Visual Specifications for Modular Reasoning about Asynchronous Systems

Nina Amla[1], E. Allen Emerson[2], Kedar S. Namjoshi[3], and Richard J. Trefler[4]

[1] Cadence Design Systems
[2] Department of Computer Sciences, University of Texas at Austin‡
[3] Bell Laboratories, Lucent Technologies
[4] School of Computer Science, University of Waterloo

Abstract. We propose a framework that closely ties together visual specification and modular reasoning of asynchronous systems. The basis of the framework is a new notation, called Modular Timing Diagrams (MTD's), for specifying the *universal* properties about causality and timing of events in an asynchronous system. MTD's are *complementary* in nature to Message Sequence Charts, that are typically used to specify *existential* properties. Our framework includes two algorithms for formal reasoning with MTD's. The first is an efficient polynomial-time model checking algorithm. The second is an algorithm for automatically generating an assume-guarantee partitioning of an MTD, that exploits its inherent decompositional structure. We show how to use this decomposition for modular reasoning with MTD properties in conjunction with an asynchronous compositional reasoning rule. To illustrate the notation and our method, we describe a case study where we specified telephony features, such as call forwarding with MTD's, and verified these properties on an asynchronous telephony model. The compositional reasoning methods led to savings of 15%-80% in verification times, and comparable savings in space.

1 Introduction

Visual specification formalisms like Message Sequence Charts (MSC's) [MSC96] and their extensions are widely used to specify the high-level behavior of asynchronous processes. MSC specifications describe partially ordered scenarios of a system and they are *existential* in nature. In this paper, we present a complementary visual notation for specifying properties that are *universal* in nature; that is, properties that should hold for *every* computation of a system. This notation, called Modular Timing Diagrams (MTD's), is an extension of the Timing Diagrams (TD's) notation, that is widely used in the hardware industry to specify universal properties of hardware protocols [DJS94,Fis96,AEN99]. MTD's form the basis of our proposed framework for specifying and analyzing properties of asynchronous systems in a modular manner.

‡ Supported in part by NSF CCR 009-8141 and TARP 003658-0650-1999.

Timing Diagrams are used to describe timing and ordering properties over events (that is changes of value) of signals; an example timing diagram is shown on the right half of Figure 3. In the course of our prior work in formalizing timing diagrams [AEN99,AEKN00], we realized that while TD's are an intuitive notation, they also have some limitations: one cannot express either disjunction or iteration, making it difficult to specify some properties. Our new notation overcomes these limitations, while retaining the intuitive and visual nature of timing diagrams. An MTD is built out of asynchronous TD modules, that are linked together by constructs for forking (conjunction), deterministic choice, and iteration. We illustrate the practical utility of MTD's with example specifications of telephony properties. We also show that MTD's, unlike TD's, are expressive enough to describe any ω-regular property over events.

As part of the framework, we present efficient algorithms for formal analysis with MTD's. We give an efficient translation of an MTD to a Büchi automaton describing the *negated* MTD property, which results in a model checking algorithm of complexity $O(m * n * k^3)$, where m is the size of the program, n is the number of TD nodes in the MTD, and k is the largest node size. For untimed MTD's (those with only ordering constraints), this Büchi automaton expresses a stuttering-closed language. This property can be exploited by model checkers, such as SPIN [Hol97], that are optimized to handle stuttering-closed properties [HP94].

An asynchronous program is usually composed of several concurrent, interacting components. In this situation, the state space of the program is exponential in its size, and is often too large for effective analysis. This well-known "state explosion" problem is one of the major barriers to the application of model checking in practice. A solution to this problem can be found in methods for *compositional reasoning* [dRLP97,dRdBH$^+$01], that utilize a decomposition of the global specification into local, assume-guarantee checks for the individual components. While this type of reasoning has been applied quite successfully in practice, it requires a certain degree of manual effort. In particular, the decomposition of the specification is usually done by hand since it is difficult to automatically decompose specifications written using temporal logic or automata. In this respect, MTD's offer a key advantage: they can be syntactically partitioned into a collection of assume-guarantee type properties about individual components of the system. We show in this paper how to perform such a partitioning, *fully automatically*, and how to utilize the resulting assume-guarantee properties for compositional reasoning.

In order to validate both the notation and our method, we applied it to the analysis of an asynchronous telephony model consisting of a central switch and several users. We chose this particular case study for a number of reasons. First, the model is highly asynchronous and nondeterministic in nature. Second, there is significant prior work in model checking such systems (e.g., [SHE01,PR99]), which gave us a reasonable way to gauge the utility of MTD's. Finally, the inherent modularity in these systems made the application of compositional reasoning an attractive option. MTD's were used to specify properties of telephony features such as call forwarding; this turned out to be quite an easy task. Our experience with the compositional reasoning method was encouraging – the partitioning algorithm worked very well, saving 15%-80% in the model checking times, and yielding smaller, yet significant reductions in space.

Related Work.

Message Sequence Charts are a standardized and popular way of visually describing scenarios in communication protocols. MSC's are well-studied and occur in many flavors with different semantics [LL94]. They have been extended to MSC-graphs (with the addition of choice and iteration), and to High-level MSC's (by adding more structure and abstractions) [MSC96]. Model checking algorithms for MSC's focus on either checking that an MSC-graph satisfies a temporal property [AY99], or on matching the executions of two MSC-graphs [MPS98,MP00]. This is because, as mentioned earlier, MSC's describe high level implementation scenarios. They, therefore, specify "lower bounds" on implementations. In contrast, MTD's specify "upper bounds"– properties that must hold of *all* computations of an implementation. Thus, MTD's are indeed *complementary* to MSC's (and the MSC variants) in terms of their intended usage. These observations also apply to a shared variable version of MSC's called Shared Variables Interaction Diagrams [AG01].

