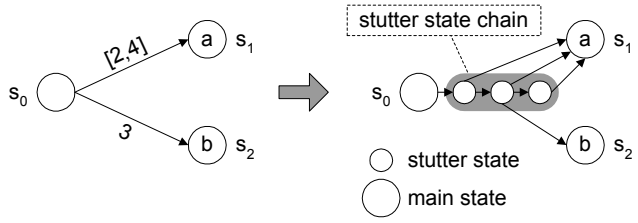

Fig. 2. Expansion of interval structure to Kripke structure.

The symbolic representation of interval structures is realized by extended characteristic functions. We group together all configurations containing the same interval structure state and map this state to the set of the corresponding clock values. All other states are mapped to $\emptyset$. For the symbolic representation of the transition relation, we map pairs of states to the set of corresponding delay times.

2.2 I/O-interval structures

Interval structures are well suited for modeling single real-time processes, but less for systems consisting of several components. Interval structures have no explicit input variables, i.e., if different interval structures communicate with each other, they have to share state variables. I/O-interval structures are better suited for modeling communicating processes. However, the result of composing I/O-interval structures is again one interval structure, the latter being better suited for model checking.

I/O-interval structures have a set of dedicated input variables P_{in}. Transitions are labeled with Boolean conditions with regard to these inputs. Input conditions have to hold during the corresponding transition times, i.e., input insensitive edges carry the formula *true*.

We formalize Boolean formulas using sets of allowed values for the input variables. An element of the set $Inp := \wp(P_{in})$ defines exactly one value for each input variable: the propositions contained in the set are *true*, all others are *false*. Hence, an element of the set $\wp(P_{in})$ defines all possible combinations of input values for one edge. For example, given the inputs a and b, the set $\{\{a\}, \{a, b\}\}$ is represented by the Boolean function $(a \wedge \neg b) \vee (a \wedge b) = a$. This example shows that the variable b does not affect the formula, i.e., the transition labeled with the formula a may be taken independently from input b.

Definition 5 An I/O-interval structure $\Im_{I/O}$ is defined by a tuple $\Im_{I/O} = (P, P_{in}, S, S_0, T, L, I, L_{in})$. The components P, S, S_0, L and I are defined analogously to interval structures, P_{in} is a finite set of atomic input propositions, the transition relation $T \subseteq S \times S$ connects pairs of states, and $L_{in} : T \mapsto \wp(Inp)$ is a transition input labeling function.

To access the first (second) state of a transition $t \in S \times S$ we write $t[1]$ ($t[2]$). This access operator is defined for all elements consisting of multiple components. We assume the following restriction on the input labeling:

$$\forall t_1, t_2 \in T. \left(\begin{array}{c} (t_1[1] = t_2[1] \wedge t_1 \neq t_2) \rightarrow \\ (L_{in}(t_1) = L_{in}(t_2) \vee L_{in}(t_1) \cap L_{in}(t_2) = \emptyset) \end{array} \right) \tag{3}$$

Formula (3) ensures that if there exist several edges starting in the same state, then their input restrictions are either equal or disjoint. Thus, input valuations on timed edges build clusters, i.e., disjoint sets of input values. All clusters of one state are disjoint, i.e., each valuation of input variables is a representative of its cluster. This condition is important to ensure an efficient translation of I/O-interval structures into Kripke structures for composition. For every cluster we have to introduce a separate chain of stutter states.

Definition 6 The cluster function $C : S \times Inp \mapsto \wp(Inp)$ computes all input valuations of a cluster represented by an arbitrary member (input valuation) of the cluster

$$C(s, i) := \begin{cases} L_{in}(s, s') & \text{if } \exists s' \in S.(s, s') \in T \wedge i \in L_{in}(s, s') \\ \emptyset & \text{otherwise} \end{cases} . \tag{4}$$

Now we describe the semantics of the I/O-interval structures by defining runs. We first need the maximal state time.

Definition 7 The maximal state time $maxTime : S \times Inp \mapsto \mathbb{N}$ is the maximal delay time of all outgoing transitions with respect to the input cluster.

$$\begin{aligned} &maxTime(s,i) := \\ &\quad \max\{v \mid \exists s' \in S.(s,s') \in T \wedge i \in L_{in}(s,s') \wedge v \in \max(I(s,s'))\} \end{aligned} \tag{5}$$

Configurations in I/O-interval structures consist of the actual state, the elapsed time, and the actual input values.

Definition 8 A configuration $g = (s,i,v)$ of an I/O-interval structure is a state s associated with an input valuation $i \in Inp$ and a clock value v. The set of all configurations of an I/O-interval structure is given by:

$$G = \{(s,i,v) \mid s \in S \wedge i \in C(s) \wedge v < maxTime(s,i)\} \tag{6}$$

Definition 9 Let $\Im_{I/O} = (P, P_{in}, S, S_0, T, L, I, L_{in})$ be an I/O-interval structure. A run is a sequence of configurations $r = (g_0, g_1, ...)$ with $g_j = (s_j, i_j, v_j) \in G$ and for all j it holds that either

- $g_{j+1} = (s_j, i_{j+1}, v_j + 1)$ with $v_j + 1 < maxTime(s_j, i_j) \wedge i_{j+1} \in C(s_j, i_j)$ or
- $g_{j+1} = (s_{j+1}, i_{j+1}, 0)$ with $t = (s_j, s_{j+1}) \in T \wedge v_j + 1 \in I(t) \wedge i_{j+1} \in P_{in}$.

We assume that for every configuration and for every input valuation there exists a successor configuration. Hence, for edges with delay times greater than one there has to be a fail state which is visited if the actual input does not fulfill the current input restriction and the delay time is not reached. This implies that transitions either have no input restriction or, if a transition with delay time δ has an input restriction $f(a_1, ..., a_n)$, there must exist a transition with interval $[1, \delta - 1]$ and input condition $\neg f(a_1, ..., a_n)$ which connects the starting state with the fail state. For unit-delay edges we have to ensure that for all input evaluations there exists a successor state.

We use a graphical notation for I/O-interval structure as shown in Fig. 3. Variables a_i represent the inputs of the modeled sub-system, and f denotes a function $f : P_{in}^n \mapsto \mathbb{B}$. We omit the delay time if $\delta_1 = \delta_2 = 1$.

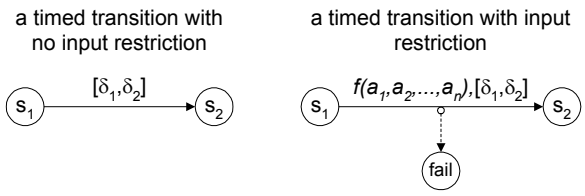

Fig. 3. Graphical notation for I/O-interval structures.