MTD's have several advantages over other graphical notations such as Live Sequence Charts (LSC's) [DH01] and Timeline [SHE01], in terms of expressiveness and efficient formal analysis. Live Sequence Charts (LSC's) are a modification of MSC's that allow specification of universal properties. LSC's use parts of an MSC to serve as an activation condition, while the remainder is a property that must hold of the system. Thus, an LSC is very close to a single timing diagram, which also has a precondition and a postcondition part. A key difference is the complexity of the model checking procedures. The model checking procedure described in [KW01] has high worst-case complexity, due to the potential exponential blowup in the conversion from a partially ordered LSC to a timed Büchi automaton that works on consistent cuts of the LSC partial order. In contrast, although MTD's allow partial ordering of events, our translation goes directly to an automaton with a small polynomial blowup. Furthermore, there does not appear to be an LSC counterpart to our automated partitioning procedure for compositional reasoning, which is essential for checking properties of large systems.

The Timeline notation [SHE01] has also been used to specify universal properties of asynchronous systems. It is limited, however, to describing a single totally ordered sequence of events. A nice feature of the notation is the ability to interleave pre- and post- conditions on a timeline; we have incorporated this in MTD's. Indeed, an MTD may be viewed as a compact form of a set of timelines.

In our previous work on a *synchronous* version of Timing Diagrams (SRTD's) we presented algorithms for efficient model checking and assume-guarantee partitioning [AEKN00,AENT01]. In this paper, we adopt a similar approach, but show how to apply it to a more general notation and to the different domain of asynchronous processes. The encouraging experimental results for the new framework lead us to believe that it is flexible, and holds great promise for practical applications.

In summary, we believe that this paper proposes a new and interesting framework for combining visual specifications with modular reasoning about asynchronous systems. By closely tying together these two aspects, we derive several advantages for the MTD framework:

- We make formal and visually explicit the informal combination of timing diagrams for individual modules.
- We include a polynomial-time model checking algorithm for checking MTD properties.
- The framework is supported by automated compositional reasoning methods to ameliorate the state explosion problem, that have performed well on examples.
- We generalize and improve upon existing proposals, such as Timing Diagrams, Live Sequence Charts, and Timeline.
- The framework complements the scenario-based notation of Message Sequence Charts.

2 Modular Timing Diagrams

We illustrate the MTD syntax and semantics through the Request-Grant property shown in Figure 1. Each node in an MTD contains a Regular Timing Diagram (RTD) [AEN99]. Section 2.1 presents the formal definitions of both MTD's and RTD's. The dotted nodes on the left are *precondition* nodes, while the solid nodes on the right are *postcondition* nodes. The filled node is a special shorthand indicating an initial, precondition node containing an empty RTD. The $\vee$ symbol indicates deterministic choice, while the $\wedge$ symbol indicates forking (conjunction). Each non-empty RTD shows a number of waveforms describing the change of values for various signals. These changes (events) are linked together with *dependencies* to form a pattern of event occurrences. The arrow from `res.reply` to `usr.req` indicates that `usr.req` must change value at least 1 clock unit after `res.reply` changes its value. We assume that timing properties are taken relative to timers, that are atomic propositions in the program. Some of the nodes of an MTD may be designated as being *fair*, which is indicated visually by placing a "*" above the node.

Informally, an MTD acts as follows. MTD checks are begun at the initial nodes. If the current node is a precondition then its successor nodes are checked only if the current pattern holds. Thus, a precondition node *may* not hold. On the other hand, if the current node is a postcondition, its pattern *must* hold. The set of successors is determined by the connector associated with the current node. An $\vee$-connector, as in this case, is resolved by picking the unique successor whose guard expression (labeling the edge between nodes) holds at the final state of the current node. An $\wedge$-connector indicates that all successor nodes must be checked. We sometimes omit showing the connector when there is a single successor; the connector is assumed to be an $\wedge$-connector. This checking process begins at all

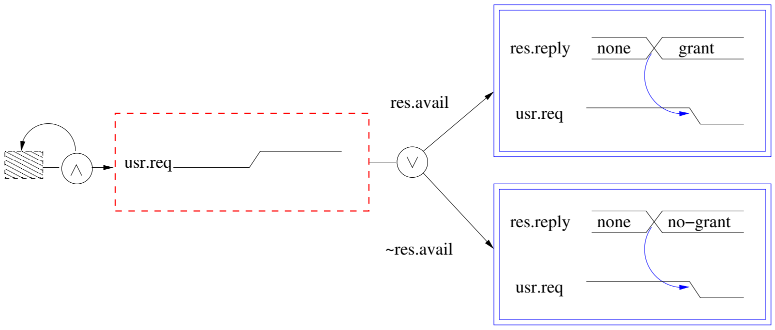

Fig. 1. A Request-Grant Property

initial nodes of the MTD, and fails only if some postcondition fails to hold, or if the fairness property of the MTD fails to hold.

In the precondition of the MTD in Figure 1, a user process, **usr**, requests a resource by setting the value of boolean variable **usr.req** to *true*. The postcondition specifies the two possible outcomes. First, if the resource is available (**res.avail** is *true*) then the variable **res.reply** is set to the value **grant**. After receiving **res.reply**, **usr** resets **usr.req**. In the second postcondition, the resource is not available and the variable **res.reply** is assigned to the value **no-grant**. The $\wedge$-connector at the initial node ensures that this check is enabled at every point along a computation – the MTD "forks off" a new check at each step due to the loop through the connector.

This is a good example of how MTD's allow sharing of patterns. In other notations such as timing diagrams or Timeline, this property would have to be split into two separate properties that duplicate the precondition pattern. For more complex properties with multiple pre- and post-condition nodes, such splitting results in a large amount of replication, which can have unfortunate consequences for understanding and maintenance of specifications.

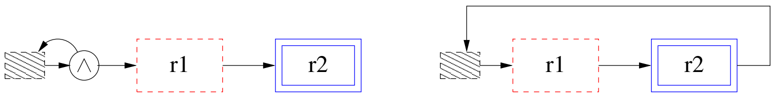

Fig. 2. Overlapping (left) and Non-overlapping (right) Semantics

There are two common semantics of timing diagrams [Fis96,AEKN00]: *overlapping* semantics, where the precondition of a diagram is checked at every point on a computation, and *non-overlapping*, where the precondition is checked at every point until it holds, following which the checking is suspended while the postcondition pattern is being checked. In Figure 2, we show how these styles are represented with MTD's. Notice how the loop through the $\wedge$-connector on the left enforces the overlapping semantics, and how the loop through the post-node on the right ensures the non-overlapping semantics.