3 Property Specification with Real-Time Temporal Logics

In the previous section we have detailed our formalism called I/O-interval structures for describing timed system models. On these models, we want to check or prove real-time properties.

We are mainly dealing with reactive systems, and temporal logic is a formalism for describing transition sequences in such systems. A temporal logic provides *path quantifiers* A (all) and E (exists), and *temporal operators* X (next), F (eventually/in the future), G (globally), and U (until) additional to the typical boolean connectives.

The most widely used temporal logics in model checking and related formal verification techniques are *Computation Tree Logic* (CTL) and *Linear Temporal Logic* (LTL). Both describe overlapping subsets of CTL*, i.e. CTL* is expressive enough to state all formulas from LTL and CTL, but there exist formulas in LTL that are not expressible in CTL and vice versa. For a more detailed discussion on these fundamental temporal logics refer to chapter 3 in [1].

A major characteristic of temporal languages is the underlying model of time. In branching temporal logics, a moment in time can branch into various futures. Thus, infinite computation trees describe the systems that are subject to property checking. CTL is one such temporal logic, and it is well suited for algorithmic verification. Checking a transition system against a property given in CTL takes time linear in the length of the property specification. However, in linear temporal logics like LTL each moment in time has only one possible future. Therefore, formulas in linear temporal logics are interpreted over linear sequences that describe one computation of a system. Model checking takes time exponential in the length of a LTL specification. However, LTL is commonly regarded as more intuitive. Furthermore, dynamic validation is inherently linear as computation sequences are generated. This allows linear temporal logic specifications to be used in contexts ranging from dynamic validation to full formal verification. A thorough discussion on the trade-offs between branching and linear temporal logics is presented in [2].

All these logics have in common that they express time only implicitly, e.g. a property specifies that a state is eventually or never reached. However, real-time systems such as production automation systems often require conformance to strict time bounds. Time constraints are very important to maximize throughput times and to minimize wait times of workpieces. But furthermore, timing constraints also have a safety aspect, since the movements in such a production automation system consume time, they have to be scheduled such that no accident occurs.

In order to make time constraints explicit in property specifications we have introduced a variant of CTL called *Clocked CTL* (CCTL), and a variant of LTL called *Finite Linear Time Temporal Logic* (FLTL). We will briefly describe these two temporal logics in this section.

3.1 CCTL

CCTL [3] is a temporal logic extending CTL with quantitative bounded temporal operators. In contrast to CTL its semantics is defined over interval structures and it contains two new operators which make the specification of timed properties easier. It is a variant of RTCTL [4] adapted to our needs. The syntax of CCTL is the following:

Definition 10 Let P be a set of atomic propositions, $m \in \mathbb{N}$, and $n \in \mathbb{N} \cup \{\infty\}$. The set of all syntactically correct CCTL formulas is the smallest set satisfying the following properties:

- $P \subseteq \mathcal{F}_{CCTL}$
- if $\phi, \psi \in \mathcal{F}_{CCTL}$, then also $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \leftrightarrow \psi \in \mathcal{F}_{CCTL}$
- if $\phi, \psi \in \mathcal{F}_{CCTL}$, then also
  $\mathsf{AX}_{[m]}\phi$, $\mathsf{AG}_{[m,n]}\phi$, $\mathsf{AF}_{[m,n]}\phi$, $\mathsf{A}(\phi\mathsf{U}_{[m,n]}\psi)$, $\mathsf{A}(\phi\mathsf{C}_{[m]}\psi)$, $\mathsf{A}(\phi\mathsf{S}_{[m]}\psi) \in \mathcal{F}_{CCTL}$
- if $\phi, \psi \in \mathcal{F}_{CCTL}$, then also
  $\mathsf{EX}_{[m]}\phi$, $\mathsf{EG}_{[m,n]}\phi$, $\mathsf{EF}_{[m,n]}\phi$, $\mathsf{E}(\phi\mathsf{U}_{[m,n]}\psi)$, $\mathsf{E}(\phi\mathsf{C}_{[m]}\psi)$, $\mathsf{E}(\phi\mathsf{S}_{[m]}\psi) \in \mathcal{F}_{CCTL}$

We also support the temporal operators C (conditional) and S (successor). Operator C requires formula ψ to hold if ϕ was true in the previous $m-1$ steps, and operator S is a special case of operator U with $m = n$.

All interval operators can also be used with a single time-bound. In this case the lower bound is set to zero by default. If no interval is specified, the lower bound is implicitly set to zero and the upper bound is set to infinity. If the EX-operator has no time bound, it is implicitly set to one. A definition of the formal semantics of CCTL is given in [5].

Example: Signals a and b will become true simultaneously in the next 30 time steps: $\mathsf{EF}_{[30]}a \wedge b$.

3.2 FLTL

FLTL extends LTL with bounded temporal operators. The main difference however lies in the definition of the formal semantics. LTL is defined over infinite sequences, whereas FLTL is defined over finite sequences. The reason for defining FLTL over finite state sequences comes from its application in simulation for validating formal properties. A simulation run always generates only a finite trace of the system's behavior. If the simulation terminates one does still like to argue about the specification's state, i.e. if the formula holds or not. Because this predication is not always decidable with finite sequences, the definition of the formal semantics of FLTL applies a third state: *pending*. For a detailed discussion and definition of the semantics of FLTL refer to [6].

Definition 11 Let P be a set of atomic propositions, $m \in \mathbb{N}$, and $n \in \mathbb{N} \cup \{\infty\}$. The set of all syntactically correct FLTL formulas is the smallest set satisfying the following properties:

- $P \subseteq \mathcal{F}_{FLTL}$
- if $\phi, \psi \in \mathcal{F}_{FLTL}$, then also $\neg\phi, \phi \wedge \psi, \phi \vee \psi, \phi \rightarrow \psi, \phi \leftrightarrow \psi \in \mathcal{F}_{FLTL}$
- if $\phi, \psi \in \mathcal{F}_{FLTL}$, then also $\mathsf{X}_{[m]}\phi$, $\mathsf{G}_{[m,n]}\phi$, $\mathsf{F}_{[m,n]}\phi$, $\phi\mathsf{U}_{[m,n]}\psi \in \mathcal{F}_{FLTL}$.

Example: Signal a will become active for the first time at time step 300: $\neg a\mathsf{U}_{[300]}a$.

3.3 Higher level property specification

Formulating property specifications in temporal logics has turned out to be difficult even for engineers with a mathematical background. Therefore, efforts were taken to provide means of specifying properties at a higher abstraction level such that they are easier to grasp for the human reader. We integrated two approaches into our toolchain:

1. Graphical notations of properties with Live Sequence Charts (LSC).
2. Natural language property specification based on specification patterns.

The brief discussion on higher level property specification in this section will focus on these two techniques.