In the rest of this section, we define the syntax and semantics of MTD's precisely, and describe the efficient model checking algorithm. First, we will define some terms that we will use in the sequel. There are a number of different notions of acceptance used with automata on infinite strings. A Büchi acceptance condition states that F holds infinitely often (expressed in Linear Temporal Logic (LTL) as $\mathsf{GF}(F)$) and a co-Büchi acceptance specifies that F holds from some point onwards (written in LTL as $\mathsf{FG}(F)$). One may also express an acceptance condition as a disjunction of co-Büchi and Büchi conditions [EL87]. Informally, a safety property specifies that something "bad" never happens during an execution, while a liveness property states that something "good" eventually happens. We use $\mathbf{N}$ to represent the natural numbers.

2.1 Syntax

An MTD, T, is specified over a set of variables (sometimes called "signals"), each with an associated domain of values. T specifies the ordering of events (usually, these are changes in value of the variables) and timed dependencies between events. Syntactically, an *event* is a pair (s, i), where s is a variable name with an associated domain of values $\mathcal{D}_s$, and $i \in \mathbf{N}$ is the position of the event. We use $v(s, i)$ to refer to the value of the event (s, i). An MTD may be viewed as a graph with nodes that are associated with regular timing diagrams (RTD's); the picture in Figure 3 shows an example MTD on the left, and a component RTD on the right.

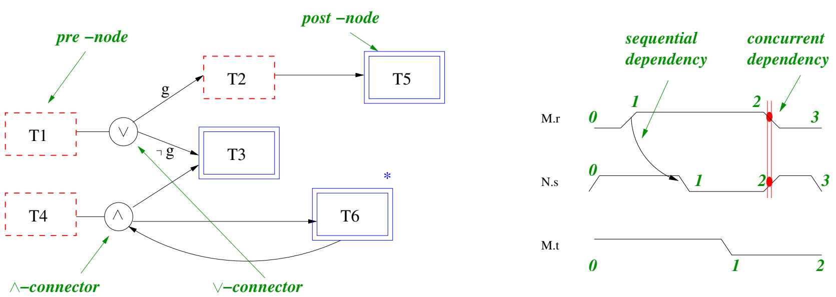

Fig. 3. A Modular Timing Diagram and a component RTD

A *regular timing diagram* (cf. [AEN99]) is a tuple $(S, \mathcal{E}, SD, CD)$, where

- S is a finite set of variable names.
- For each $s \in S$ there is a finite set of events, $\mathcal{E}(s)$, represented as $\{(s, 0), \ldots, (s, n_s - 1)\}$, called the *waveform* for s. The set of all events $\mathcal{E}$, is $\cup_{s \in S} \mathcal{E}(s)$.

- SD is the set of *sequential* dependencies on the events of $\mathcal{E}$. Each dependency is specified as $(s,i) \xrightarrow{c} (t,j)$. The timing constraint for the dependency, c, has the form $(clock, [a,b\rangle)$, where *clock* is an atomic proposition that acts as a timer, $a \in \mathbf{N}$, $b \in \mathbf{N} \cup \{\infty\}$, and $1 \leq a \leq b$.
- CD is a collection of mutually disjoint sets of events. Each set is called a *concurrent dependency.*

We refer to the components of an RTD r as S_r, $\mathcal{E}_r$, SD_r and CD_r, respectively. The *size* of an RTD r, $|r|$, is $|\mathcal{E}_r| + |SD_r| + |CD_r|$. An event $(s, i+1)$ is a *change event* if $v(s,i+1) \neq v(s,i)$. An RTD is *well-formed* if (a) the source of each sequential dependency is a change event, and the destination is either a change event or a concurrent dependency, (b) each concurrent dependency contains at least one change event, (c) the set of dependencies of the RTD (not considering the mutual concurrent dependencies) is acyclic, and (d) any non-empty RTD contains at least two events per signal. In the sequel, we consider only well-formed RTD's. For a non-empty RTD, the set $\{(s, n_s - 1) \mid s \in S\}$ of the final events of each signal forms a pre-defined concurrent dependency, one of whose change events is designated as the *final* event.

For example, the RTD in Figure 3 has three variables: $M.r, N.s$ and $M.t$. There is a sequential dependency from $(M.r, 1)$ to $(N.s, 1)$, and a concurrent dependency containing $(M.r, 2)$ and $(N.s, 2)$.

Modular Timing Diagrams are built by composing together RTD's. A *modular timing diagram* is defined by a tuple (N, C, I, F), where

- N is a finite set of *nodes*, partitioned into N_{pre}, the set of *pre-nodes*, and N_{post}, the set of *post-nodes.* Each node is labeled with an RTD.
- C is a set of *connectors.* An $\wedge$-connector is an element of $N \times 2^N$, while an $\vee$-connector is an element of $N \times 2^{G \times N}$, where G is a set of *guards*, which are boolean-typed expressions over the variables of the MTD.
- $I \subseteq N_{pre}$ is a set of *initial* nodes.
- $F \subseteq N$ is a set of *fair* nodes, defining a co-Büchi acceptance condition.

The $\wedge$-connectors indicate forking, $\vee$-connectors indicate deterministic choice and iteration is allowed by looping in the MTD graph. Given a connector $c = (r, R)$, we say that c is an *outgoing* connector from r. The size of an $\wedge$-connector (r, R) is $|R|$; the size of an $\vee$-connector $(r, \{g_i, r_i\})$ is the sum of the lengths of the formulas defining the guards g_i. The *size* of an MTD is the sum of the sizes of its component RTD's and the sizes of its connectors. An MTD is *well-formed* if (a) for each pre-node RTD, its dependency graph is a total order, (b) every pre-node has one outgoing connector, and every post node has at most one outgoing connector, (c) for every $\vee$-connector, the guards are mutually disjoint and complete (that is, $g_i \wedge g_j \equiv \mathit{false}$ for $i \neq j$, and $\bigvee_i g_i = \mathit{true}$). We use N_{term} for the subset of post-nodes, called *terminal* nodes, that have no associated outgoing connector. In the sequel, we consider only well-formed MTD's.

2.2 Semantics

The semantics of an MTD is a set of infinite sequences over *states*; each state is a vector of values for the variables appearing in the component RTD's. The informal semantics of an MTD was given earlier; here, we specify the semantics formally by describing a $\forall$-automaton whose language (the set of accepted sequences) is the semantics of the MTD.

A $\forall$FA, A, is specified by a tuple $(\Sigma, Q, Q_0, \delta, F)$ where Σ is a finite alphabet; Q is a finite set of states; Q_0, a non-empty subset of Q, is the set of initial states; $\delta : Q \times \Sigma \rightarrow 2^{Q^+}$ is the transition relation, where $Q^+ = Q \cup \{\epsilon(q) : q \in Q\}$; and F, a subset of Q^ω, is the acceptance condition. States of the form $\epsilon(q)$ in Q^+ are used to indicate an ϵ move. A run of A on input string $\sigma \in \Sigma^\omega$ is an infinite sequence ρ from $(Q \times \mathbf{N})^\omega$. A configuration (q, i) on the run indicates that the automaton is in state q and is reading symbol σ_i. The run is valid if for $\rho_0 = (q, a)$, $q \in Q_0$ and $a = 0$, and for all $i \in \mathbf{N}$, if $\rho_i = (q, k)$ and $\rho_{i+1} = (q', k')$, then either $k' = k$ and $\epsilon(q') \in \delta(q, \sigma_k)$, or $k' = k + 1$ and $q' \in \delta(q, \sigma_k)$. The run is *accepting* if the projection of ρ on Q is in F, or if there are only ϵ moves from some point on. $\forall$FA's differ from NFA's in that a string σ is in the language of A if, and only if, *all* runs of A on σ are accepting runs (for NFA's, only one run need be accepting). As observed in [MP87], one can generate an NFA for the complement of a $\forall$FA language, and vice versa, in linear time by keeping the same transition structure but negating the acceptance condition. Therefore, as NFA's with Büchi acceptance conditions (that is, $\mathsf{GF}(\mathit{accept})$) suffice to define the ω-regular languages, the same is true for $\forall$FA's with the complemented Büchi condition (that is, $\mathsf{FG}(\mathit{accept})$).

RTD Language: Let $r = (S, \mathcal{E}, SD, CD)$ be a non-empty RTD. The language of r is the set of finite strings z over Σ^* for which there exists a *locator* function $\lambda_z : \mathcal{E} \rightarrow [0..|z| - 1]$, such that:

- Every event in $\mathcal{E}$ can be located on z and has a value consistent with that in r. That is, λ_z is total, and if $\lambda_z(s, i) = k$, then $z_k(s)$, the value of s at the kth position on z, equals $v(s, i)$.
- Let $\lambda_z(s, i) = k$ and $\lambda_z(s, i + 1) = l$. For every j in $[k, l)$, $z_j(s) = v(s, i)$.
- For each sequential dependency $(s, i) \overset{(c,[a,b\rangle)}{\rightarrow} (t, j)$, the number of c events between $\lambda_z(s, i)$ and $\lambda_z(t, j)$ is within $[a, b\rangle$.
- For each concurrent dependency c, and each pair of events $(s, i), (t, j)$ in c, $\lambda_z(s, i) = \lambda_z(t, j)$.

From the well-formedness conditions on an RTD and the first two conditions above, each change event has a unique location, that can be computed by a finite automaton that follows the changes of value on its signal. Once the start of each dependency is located on z, the third and fourth conditions can be determined by finite automata, one for each dependency. Thus, the language of r is determined by a product automaton and therefore is regular. The DFA for this language has a single initial state. An empty RTD has language $\{\epsilon\}$.

MTD Semantics: Let $T{=}(N, C, I, F)$ be an MTD, composed of RTD's $\{r_i\}$. The semantics of T is given by a $\forall$FA, B_T, that is obtained as follows. Let

$B_i = (\Sigma, Q_i, \{q^0\}, \delta_i, F_i)$ be the DFA for r_i, where F_i is the set of final states. The set of states of B_T is $(\cup_i Q_i) \cup \{t_i \mid i \in N_{term}\}$. The initial states of B_T are the initial states of the DFA's for the initial nodes of the MTD. The transitions of B_T include the transitions of each B_i, together with the following new transitions, added in the given order.

1. If r_i is a terminal post node, then for each $q \in F_i$ and each $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(t_i)\}$. For each b, add $\delta(t_i, b) = \{t_i\}$.
2. If r_i is $\wedge$-connected to $r_j, \ldots, r_k$, then for each $q \in F_i$, and each $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(q_j^0), \ldots, \epsilon(q_k^0)\}$.
3. If r_i is $\vee$-connected by guards $g_j, \ldots, g_k$ to RTD's $r_j, \ldots, r_k$, then for each $q \in F_i$, and each $a \in \Sigma$, add $\delta(q, a) = \{\epsilon(q_l^0)\}$, where $g_l(a)$ is the unique guard that holds for a.
4. For each state q that is in its own ϵ-closure (that is, reachable from itself by ϵ moves) add q to each $\delta(q, a)$.

The first rule adds a transition from the final state of a terminal post node to a state that accepts any subsequent sequence of values. The second and third rules implement forking and deterministic choice. The last rule is needed to ensure that MTD's with overlapping checks (e.g., Figure 2) have the desired semantics. The fairness condition of B_T is that a run is rejecting if (a) it gets stuck in the states of some post-node, from some point onwards, or (b) infinitely often, it exits out of DFA's for nodes that are not in F. This can be written as a disjunction of co-Büchi and Büchi conditions. Pre- and post- nodes behave as indicated by their names: the acceptance condition ensures that whenever a run enters a post-RTD, it *must* satisfy the entire pattern in that node; thus, every postcondition that is enabled must hold. On the other hand, a pre-node may not be satisfied but, if it is, the automaton construction ensures that the following nodes are checked.