Live Sequence Charts Live Sequence Charts [7] were introduced to overcome the major shortcomings of Message Sequence Charts (MSC) and Sequence Diagrams (SD) from UML [8]. The criticism of MSCs and SDs concentrates on these points:

- Only an existential view of the system is supported.
- The point of activation of the chart is unclear.
- No means to specify the necessity to reach certain points in the chart.
- There is no formal semantics for SDs.

In [9], an algorithm is detailed that allows extraction of an automaton from a LSC. This automaton is then checked against a system model. Thus, LSCs are used as graphical notation for property specifications.

Natural language property specification Another idea to facilitate property specifications is to use natural language expressions and convert them into temporal logic formulas. The idea originated in the context of specification patterns [10]. These patterns classify common property specifications into categories for later reuse. In [11], a predefined grammar consisting of structured English sentences is introduced with which the user can specify properties. These will be translated into CCTL formulas thereby fixing the semantics of the structured sentences.

RT-OCL Industrial modeling mostly relies on UML. In accompanying work [12], a state-oriented real-time OCL extension (RT-OCL) was developed that allows modelers to specify state-oriented real-time constraints over UML models. The semantics of RT-OCL is described by mapping temporal OCL expressions to CCTL formulas. Thus, we have a transition path from UML to our lower level formalisms.

4 Model Checking CCTL Formulas

After presenting real-time property specifications and I/O-interval structures to describe modular timed systems, we will now explain how classical CTL model checking algorithms can be extended to cope with such systems and CCTL specifications.

4.1 Composition

Typical model checking algorithms work on one structure. The link between modular system descriptions and a monolithic structure necessary for model checking is the definition of structure *composition*. Composition defines the product structure of the original module structures. It also takes into account communication between structures. The outline of the algorithm for interval structure composition looks as follows:

1. Expand all modules and substitute their input variables by the connected outputs.
2. Compose the expanded structures.
3. Reduce the composed Kripke structure.

Composition of Kripke structures is established model checking technology. Therefore, we define the composition of interval structures by means of Kripke structures. First, we convert interval structures to equivalent Kripke structures by expansion (see figure 2). Thereafter, a reduction operation *reduce*() on Kripke structures replaces adjacent stutter states by timed edges.

The composition of I/O-interval structures where every free input is bound to an output of another module results in an interval structure, i.e., no free inputs remain. If we restrict ourselves to closed systems, then this definition can also be applied to I/O-interval structures.

4.2 Model Checking

After composition our approach allows the direct transfer of symbolic CTL model checking techniques to real-time model checking. As a consequence, the basic approach is similar to CTL model checking: building the syntax graph, computing sets of configurations (extension sets) representing subformulas of the checked property, and checking if the set of initial states is in the extension set of the complete specification.

Definition 12 Let $\Im = (P, S, S_0, T, L, I)$ be an interval structure and let ϕ be a CCTL formula. The extension set of ϕ is defined through:

$$[\phi] := \{g \in G \mid g \models \phi\} \tag{7}$$

CTL model checking algorithms can be directly used to check CCTL specifications. However, adaptations have to be made if operators carry time bounds. In this case, the fixpoint iteration is aborted if the time bound is reached. If the time bound of the operator is infinity, then the recursion has to be performed until a fixpoint is reached.

The main operation carried out during extension set computation for CCTL operators is determining predecessor configurations. This is done by taking the union of local and global predecessor computations. Local predecessor computation is restricted to configurations with non-zero clock values, whereas global predecessor computation is confined to configurations with zero clock values. The latter requires taking into account the whole transition relation.

Recursive definitions for all operators and algorithms for optimized extension set computation are given in [3]. The real-time model checking algorithms implemented in RAVEN have been formally verified in [13].

4.3 Implementation

In symbolic CTL model checking, state sets and transition relations are represented by characteristic functions which in turn are represented by reduced ordered binary decision diagrams (ROBDDs, [14]). For timed model checking state sets and relations are – due to the additional timing information – represented by extended characteristic functions which can in turn be represented by multi terminal BDDs (MTBDDs, [15]).

5 The Overall Picture

In the preceding sections we have established the necessary theoretical and technological background underlying our verification techniques. In this section we give a general overview of the tools and how they relate to each other.

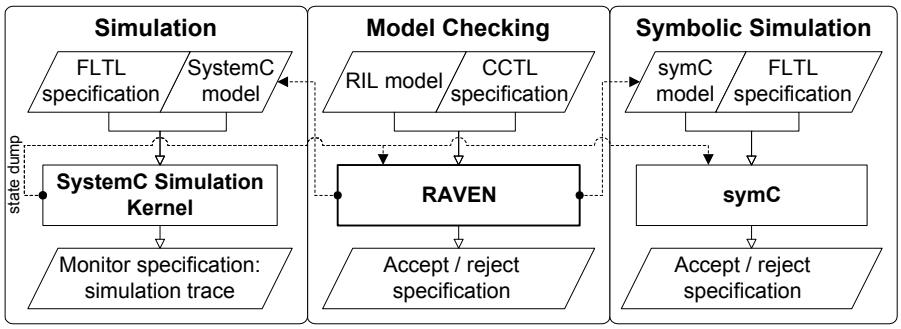

Fig. 4. Outline of the toolchain. Solid arrows show the typical flow of one tool, whereas dashed arrows represent transformations such that input for another verification technique is generated.

Fig. 4 shows the top-level outline of our toolchain. At its heart lies RAVEN (Real-Time Analysis and Verification Environment), a MTBDD- based model checker that supports CCTL property specifications. RAVEN's major invention is to allow checking discrete timing properties of the modeled system. In addition to these property checking facilities, RAVEN features algorithms to analyze system latencies and response times.

5.1 RAVEN

The system model for RAVEN is given in RIL (RAVEN Input Language, a notation for I/O-interval structures), which will be discussed in more detail in Section 6. This model description can be translated into other formats by RAVEN, i.e. the RIL model is the reference description for our other tools. We therefore provide a consistent integration of all tools with respect to one system model.

The main tasks of RAVEN after parsing the input file are the construction of the MTBDDs for each process, and composition and synthesis of the MTBDD for the system transition relation. The resulting MTBDDs are then used for checking specifications and for answering timing queries.

After composition, RAVEN can be switched to an interactive mode allowing the user to manipulate his specifications and queries and to add new ones. RAVEN supports a command line interface as well as a graphical user interface.

5.2 SystemC

On the one hand, the RIL model can be translated into a SystemC [16] model. This model can be compiled into an executable system model which is run with the SystemC simulation kernel and its associated semantics [17]. In Section 7.2, a short overview of SystemC is given.

SystemC supports traditional trace generation in various waveform formats, but in [6] a checker for FLTL formulas was presented that can be linked against SystemC code. Thus, functional verification is augmented by formal methods.