Expressiveness: The following theorem shows that MTD's can express any ω-regular property over events.

Theorem 0. For any ω-regular language $\mathcal{L}$ of change event sequences, there is an MTD T such that the language of T is the set of those state sequences which, when projected on to change events, form $\mathcal{L}$.

Proof Sketch. Since $\mathcal{L}$ is ω-regular, it can be represented by a $\forall$FA A with a co-Büchi acceptance condition on transitions [MP87]. The MTD, T, is constructed from A as follows: transitions of A are turned into pre-nodes, and states of A become $\wedge$-connectors. The co-Büchi acceptance condition of A is turned into a corresponding co-Büchi fairness condition on the nodes of T. $\square$

2.3 Model Checking with MTD's

We gave the semantics of an MTD T as a $\forall$FA, B_T. This automaton, could be quite large, and may not be appropriate for model checking. This is because the DFA for a post node can be exponential in its size, since it must resolve all non-deterministic choices between unordered events. We now sketch the construction of an alternative $\forall$FA, that is more complex, but of size cubic in the size of the MTD.

There are two key parts to this construction. First, since the events of a pre-node are totally ordered, there is a DFA for the pre-node of size linear in the size of the node. This DFA just checks for the events in the order given by the total ordering. Second, for a post-node, one can construct a $\forall$FA on finite strings instead of a DFA, along the lines indicated in [AEN99]. This $\forall$FA runs several small DFA checks in parallel. The DFA's check (a) each waveform, (b) each sequential dependency, using the linear size DFA's for the location function, and (c) each concurrent dependency, by locating one event e of that dependency, and checking pairwise that for each other event f in the dependency, e and f occur at the same instant. As shown in [AEN99], these checks can be done with a $\forall$FA of size cubic in the RTD size. We can replace the DFA's for the post nodes in the semantics with these $\forall$FA's. Since the set of final events of an RTD is a concurrent dependency, we can pick the change event of that dependency to signal the end of checking the RTD. The final states of the DFA in the semantics are now replaced by the final state for the waveform DFA that contains this final change event. These replacements result in a $\forall$FA, that we call $\mathcal{A}_T$, that has size[1] at most cubic in the MTD size.

Theorem 1. For any MTD T, $\mathcal{L}(B_T) = \mathcal{L}(\mathcal{A}_T)$.

Proof Sketch. The main component of this proof is the observation that the $\forall$FA replacing a postcondition DFA accepts the same set of finite strings. This is because it checks the same conditions, using DFA's running in parallel instead of a product construction. □

Since $\mathcal{A}_T$ accepts the language of T, the complementary NFA $\overline{\mathcal{A}}_T$, derived from $\mathcal{A}_T$, accepts the complemented language. The model checking problem, stated as $M \models T$, is to show that the language of M is contained in that of T. For an MTD T, this can be done by forming the product $M \times \overline{\mathcal{A}}_T$, which is an NFA with a fairness condition of the form $\mathsf{GFp} \vee \mathsf{FGq}$, and checking for emptiness.

Theorem 2. For MTD T and process M, $M \models T$ can be determined in time at most cubic in the size of T, and linear in the size of M.

Proof. By Theorem 1 and the procedure given above, $M \models T$ iff $\mathcal{L}(\overline{\mathcal{A}}_T \times M) = \emptyset$. This is equivalent to searching for a rejecting run of $\mathcal{A}_T$ on some computation of M. This can be done in time linear in the size of M and linear in the size of $\overline{\mathcal{A}}_T$ [EL87]. Since $\overline{\mathcal{A}}_T$ has size at most cubic in the size of T, the result follows. □

3 Compositional Reasoning with MTD's

Large systems are often composed of many concurrent, interacting processes. This leads to a blowup in the state space of the system, commonly referred to

[1] The size of $\mathcal{A}_T$ is the size of the transition relation, which is the total length of the formulas labeling each transition, plus the number of states.

as "state explosion", which is one of the main barriers to successfully applying model checking in practice. One important way of ameliorating this state explosion is to reason not about the entire system as a whole, but about individual components. The compositional reasoning rule that we use is based on so-called 'circular', assume-guarantee style reasoning, where individual components guarantee certain properties based on assumptions about their environment. The proof rule that we use, from [AENT02], is both sound and complete.

First we describe the model of computation used in the sequel; a detailed description can be found in the reference above. The definitions of processes and asynchronous composition are largely based on those in [AL95]. We assume an unbounded set of variable names. A *process* is given by a tuple $(V, \mathcal{I}, T, \mathcal{F})$, where $V = V_l \cup V_i$ is a set of variables, V_l is a non-empty set of "local" variables and V_i is a disjoint set of "interface" variables. $\mathcal{I}(V_l)$ is an initial condition, $T(V, V')$ is a transition relation, where V' is a set of variable names in 1-1 correspondence with V that define the next state, and $\mathcal{F}(V, V')$ is a fairness condition. A *computation* of the process is a sequence of states that starts at an initial state, where each pair of adjacent states either satisfies the transition relation of the process, or is an "environment" move that does not change the values of local variables.

The interleaving *composition* of processes P_1 and P_2, written as $P = P_1 \,[]\, P_2$, is defined provided that, for each process, its local variables are disjoint from the variables of the other, that is $V_l(P_1) \cap V(P_2) = \emptyset$ and $V_l(P_2) \cap V(P_1) = \emptyset$. Furthermore, P has local variables $V_l(P_1) \cup V_l(P_2)$, interface variables $V_i(P_1) \cup V_i(P_2)$, initial condition $\mathcal{I}(P_1) \wedge \mathcal{I}(P_2)$, transition relation $T(P_1) \vee T(P_2)$, and fairness condition $\mathcal{F}(P_1) \wedge \mathcal{F}(P_2)$.

We use the notation $W' = W$, for a set of variables W, to mean that $(\forall w : w \in W : w' = w)$. The set of infinite executions of process P, $L_\omega(P)$, is the set of infinite sequences satisfying the formula $\mathcal{I}(P) \wedge \mathsf{G}(T_P \vee V_l'(P) = V_l(P)) \wedge \mathcal{F}(P)$. The set of finite executions, $L_*(P)$, is the set of finite sequences satisfying $I(P) \wedge \mathsf{G}(T_P \vee V_l'(P) = V_l(P))$. The *language* of P, denoted by $L(P)$ is $L_\omega(P) \cup L_*(P)$. The formula for $L_\omega(P)$ expresses the constraints that the first state is an initial one, every transition is either that of P or an "environment" transition, which keeps the local variables of P unchanged, and the fairness condition holds. The interleaving of environment and process moves is inspired by [AL95]. This is the appropriate choice for modeling computations of programs that are open to the environment – notice that each component of a composition $P \,[]\, Q$ is, of necessity, an open program. It also has the nice consequence that composition can be viewed as the conjunction of languages. The external language of process P, $L_{ext}(P)$, is the projection of its language on the interface variables, that is, $L_{ext}(P) \equiv (\exists V_l(P) : L(P))$. The relationship $P \models Q$ is defined only when $V_i(Q) \subseteq V_i(P)$, and holds if, and only if, $L_{ext}(P) \Rightarrow L_{ext}(Q)$ is valid.

The *choice* of processes P and Q, denoted by $P + Q$, is the process that behaves either as P or as Q. The *closure* of process P, denoted by $CL(P)$, is a process whose language is the safety closure[2] of the language of P; this is just P with fairness condition *true*.

[2] The safety closure of L is the strongest safety property containing L [AS85].

Compositional Reasoning[AENT02]: For processes P_1, P_2 and T, to show that $P_1 \,[]\, P_2 \models T$, find abstract processes Q_1 and Q_2 such that

1. $P_1 \,[]\, Q_2 \models Q_1$, $Q_1 \,[]\, P_2 \models Q_2$, and
2. $Q_1 \,[]\, Q_2 \models T$, and
3. either $P_1 \,[]\, CL(T) \models (T + Q_1 + Q_2)$, or $P_2 \,[]\, CL(T) \models (T + Q_1 + Q_2)$.

Note that the last check in the proof rule is not needed if one of Q_1, Q_2, or T is a safety property. The auxiliary processes Q_1 and Q_2 act as mutual assumptions and guarantees in the first condition. For instance, P_1 assuming Q_2 satisfies Q_1 and symmetrically P_2 satisfying Q_2 under the assumption Q_1. The second condition ensures that they jointly satisfy T. The last check guards against well-known pitfalls (see [NT00,AENT01]) in applying circular reasoning in the presence of fairness.

In the next section, we present an automated method of partitioning an MTD T into components T_1 and T_2, and for generating the auxiliary processes Q_1 and Q_2 from this partitioning.

3.1 Decomposing an MTD Property

Let T be a specification MTD for $P_1 \,[]\, P_2$. To use T for assume guarantee reasoning, we must decompose T into parts T_1 and T_2 corresponding to processes P_1 and P_2, respectively. We call these the *fragments* of T.

Consider the $\forall$FA, $\mathcal{A}_T$, for T as constructed in Section 2.3. Let $\mathcal{A}_1$ be the $\forall$FA defining fragment T_1, that is obtained from $\mathcal{A}_T$ by the following pruning procedure. In $\mathcal{A}_1$, first retain all the post-node DFA's that either check a waveform for a variable of P_1, check a sequential dependency that ends in an event of P_1, or check for the final event of a node that contains a variable of P_1. Then, for all nodes that are on a path to the post-node DFA's chosen earlier, retain the sub-automata for all pre-nodes, and for all post-nodes that do not contain a variable of P_1, retain only the check for the final event. Prune away all other sub-automata. The form of the acceptance condition remains unchanged. Define $\mathcal{L}(T_1)$ to be the language of $\mathcal{A}_1$.

Theorem 3. For MTD T and fragments T_1 and T_2, $\mathcal{L}(T) = \mathcal{L}(T_1) \cap \mathcal{L}(T_2)$.

Proof Sketch. Since the languages are defined by $\forall$FA, it is more convenient to show the contrapositive. From left-to-right, a rejecting run for a sequence in $\mathcal{L}(T)$ must either get stuck in some post-DFA, in this case, it induces a rejecting run in $\mathcal{A}_i$, where i is the process to which the DFA corresponds; or, it passes through a non-fair node infinitely often and the same run is a rejecting run in one of $\mathcal{A}_1$ or $\mathcal{A}_2$. In the other direction, since the $\forall$FA's for the fragments are sub-automata of $\mathcal{A}_T$, a rejecting run in one of them is a rejecting run for $\mathcal{A}_T$. $\square$

3.2 Constructing Abstract Processes

From the fragments T_1 and T_2 constructed earlier, we can construct the auxiliary processes Q_1 and Q_2, respectively. Informally speaking, Q_i is the automaton $\mathcal{A}_i$

considered as a process (this sort of duality is used, for instance, in the COSPAN model checker). For simplicity, we assume that the processes P_1 and P_2 do not write to a shared variable. The process Q_1 is obtained by determinizing $\mathcal{A}_1$, and modifying the resulting automaton, that accepts sequences, into a process, that generates sequences, as follows: for each automaton transition of a post-node, Q_1 reads variables written by P_2, and writes values of variables written by P_1, according to the transition. For a transition corresponding to a pre-node, Q_1 sets its variables non-deterministically, until the precondition is met. The acceptance conditions on $\mathcal{A}_1$ are turned into fairness conditions for Q_1. To prevent deadlocks in Q_1, we make the following assumption on T: if a variable v occurs in a node r of T, then v may occur in any of r's $\vee$-successors, but v can only occur in one of r's $\wedge$-successors. We have not found this to be a restriction in our case study.