This approach can be carried even further. The formulas linked against the SystemC code can describe *critical states*. If a critical state is reached, the current system state is dumped to disk. On this snapshot of the system state, bounded model checking can be performed with RAVEN to get exhaustive property checking in local areas of the state space.

5.3 symC

On the other hand, a state transition model appropriate for the bounded property checker symC [18] can be generated. Again, properties are specified in FLTL. In contrast to a classical model checker, symC performs one forward image computation at a time, i.e. the current set of states is replaced by the set of states reachable with one transition. This results in an efficient verification for properties with large time bounds by avoiding fixpoint iterations and reachable state set computations.

However, both the SystemC checker and the bounded property checker symC can be used as standalone tools. In this case, the outcome of the design phase is a SystemC model or a symC model, respectively.

6 The Input Format RIL

We now give an informal introduction to the contents and structure of a RIL file. RIL is a simple format for specifying I/O-interval structures, property specifications as temporal logic formulas, and analysis queries. A more detailed description is available in [19], along with a complete grammar for RIL.

6.1 I/O-interval structures

Each RIL module contains one I/O-interval structure. The structures are defined as state transition graphs. The transitions are labeled with time intervals and input restrictions. Inputs are functionally connected to output variables of other modules. A RIL description of the I/O-interval structure depicted in Fig. 1 looks as follows:

```
MODULE structure
  SIGNAL  a : BOOL
  INPUT   i := m.output  // connect to output of module m
  DEFINE  s0 := !a
  INIT    s0
  TRANS   |- s0 -- !i : [1,3] --> a := true
END
```

An I/O-interval structure is introduced by the keyword `MODULE` followed by the module's name. The following sections define a module's behavior:

Signals. The definition of signals is a white space separated list of single signal definitions, which consist of a signal identifier and its associated type. Each module has to have at least one signal.

Input signals. An input definition is a Boolean formula over the signals and definitions of other modules. This is the only interface of a module to other modules, i.e. the remaining module description may only access local identifiers.

Definitions. A definition is an abbreviation of a Boolean equation by an identifier. These definitions have the same syntax as input definitions with the exception that only local identifiers may be used in the formula. Identifiers used for signals, inputs, and definitions have to be unique in a module, but different modules may use identifiers with the same name.

Initial states. These definitions are given as Boolean formulas. In the example, it is `s0`, i.e. $a = \mathit{false}$. There exists a `CHOOSE`-operator to select nondeterministically an initial value out of a set of values.

State transitions. Transitions[2] are specified with statements conforming to the following syntax:

```
|- start-state-equation -- input-condition : time-restriction
   --> signal-assignments !-> alternative-signal-assignments
```

The start state and input restrictions are both described with a Boolean formula. The formulas may use all local signals, input signals, and definitions. Time restrictions are comma separated lists of intervals or single expressions over natural numbers. All specified values are interpreted as possible delay times. There also exists the possibility to use global time constants. Operator `!->` describes alternative states if the input condition fails during the delay time. RAVEN allows mixing timed modules with fully synchronous modules. Their transition relation is defined by a conjunctively connected sequence of Boolean formulas with no timing information.

6.2 Global Definitions

Time definitions. RAVEN can only work with constant time values, but for an easier parameterization of modules with different times there exists the possibility to define global time constants which may be used in the delay time specification of the modules.

Boolean functions. Each global function is associated with an identifier. These functions can be interpreted as modules without signals and therefore without a transition relation and without states.

Property specifications. RAVEN checks each CCTL formula present in section SPEC of the RIL input file. A CCTL formula is specified in this way:

identifier := *cctl-formula*

The formulas are built upon the signals, definitions, and inputs of modules as well as the global definition names. For identifying local signal names, they are preceded by the module name and separated by a dot.

Property analysis. Property analysis allows a designer to extract characteristic time bounds from a system description. Typical problems are minimal and maximal delay times between events, e.g., how long does it take to process a workpiece in a production system. The current version of RAVEN supports three different algorithms[3]:

MIN. Requires two sets of configurations (see Definition 8): the start and the destination configurations. Then this algorithm computes the minimal delay time which is necessary to reach a configuration of the destination set starting in a configuration of the start set.

MAX. Computes the maximal delay time which may appear between a configuration of the destination set starting in a configuration of the start set.

[2] RAVEN supports a more intuitive state transition notation since version 2.

[3] The algorithms are directly derived from ideas of Campos and Clarke which used similar algorithms based on a ROBDD representations [20].

STABLE. Requires one set of configurations and computes the length of the longest path which does not leave the given set.

A set of configurations is specified via CCTL formulas. For example, the following query computes the time a machine may stay idle:

machine_idle := STABLE(¬machine.processing)

7 Verification Approaches Augmenting Model Checking

We have explained the major features of RAVEN in Section 5.1. Once we have a RIL model, we can verify it not only with RAVEN itself, but also with the other available techniques and software artifacts from our toolset as outlined in Fig. 4. Key to this procedure are RAVEN's built-in model translators. In this section we describe our verification approaches augmenting model checking.

7.1 Time Bounded Property Checking with symC

In [18] we proposed a formal verification technique for time bounded property checking. The technique performs forward image computation for state traversal, a characteristic shared by forward model checking [21][4]. Properties are specified with FLTL formulas, therefore a tight integration with other property checking tools is provided. The temporal logic formulas are converted to special finite state machines, so-called AR-automata [6], which can then be used in the symbolic execution phase.

For system description, symC uses its own input language that captures finite state machines. Such symC models can either be written by hand, or they can be generated from Verilog netlists or RIL models. The automatic transformation from RAVEN to symC input files allows us to check one model description with both verification tools. Fig. 5 shows the general operation of symC, the tool based on this approach.

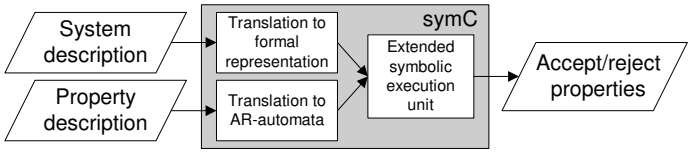

Fig. 5. Overview of symC operation.

The current implementation of symC uses a BDD-based approach, where one symbolic execution step corresponds to one forward image computation of the given state set. In contrast to standard state space traversal techniques, in

[4] However, our property checking algorithms are quite different.

this method we forget already visited states. The symbolic execution is stopped if a given time bound k is reached, or the property can either be proven correct or incorrect in the current state. The time bound k is either predefined by the user or determined by the formula if no infinite operators are used.

Both, the system description and the AR-automata, are translated to BDDs. In order to avoid the construction of the complete transition relation we use a set of transition relation partitions together forming the whole relation T.