Theorem 4. For each fragment T_i of MTD T, $\mathcal{L}(Q_i) = \mathcal{L}(T_i)$. □

Since $\mathcal{A}_i$ is a $\forall$FA, we may, in general, need to determinize it in order to consider it as a process. This can incur an exponential blowup. However, by the Lichtenstein-Pnueli thesis [LP85], the size of the specification is usually much smaller than the size of the process. Thus, one may assume that, for instance, the size of T_1 is much smaller than the size of P_1, so the potential exponential blowup in going from T_1 to Q_1 still results in a process that is smaller than P_1. Recall that Q_1 replaces P_1 in the compositional checks. Theorem 4 lets us reformulate the check $P_1 \,[]\, Q_2 \models Q_1$, in step 1 of the compositional rule, as $P_1 \,[]\, Q_2 \models \mathcal{A}_1$. Using $\mathcal{A}_1$ on the right hand side allows us use the efficient complementation property of $\forall$FA, that reduces the complexity of this check. Theorems 3 and 4 ensure that we do not need to check step 2 of the compositional rule.

4 Applications

We used the verification tool COSPAN/FormalCheck [HHK96] to verify a number of properties of a Plain Old Telephone System (POTS). We chose COSPAN for its efficient symbolic model checking implementation. However, since it is based on a synchronous process composition model, we had to simulate asynchronous composition by introducing an environment variable, *turn*, that is set nondeterministically, ensuring that at each point, only one component can make a move.

The POTS model consists of a `Switch`, a `Phones` process and the environment process `Env`, that is responsible for assigning the *turn* variable. The `Phones` process contains three telephones, `T0`, `T1` and `T2`, and each telephone is connected to the `Switch`, as shown in Figure 4 for telephone `T0`. Each telephone in `Phones` communicates with the `Switch` through the following variables: `cmd`, `busy` and `outnum`. The `Switch` send messages to the telephones through the variables `msg0`, `msg1` and `msg2`.

We specified and verified the following telephone features: the basic telephone call, call forwarding and changing the existing call forwarding plan. The *call forwarding* feature specified in Figure 5 allows a user to forward a telephone call

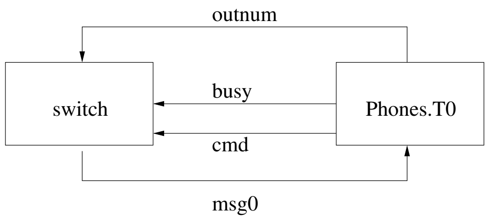

Fig. 4. The Plain Old Telephone System (POTS)

to another phone if the current phone is either busy or if it is not answered after exactly four rings. The first precondition states that telephone `T1` places a call to `T2`, that has a call forwarding plan (`switch.fplan[T2]` = `yes`) that forwards calls to `T0` (`switch.fnum[T2]` = `T0`). If either `T2` is busy or if there is no answer after four rings (`Switch.ringer` = 4), the `Switch` forwards the call. In the postcondition, that specifies the forwarding protocol, the `Switch` (a) informs `T1` that it is forwarding the call (`msg1` = `forwarding`), (b) checks if `T0` is busy (`msg0` = `ru-busy`) and (c) waits for `T1` to acknowledge the forwarding message (`msg1` = `forwarding`).

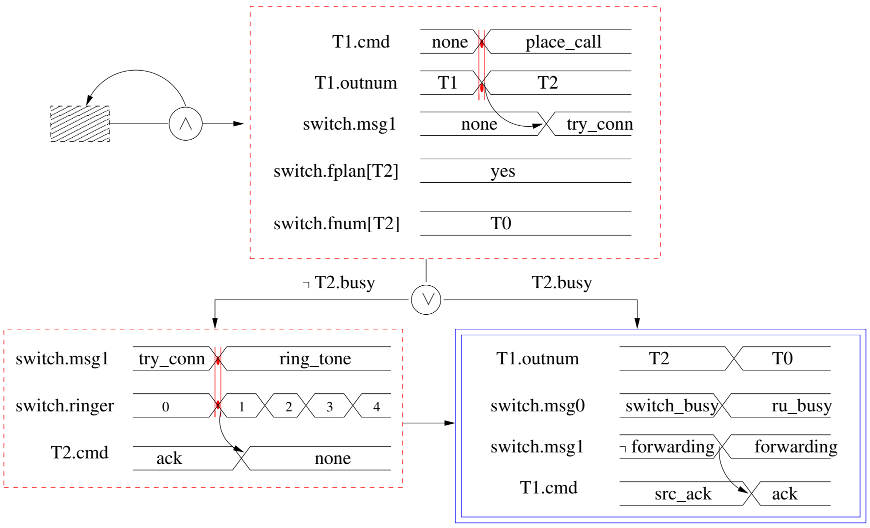

Fig. 5. POTS: Call Forwarding Feature

We did the verification both non-compositionally and compositionally using our assume-guarantee proof rule. Furthermore, we used the heuristic to partition the MTD property `T` into fragments, `Tp` and `Ts`, and to generate the abstract

Model Checking Task	Reachable States	BDD size	Time (seconds)	Space (Mb)
Basic telephone call				
Compositional check 1	2.7 x e11	1060069	91	728
Compositional check 2	7.0 x e06	32553	3	7
Non-Compositional check	7.4 x e09	4707116	263	3923
Call forwarding				
Compositional check 1	1.2 x e11	1610343	190	1234
Compositional check 2	1.4 x e08	23283	2	11
Non-Compositional check	3.1 x e09	2273486	100	1457
Changing forwarding plan				
Compositional check 1	2.6 x e11	1388866	165	789
Compositional check 2	47150	10295	1	1
Non-Compositional check	4.2 x e09	3519750	337	2202

Table 1. Experimental Results

processes, `Qp` and `Qs`. Since the abstract process for the `Switch` had no fairness, we did not need to check the last condition in the proof rule. Moreover, as a consequence of Theorem 4, we did not have to check the second condition (`Qp` [] `Qs` $\models$ `T`) either. The table summarizes our verification results for three properties, where the first two rows correspond to the assume-guarantee checks (`Switch` [] `Qp` [] `Env` $\models$ `Ts` and `Qs` [] `Phones` [] `Env` $\models$ `Tp`) and the last row is the non-compositional verification (`Switch` [] `Phones` [] `Env` $\models$ `T`).

The results in Table 1 indicate that the two compositional checks led to savings of 15%-80% in verification times and comparable savings in BDD-nodes used, when compared to the single non-compositional check. The partitioning heuristic worked without any modification for the first property. However, for the other properties, we had to strengthen the precondition and redo the partitioning and abstract process generation. The checks involving the abstract phones was more expensive because the `Phones` process, being the initiator of all transactions, has more non-determinism. As part of future work, we intend to apply this technique on other examples and investigate ways of refining the abstract process without having to modify the property.

5 Conclusions

The goal of this work, inspired by the success of MSC-based notations for describing asynchronous systems, is to provide a framework for visually specifying asynchronous behavior, coupled with modular reasoning algorithms. This paper does so by proposing a new notation, Modular Timing Diagrams, together with an efficient, polynomial-time model checking algorithm, and an assume-guarantee partitioning algorithm that appears to work quite well in practice.

In [MP87] it was proposed that $\forall$FA be used as a visual specification language; our work brings this idea closer to practical use. While MTD's are similar to ∀FA, the visual notation specifies ordering constraints more compactly, which is evident in the efficient but tricky translation of MTD's to ∀FA as opposed to the simple and direct translation in the reverse direction indicated in Theorem 0. Moreover, we find that visual notations make explicit the information necessary for property decomposition, and therefore are particularly well suited for *automated* compositional reasoning. One may also consider replacing the RTD's in the MTD nodes with synchronous timing diagrams; the translation results and the partitioning heuristics should carry over in a fairly straightforward manner. In future work, we hope to extend the functionality of the RTDT tool [AEKN01] to MTD's.

References

[AEKN00] N. Amla, E.A. Emerson, R.P. Kurshan, and K.S. Namjoshi. Model checking synchronous timing diagrams. In *FMCAD*, 2000.

[AEKN01] N. Amla, E.A. Emerson, R.P. Kurshan, and K.S. Namjoshi. RTDT: a front-end for efficient model checking of synchronous timing diagrams. In *CAV*, 2001.

[AEN99] N. Amla, E.A. Emerson, and K.S. Namjoshi. Efficient decompositional model checking for regular timing diagrams. In *CHARME*, 1999.

[AENT01] N. Amla, E.A. Emerson, K.S. Namjoshi, and R. Trefler. Assume-guarantee based compositional reasoning for synchronous timing diagrams. In *TACAS*, volume 2031 of *LNCS*, 2001.

[AENT02] N. Amla, E.A. Emerson, K. Namjoshi, and R. Trefler. Compositional Reasoning for Asynchronous Systems, 2002. URL: `http://-www.cs.bell-labs.com/who/kedar/publications.html`.

[AG01] R. Alur and R. Grosu. Shared variable interaction diagrams. In *16th IEEE International Conference on Automated Software Engineering*, 2001.

[AL95] M. Abadi and L. Lamport. Conjoining specifications. *ACM Trans. on Programming Languages and Systems (TOPLAS)*, May 1995.

[AS85] B. Alpern and F. Schneider. Defining liveness. *Information Processing Letters*, 21(4), 1985.

[AY99] R. Alur and M. Yannakakis. Model checking of message sequence charts. In *Proc. Tenth International Conference on Concurrency Theory*, 1999.

[DH01] W. Damm and D. Harel. LSCs: Breathing life into message sequence charts. *Formal Methods in System Design*, 19(1), 2001.

[DJS94] W. Damm, B. Josko, and Rainer Schlör. Specification and verification of VHDL-based system-level hardware designs. In Egon Borger, editor, *Specification and Validation Methods*. Oxford University Press, 1994.

[dRdBH+01] W-P. de Roever, F. de Boer, U. Hannemann, J. Hooman, Y. Lakhnech, M. Poel, and J. Zwiers. *Concurrency Verification: Introduction to Compositional and Noncompositional Proof Methods*. Cambridge University Press, 2001.

[dRLP97] W-P. de Roever, H. Langmaack, and A. Pnueli, editors. *Compositionality: The Significant Difference*, volume 1536 of *LNCS*. Springer-Verlag, 1997.

[EL87] E.A. Emerson and C. Lei. Modalities for model checking: Branching time logic strikes back. *Science of Computer Programming*, 8(3):275–306, 1987.

[Fis96] K. Fisler. *A Unified Approach to Hardware Verification Through a Heterogeneous Logic of Design Diagrams.* PhD thesis, Computer Science Department, Indiana University, August 1996.

[HHK96] R.H. Hardin, Z. Har'el, and R.P. Kurshan. COSPAN. In *CAV*, volume 1102 of *LNCS*, 1996.

[Hol97] G. Holzmann. The SPIN model checker. *IEEE Transactions on Software Engineering*, 23(5), May 1997.

[HP94] G.J. Holzmann and D. Peled. An improvement in formal verification. In *FORTE*, 1994.

[KW01] J. Klose and H. Wittke. An automata based interpretation of live sequence charts. In *TACAS*, volume 2031 of *LNCS*, 2001.

[LL94] P.B. Ladkin and S. Leue. What do message sequence charts mean? In *Formal Description Techniques*, 1994.

[LP85] O. Lichtenstein and A. Pnueli. Checking that finite state concurrent programs satisfy their linear specifications. In *POPL*, 1985.

[MP87] Z. Manna and A. Pnueli. Specification and verification of concurrent programs by forall-automata. In *POPL*, 1987.

[MP00] A. Muscholl and D. Peled. Analyzing message sequence charts. In *2nd Workshop on SDL and MSC*, 2000.

[MPS98] A. Muscholl, D. Peled, and Z. Su. Deciding properties for message sequence charts. In *FoSSaCS*, 1998.

[MSC96] *ITU-T Recommendation Z.120, Message Sequence Chart (MSC)*, 1996.

[NT00] K.S. Namjoshi and R.J. Trefler. On the completeness of compositional reasoning. In *CAV*, volume 1855 of *LNCS*. Springer-Verlag, 2000.

[PR99] M. Plath and M. Ryan. Feature integration using a feature construct. *Science of Computer Programming*, 41(1), 1999.

[SHE01] M.H. Smith, G.J. Holzmann, and K. Etessami. Events and constraints: A graphical editor for capturing logic requirements of programs. In *5th International Symposium on Requirements Engineering*, 2001.