The main iteration of our checking algorithm works in two steps. In the first step we compute the successor states of the AR-automata and we check whether a formula is accepted or rejected. In the second step of each iteration we perform one symbolic execution step on the system under inspection. During image computation we build the conjunction of all partitions on-the-fly to obtain the successor state set.

We do not build the complete state space, a feature shared with bounded model checking [22]. Rather, from a given start set we visit states reachable within a given time bound. The choice of the start set allows tuning a symC execution either towards complete coverage or towards smaller memory footprint and faster runtime. Experimental results [18] show that this approach outperforms other property checking methods for certain classes of systems and properties. This technique is well suited for properties with large time bounds.

7.2 Simulation with SystemC

Simulation and modeling with SystemC SystemC [16] is a C++ library developed to support modeling at the system level, but also at other levels of abstraction, such as register transfer level (RTL). The modeled systems may be composed both of hardware and software components. The whole library is written in ISO/ANSI compliant C++ [23] and therefore runs on all standard compliant C++ compilers. It constitutes a domain specific language embodied in the library's data types and methods.

The SystemC core language is built around an event-driven simulation kernel which allows efficient simulation of compiled SystemC models. Processes in SystemC are nonpreemptive, thus one erroneous process can deadlock the simulator. The SystemC library provides abstractions for hardware objects that allow modeling from RT up to transactional level. These abstractions include:

- Processes for modeling of simultaneously executing hardware units.
- Channels for modeling the communication of processes, as well as ports and interfaces for flexible interchangeability of channels.
- Events for modeling the interaction between processes and channels.
- Modules for modeling the structural and hierarchical composition of the described system models.
- Hardware specific data types like signals, bitvectors, and floating point numbers of fixed and variable width.
- A notion of time is supported with clock objects. Clocks generate timing signals such that events can be ordered in time.

The SystemC library and reference implementation of the simulation kernel are available for free [24] in source code. Companies are encouraged to provide Intellectual Property (IP) cores in this standardized description language.

Verification extensions for SystemC Despite the fact that we are able to handle models with symC where RAVEN is running out of memory, we still run into problems for very large designs. Here, we try to take advantage of classical simulation, but enrich it with formal methods.

The first step was to extend SystemC models with assertions expressed as temporal logic formulas [6]. Depending on the translation scheme, these assertions are either compiled into a library that is linked against the SystemC executable, or they can be added dynamically during execution. A special intermediate language [25] supports these different translation schemes. With this technique, checking executable system models against formal properties can start at the highest abstraction levels.

The methodology just mentioned is instrumental in another approach that combines functional and formal verification. Once the system model has been converted into a RIL model it can be model checked with RAVEN. However, for large designs we run into the state explosion problem. In order to still be able to perform limited checking, we support the following technique.

The RIL model is translated into an executable SystemC model and a temporal formula is checked against this model during simulation using our checker library. The formula has to conform to this structure:

$$\mathsf{AG}(critical\_state \rightarrow required\_temporal\_behavior) \tag{8}$$

Whenever the formula *critical_state* is true during simulation, the current *critical* system state is dumped to a state file.

After simulation, we use RAVEN to check the *required_temporal_behavior* against the RIL model. The dumped state files are used to set up the initial state for the model checker. The checked formula is restricted to a finite time bound t_{max}, thus we avoid the construction of the complete state space. Of course, we can now only argue about the system behavior in this time bounded state space.

Summarizing, the user guides the verification process by pointing out critical states from which local state space traversal is performed within a limited time scope. Fig. 6 shows the overall flow of this combined approach.

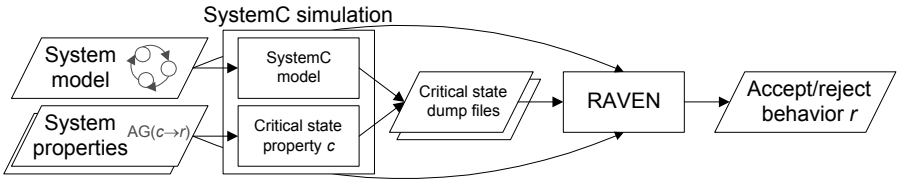

Fig. 6. Combined approach using SystemC and RAVEN.

8 Modeling Production Automation Systems in RIL

We now describe how production automation systems can be modeled with I/O-interval structures. For this purpose, a holonic production system is taken as an example.

8.1 The holonic production system

The holonic material transport system consists of an input station, three machines, an output station and three automatic transport vehicles, the so-called *holons*. Two of the three machines are for workpiece processing, one is for cleaning. All holons are identical. The task of the holons is to move workpieces to the two processing units. After processing, the workpieces have to be moved to the cleaning machine. From this station the workpieces have to be transported to the output station. Effectively, a workpiece travels from the input station to the output station, and visits all machines on the way in order. Transportation of the same workpiece can be accomplished by different holons, because a holon's task is renegotiated after it has dropped a workpiece at a unit.

8.2 RIL-model of the holonic production system

We started the modeling process of the system at a high abstraction level with a MFERT [26] description. MFERT is a language for the description of production automation systems. After analyzing this MFERT document, we have split the physical units into one or more processes. Each process is modeled by an I/O-interval structure, i.e. a RIL module. We obtained the following units: three holons, one input station, one output station, two processing machines, and one cleaning machine. From now on, we will no longer differentiate between the machines.

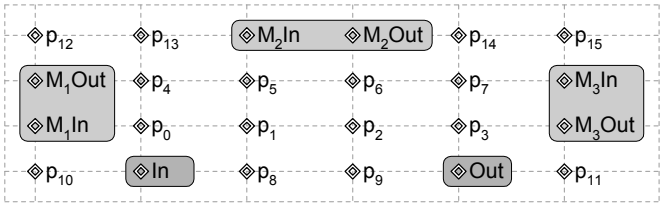

Fig. 7. Coordinate system of the holonic production system.

The units are positioned in a simple coordinate system as depicted in Fig. 7. Holons can move to all possible positions, and positions where holons interact with the machines are marked with rounded rectangles. Each machine has an associated position for receiving workpieces from holons (M_1In - M_3In), as well

as for dumping them (M_1Out - M_3Out). The input and output stations have just one position for dumping (In) and receiving (Out) workpieces, respectively.

We will now describe the behavior and structure of the identified units and how they are composed from RIL modules. All modules communicate through a common broadcast channel. A starburst in the graphical notation depicts signals broadcasted to this channel. All mentioned time delays are local to the module, and given in clock ticks of the modeled system.

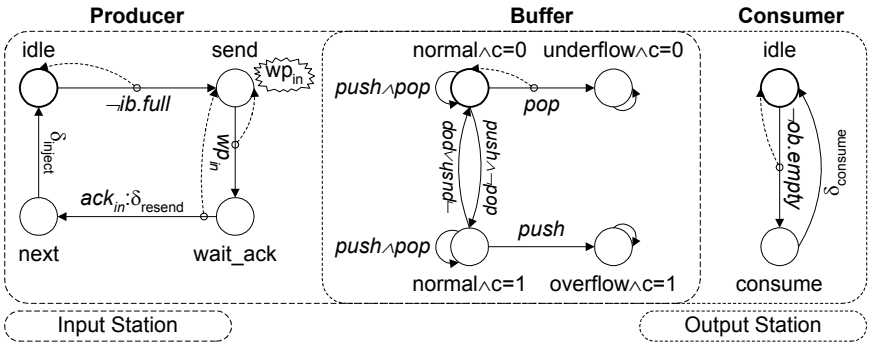

Fig. 8. Modules for input and output stations. The input station consists of producer and buffer, whereas the output station is made up of buffer and consumer.

Input station. We model the input station such that it never runs out of workpieces. It consists of two modules: a producer and a buffer.

The producer generates a raw workpiece if the buffer is not full. It then broadcasts signal wp_{in} and waits for the signal to appear on the broadcast channel. Then the producer sends the workpiece to the station's buffer whenever a holon acknowledges interest in retrieval with signal ack_{in}. If the acknowledgment is not given within δ_{resend} clock ticks, the sending procedure is repeated. Otherwise, the producer waits δ_{inject} ticks and the cycle starts anew. The producer's I/O-interval structure is given in Fig. 8.

Items are pushed from the producer to the buffer and from there they are retrieved by holons. Similar buffers will be used in other modules, the only difference is the direction of the push and pop operations. Pushing adds and popping removes one item from the buffer. The buffer can be in state *normal*, *underflow*, or *overflow*. Counter c holds the number of workpieces stored, with a fixed capacity of buf_{size} items. In Fig. 8, a buffer capable of holding one workpiece is illustrated.

Output station. The output station is modeled very similarly to the input station. However, the producer module is replaced by a consumer module. The consumer removes finished workpieces from the output buffer if the buffer is not empty. After a delay of $\delta_{consume}$ ticks the consumer is ready to retrieve another workpiece. The output buffer works exactly as the input buffer described

above, with the exception that workpieces are pushed to the buffer by holons and popped from it by the consumer. The consumer is depicted in Fig. 8.

Machines. Machines are composed of five modules. Each machine has an associated input (ib) and output buffer (ob) where holons can drop or pick up workpieces. They also have an internal buffer (mb) for holding workpieces. Finally, two modules control the machines communication behavior and the transportation of workpieces between the buffers (see Fig. 9).

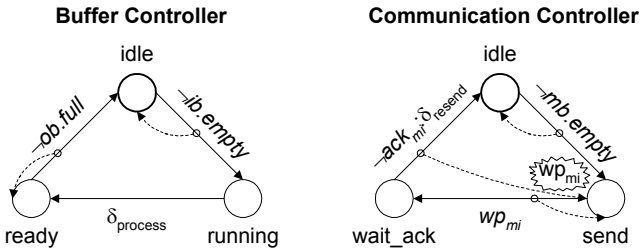

Fig. 9. Modules of machine controllers. The buffer modules are omitted.

The buffer controller first checks for a workpiece in input buffer ib and changes to state *running* if ib is not empty. After a delay of $\delta_{process}$ ticks it transitions to state *ready*. If output buffer ob is not full the controller can finally become *idle* again.

The communication controller first polls for a nonempty machine buffer mb and then goes to state *send*. There it broadcasts signal wp_{mi} to request a holon. If this signal appears on the broadcast channel, the controller waits for a holon to acknowledge the request within δ_{resend} clock ticks.

A workpiece moves through a machine's buffers in this order: First, a holon has to drop a workpiece at the machine's input buffer from where it has to be transported to the machine's internal buffer. After processing, it is finally stored in the output buffer where a holon has to pick it up. The buffers follow the same scheme as shown in Fig. 8.

Holons. A holon's state is encoded with the following variables:

- The task currently assigned to the holon (s).
- Its position (*pos*, see Fig. 7) and orientation (*dir*: left, right, up, down).
- Flags denoting if the holon is currently *moving* or *finishing* a rotation[5].

The large amount of states (82) prohibits the module's complete graphical presentation. Therefore, we explain the module's main characteristics informally. A holon basically behaves as follows:

A holon starts by waiting for orders. Now, one of the machines or the input station can request retrieval of a workpiece at its output buffer by broadcasting

[5] Holons can move forward one step at a time, and they can be rotated by 90°.

to the communication channel (wp_x). All holons waiting for orders answer by sending their distance to the requesting unit. The signal of the nearest holon prevails[6], and this holon acknowledges the request (ack_x). It follows a predetermined path to the unit and retrieves the workpiece. The holon's task is updated to transporting the item to the input buffer of the next unit, e.g. after retrieving a workpiece from the last machine, the item is transported to the output station. If the buffer is full, the holon waits for an empty slot. Then, it is free to wait for new orders.

The bulk of a holon's module consists of the path finding algorithm. Basically, whenever a holon is assigned the task to move to some machine and pick up or put down a workpiece there, the same procedure is applied. Depending on the current position and orientation, a holon either turns into the right direction, or, if the direction is correct, it moves one step forward after the specified move delay time. If a holon finds its way blocked – a holon checks if another holon occupies the position one step ahead and one step ahead to the right – it rotates and tries an alternative path to its destination. This procedure is repeated until the holon arrives at the destination and performs the requested operation. Thus, for every coordinate, a path to all machines is encoded in these transitions. The path finding algorithm provides basic collision protection, but no special means are taken to prevent deadlocks.

9 Experimental Results

The model of the holonic production system described in Section 8 is the basis for the results reported here. We have created three variants of the model for our experiments. They differ only in the number of holons present in the system. Currently, we have models with one (M_1), two (M_2) and three holons (M_3).

9.1 Checked properties

We perform two measurements with the applicable tools and report the resulting runtimes. The first property P_1 is intended to give a general idea of how the tools compare to each other, whereas the second property P_2 tests a real life requirement of the model.

P_1 This property simply checks if the output station Out will eventually consume a workpiece, i.e. a workpiece was processed by all machines and the holons provided the necessary transportation between the stations. We check for the occurrence of this event within n time steps, both existentially and universally:

$$\mathsf{EF}[n](\text{OutConsumer.s} == \text{OutConsumer.consume}) \tag{9}$$

$$\mathsf{AF}[n](\text{OutConsumer.s} == \text{OutConsumer.consume}) \tag{10}$$

[6] Distances are given in chessboard metric.

***P*$_2$** The holonic production system does not contain explicit collision freeness by design. Therefore, it is crucial to check the system model for this property, i.e. the model checker has to prove that the system's behavior implicitly models collision free operation. We check that whenever a holon is blocked, in the next time step after the move delay time (mt) holons do not occupy the same position:

$$\begin{aligned}&\mathsf{AG}(\ (\text{h1.blocked} \vee \text{h2.blocked} \vee \text{h3.blocked}) \rightarrow \mathsf{AG}[mt+1](\\ &\quad (\text{h1.pos} \mathrel{!}= \text{h2.pos}) \wedge (\text{h1.pos} \mathrel{!}= \text{h3.pos}) \wedge (\text{h2.pos} \mathrel{!}= \text{h3.pos})\ ))\end{aligned} \tag{11}$$

9.2 Measurements

All measurements were conducted on a Linux PC with a 2.8 GHz Pentium 4 processor and 1 GB of RAM installed.

We used this configuration for the experiments: $\delta_{move} = 4$, $\delta_{rotate} = 1$, $c = 1$ for the input and output station's buffer, $c = 2$ for the machines' input and output buffers, $\delta_{inject} = 4$, $\delta_{resend} = 2$, $\delta_{process} = 3$, and $\delta_{consume} = 1$.

***P*$_1$** Checking property P_1 only makes sense with RAVEN and symC because Formulas (9) and (10) do not conform to the structure required by the SystemC approach (see Formula (8)). We report the result for models M_1, M_2, and M_3 with different time bounds n in Table 1.

time bound	10	50	100	200	500	∞
RAVEN						
M_1, existentially	57.44 (f)	57.85 (f)	58.31 (f)	58.89 (t)	59.03 (t)	62.31 (t)
M_1, universally	57.92 (f)	58.07 (f)	58.37 (f)	59.0 (f)	59.22 (f)	62.1 (f)
M_2	measurement terminated after one hour ($mtaoh$)					
M_3	$mtaoh$					
symC						
M_1, existentially	0.03 (f)	0.39 (f)	2.16 (f)	3.36 (t)	3.45 (t)	n/a
M_1, universally	0.03 (f)	0.38 (f)	2.14 (f)	22.55 (f)	296.12 (f)	n/a
M_2, existentially	0.07 (f)	0.76 (f)	7.32 (f)	150.86 (t)	151.1 (t)	n/a
M_2, universally	0.08 (f)	0.77 (f)	7.31 (f)	1540.78 (f)	$mtaoh$ (f)	n/a
M_3, existentially	0.25 (f)	1.29 (f)	17.23 (f)	979.36 (t)	972.12 (t)	n/a
M_3, universally	0.26 (f)	1.22 (f)	16.77 (f)	$mtaoh$ (f)	$mtaoh$ (f)	n/a

Table 1. Results for checking P_1 using RAVEN and symC with different time bounds. Times are given in seconds, followed by an indicator if P_1 is true (t) or false (f).

We see that RAVEN is unable to check properties for models with more than one holon in the system. RAVEN runs out of memory in the reduction phase of I/O-interval structure composition (see Section 4.1), because the model becomes too large. On the other hand, once all preparations for model composition have finished, the actual model checking does not vary significantly depending on time bound n.

Global definitions in the RIL model allow configuring buffer capacities and time delays of the system. Some of these configurations may produce systems

that contain paths which do not consume workpieces. The configuration used in the experiments produces such an unreliable system. This can be observed in the results for checking P_1 universally.

We are able to check properties with symC against all three models within given time bounds. In most cases, symC outperforms RAVEN. However, once symC has to traverse a huge state space for formulas that are still pending RAVEN may overtake again, e.g. universal quantification with a time bound of 500 steps.

***P*$_2$** For property P_2, we restricted the tests to symC and the combined SystemC/RAVEN approach. RAVEN was unable to check models with more than one holon, but for collision detection in P_2 we need at least two holons in the system.

Checking P_2 with SystemC is performed in the three-step procedure described in Section 7.2. First, the RIL model is translated into SystemC model. The model is simulated and our checker library checks the critical state condition. If the condition evaluates to *true*, the current system state is dumped. Finally, RAVEN performs a time bounded check of the required system behavior with all dumped states used as initial states. In our experiment, the local state space traversal is limited to 7 steps, because we are only interested in collisions directly after the blocking of at least one holon. In Table 2, we report the time for simulating 10000 time steps of the SystemC model, the number of critical states collected, and the time for model checking the local state space, both for M_2 and M_3.

	simulation time	nr. of critical states	model checking
M_2	4.6 sec.	161	7.24 sec. (t)
M_3	83.3 sec.	9685	16.45 sec. (t)

Table 2. Results for checking P_2 using SystemC (model running for 10000 cycles) and RAVEN.

To check P_2 with symC, we universally test for the required temporal behavior from the initial state. Thus, the comparison with the SystemC approach does not completely match. As we see in Table 3, we can also check the property with symC, however we get complete coverage within this range. For model M_3, symC even beats the combined SystemC and RAVEN approach.

The results for checking P_2 are always true with both verification approaches, which means that for the observed simulation runs we can guarantee collision freeness.

	1000	2500	5000	10000
M_2, universally	1.33 sec. (t)	3.11 sec. (t)	9.02 sec. (t)	32.76 sec. (t)
M_3, universally	7.88 sec. (t)	9.85 sec. (t)	15.43 sec. (t)	39.47 sec. (t)

Table 3. Results for checking P_2 using symC for different time bounds.

The experiments show that depending on the checked properties and the requirements on the checks, different verification tools excel in different areas. Model checking gives the user complete coverage of the model, however it suffers from the state explosion problem. Here, the other approaches can help to validate properties. An important task of the verification engineer is to select the appropriate tool to handle a specific verification problem.

9.3 Comparison with other tools

Both RAVEN and symC were compared with other tools. However, a fair assessment is very difficult to achieve. Each tool has its strengths and weaknesses, its own semantics of the time model, its own execution model and communication scheme. In addition, the tools have specification languages with differing expressiveness.

In [3] various model checkers with support for explicit time representation were benchmarked with simple but scalable examples. The measurements showed that RAVEN is slower for smaller models, but can beat all other tools when the model's size reaches a certain threshold. Among the competing model checkers were UPPAAL [27], KRONOS [28] and Verus [29].

10 Conclusions and Future Work

We have presented a toolbox for modeling and verifying real-time systems with a special focus on production automation systems. For this purpose, we devised a formal model of timed systems. The model is represented by an interval structure, which can be composed of multiple components, the I/O-interval structures. We also introduced the time bounded temporal logics CCTL and FLTL. They allow property specifications based on linear or branching time, respectively. The detailed model of a holonic material transportation system was presented in order to show how the different software assets available in our toolchain can be applied.

Central to our various verification approaches is RAVEN, a real-time model checker. A RIL model can be translated into other models with RAVEN such that the other tools can be applied where RAVEN meets its limitations. These tools are:

- The bounded property checker symC. Time bounded properties specified in FLTL are verified against a system model encoded as a finite state machine. An exhaustive state space traversal is avoided.
- A checker library for checking FLTL formulas against executable SystemC models.
- Localized property checking with RAVEN. A critical state detection mechanism based on our checker library dumps critical state files from simulated SystemC models. These files initialize the state for model checking with RAVEN with a fixed time bound.

The experiments performed on the model of the holonic material transportation system stress the fact that not one verification technique alone suffices to meet the demands of an efficient and complete system validation process. Critical system components require a fully formal proof of correctness, whereas in other areas a semi-formal approach combining both formal and functional verification is the only way to handle large designs.

Currently, we are working on enhancing symC. The dynamic nature of symC allows guiding of the symbolic execution process, which might help in finding error states or proofs more efficiently, i.e. parts of the state space will not be traversed at all.

Furthermore, we are investigating possibilities to provide an integrated verification environment that handles all parts of our toolbox and helps the verification engineer to coordinate his efforts.

References

1. Clarke, E.M., Grumberg, O., Peled, D.E.: Model Checking. The MIT Press (1999)
2. Vardi, M.Y.: Branching vs. linear time: Final showdown. In: European Joint Conferences on Theory and Practice of Software (ETAPS 2001). (2001) Invited paper.
3. Ruf, J., Kropf, T.: Symbolic verification and analysis of discrete timed systems. Journal on Formal Methods in System Design **23(1)** (2003) 67–108
4. Emerson, E.A., Mok, A.K., Sistla, A.P., Srinivasan, J.: Quantitative temporal reasoning. In Clarke, E.M., Kurshan, R.P., eds.: Computer Aided Verification, 2nd International Workshop. Volume 531 of Lecture Notes in Computer Science., Springer (1991) 136–145
5. Ruf, J., Kropf, T.: Modeling and checking networks of communicating real-time process. In Pierre, L., Kropf, T., eds.: Correct Hardware Design and Verification Methods. Volume 1703 of Lecture Notes in Computer Science., Springer (1999) 256–279
6. Ruf, J., Hoffmann, D.W., Kropf, T., Rosenstiel, W.: Simulation-guided property checking based on a multi-valued AR-automata. [30] 742–748
7. Damm, W., Harel, D.: LSCs: Breathing life into message sequence charts. Journal on Formal Methods in System Design **19(1)** (2001) 45–80
8. Object Management Group (OMG): Unified Modeling Language (UML), Version 1.5. www.omg.org (2003) Document formal/03-03-01.
9. Klose, J., Kropf, T., Ruf, J.: A visual approach to validating system level designs. In: 15th International Symposium on Systems Synthesis, ACM Press (2002) 186–191
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21. International Conference on Software Engineering, ACM Press (1999) 411–420
11. Flake, S., Müller, W., Ruf, J.: Structured english for model checking specification. In: Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen. 3. GI/ITG/GMM Workshop, VDE Verlag (2002) 99–108
12. Flake, S., Müller, W., Ruf, J.: A UML/OCL extension for state-oriented temporal properties with applications for manufacturing systems. (2004) This volume.

13. Reif, W., Schellhorn, G., Vollmer, T., Ruf, J.: Correctness of efficient real-time model checking. Journal of Universal Computer Science, Special Issue on Tools for System Design and Verification **7(2)** (2001) 194–209
14. Bryant, R.E.: Symbolic boolean manipulation with ordered binary-decision diagrams. ACM Computing Surveys **24(3)** (1992) 293–318
15. Bahar, R.I., Frohm, E.A., Gaona, C.M., Hachtel, G.D., Macii, E., Pardo, A., Somenzi, F.: Algebraic decision diagrams and their applications. In: Proceedings of the 1993 IEEE/ACM International Conference on CAD, IEEE Computer Society Press (1993) 188–191
16. Grötker, T., Liao, S., Martin, G., Swan, S.: System Design with SystemC. Kluwer Academic Publishers (2002)
17. Müller, W., Ruf, J., Hoffmann, D.W., Gerlach, J., Kropf, T., Rosenstiel, W.: The simulation semantics of SystemC. [30] 64–70
18. Ruf, J., Peranandam, P.M., Kropf, T., Rosenstiel, W.: Bounded property checking with symbolic simulation. In: Forum on Specification and Design Languages. (2003)
19. Ruf, J.: RAVEN: Real-time analyzing and verification. Technical Report WSI 2000-3, University of Tübingen (2000)
20. Campos, S.V., Clarke, E.M.: Real-time symbolic model checking for discrete time models. In Rus, T., Rattray, C., eds.: Theories and Experiences for Real-Time System Development. Volume 2 of Amast Series In Computing. World Scientific Publishing Corporation, Inc., River Edge, NJ, USA (1994) 129–145
21. Iwashita, H., Nakata, T.: Forward model checking techniques oriented to buggy designs. In: Proceedings of the 1997 IEEE/ACM International Conference on CAD, ACM and IEEE Computer Society Press (1997) 400–4004
22. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. In Zelkowitz, M., ed.: Highly Dependable Software. Volume 58 of Advances in Computers. Academic Press (2003)
23. ISO/IEC: Programming Languages – C++. 2. edn. Number 14882:2003 in JTC1/SC22 – Programming languages, their environment and system software interfaces. International Organization for Standardization (2003)
24. VA Software Corporation, Open SystemC Initiative: Open SystemC Initiative. www.systemc.org (2004)
25. Krebs, A., Ruf, J.: Optimized temporal logic compilation. Journal of Universal Computer Science, Special Issue on Tools for System Design and Verification **9(2)** (2003) 120–137
26. Flake, S., Müller, W.: A UML profile for MFERT. Technical Report 4, C-LAB Paderborn (2002)
27. Bengtsson, J., Larsen, K.G., Larsson, F., Pettersson, P., Yi, W.: UPPAAL - a tool suite for automatic verification of real-time systems. In Alur, R., Henzinger, T.A., Sontag, E.D., eds.: Hybrid Systems III: Verification and Control, Springer (1996) 232–243
28. Yovine, S.: KRONOS: A verification tool for real-time systems. International Journal on Software Tools for Technology Transfer (STTT) **1 (1-2)** (1997) 123–133
29. Campos, S.V.A., Clarke, E.M., Minea, M.: The Verus tool: A quantitative approach to the formal verification of real-time systems. In Grumberg, O., ed.: Computer Aided Verification, 9th International Conference. Volume 1254 of Lecture Notes in Computer Science., Springer (1997) 452–455
30. Nebel, W., Jerraya, A., eds.: Design, Automation and Test in Europe, DATE 2001. In Nebel, W., Jerraya, A., eds.: Design, Automation and Test in Europe, DATE 2001, IEEE Press (2001)