Compositional Branching-Time Measurements

Radu Grosu[1], Doron Peled[2], C.R. Ramakrishnan[3], Scott A. Smolka[3], Scott D. Stoller[3], and Junxing Yang[3]

[1] Vienna University of Technology
[2] Bar Ilan University
[3] Stony Brook University

Abstract. Formal methods are used to increase the reliability of software and hardware systems. Methods such as model checking, verification and testing are used to search for design and coding errors, integrated in the process of system design. Beyond checking whether a system satisfies a particular specification, we may want to measure some of its quantitative properties. Earlier works on system measurements suggest extending model checking techniques to measure quantitative artifacts, based on weights associated with the transitions of a transition system. Other works allow counting while performing model checking or runtime verification. This paper presents a simple and efficient compositional measuring framework based on quantitative state testers. The framework allows combining multiple measures, such as distance and power consumption, using a variety of functions, such as min, max, and average. This supports calculation of interesting compound measures that quantitatively characterize a system's behavior.

1 Introduction

Model checking techniques [6, 12] are successfully integrated into the software and hardware development process. More than 30 years of research has produced multiple techniques. Some of them are quite impressive in the size of systems that they can handle and in verification speed. A recent trend is to look at quantitative properties, for example, providing measures on how robustly a property is satisfied in probabilistic automata [9] or weighted automata [1, 5]. Another approach is to ask for, in addition to the qualitative indication of the satisfaction of a property, a measurement, which is usually based on the accumulated time required to satisfy parts of the specification [2, 8]. These measures can be used for optimizing various parameters of the system.

We are motivated to provide a framework for measuring branching properties of a system. This has received so far little attention, whereas linear properties have been intensively studied. We aim to develop an efficient compositional measuring framework that extends the idea of testers [10, 11] to allow quantitative operations. These extended testers are applied to the states of the measured structure and conceptually communicate with each other through flow of information between adjacent states. We generically refer to nodes in the given graph

structure as "states", although the nodes may represent states in a state space, locations in a geographical space, etc.

This approach generalizes CTL specification, and in fact we show how to encode model checking of CTL in this way. This framework inherits its efficiency from the CTL logic. Nevertheless, instead of writing a CTL formula annotated with some measurement parameters, we provide a combination of recursive observers (functions) that work together to provide the desired measurement. The value associated with each state of the structure by the tester is dependent on the values at adjacent, i.e., successor and predecessor, states, henceforth called the *neighbors*. For example, the property EXp holds in a state if either p holds there, or there is a successor state where EXp holds.

In order to allow various measurements of a structure, we permit the observing functions, which combine values to produce outputs, to be over non-Boolean domains. We can describe the computation associated with this framework as a synchronized update of values associated with each state, where the new value is dependent on the previous values associated with that state and its neighbors. This allows a simple implementation, where in each step, each state applies the observing function to the previous values associated with its neighbors. On the one hand, this may not provide the most efficient implementation and should be optimized. On the other hand, it suggests a way of parallelizing the measurements.

This kind of measuring can be viewed as extending the runtime verification idea in [11] from checking whether a finite sequence can be extended to satisfy an LTL formula to measuring an entire structure. Our approach is not limited to CTL-like Boolean properties but can use different domains, including combinations of domains. Section 3 presents an example that uses two measures: the amount of battery power consumed when a robot traverses a path, and the shortest distance from the robot to a towing station. Our framework provides a more flexible measuring tool than most quantitative extensions of logic, which deal with one fixed measure, most often the average, minimum, or maximum accumulated length along paths.

When extending the framework from Boolean to more general domains, it becomes challenging to ensure that the measurement calculation terminates. Indeed, in general, applying functions recursively may not terminate, and termination itself can be undecidable. For this reason, we require that a well founded ordering is used, and successive values of a state in a measured structure must decrease in this ordering.

Our view of measurement is local, from the point of view of a state in the measured structure. To demonstrate the difference from a global view, consider CTL model checking, where a monotonicity argument can be applied on the (increasing or decreasing, respectively) size of the states while calculating the (least or greatest, respectively) fixed-points. Instead, our measurement needs to decrease locally, on the values calculated on the states. As the calculated values may not contain the information needed to show progress toward termination, we add another component: a counter that counts down from the size of the state

space (the number of states) or its diameter (the length of the longest simple path). This is a generalization of the bounds used in bounded model checking [3].

Our formalism presents a combination of observing functions, applied synchronously to the states of the measured structure, and guarded by a local well-foundedness argument to ensure termination. This can be viewed as a denotational approach, defining the measurement, but is also very close to the operational approach, defining the kind of computation needed at each state of the structure, for a finite number of steps. Therefore, we also suggest developing logics that allow more abstract and denotational representations of desired measurements, and translating (compiling) the logics into the testers used in our framework. We illustrate this approach in the example in Section 3, by presenting some CTL-like formulas extended to express various measurements.

The closest work to this paper, as far as we know, is [4]. There, an integer-based measurement of a state space is sought, and calculated using quantitative bound automata, with decision procedures based on dynamic programming. Our approach, based on testers, is more of a specification formalism, allowing us to directly describe the measure of a structure by the combination of functions (whose type can be Boolean, integer, or any other finitely representable type) and flow of information between states and their neighbors. Both approaches realize the need to bound the computation, where this is done in [4] by providing a bound on the number of iterations, and in our framework by requiring values to decrease in a well-founded ordering.

The remainder of the paper is organized as follows. Section 2 presents our framework, including our extended concept of testers, and shows how to express CTL model checking in our framework. Section 3 presents an example based on a mobile robot that illustrates the expressiveness of our framework.

2 Measuring Structures

Our computation model is based on a collection of synchronous processes that communicate via shared variables. Each process has input, output, and state variables, which may take values from arbitrary domains, including Boolean, Integers, Reals, and Cartesian products. In each step of the computation, each process updates its output and state variables based on a transition relation that relates the values of its output and next-state variables to the values of its input and current state variables. Transitions may in general be nondeterministic, although the examples in this paper focus on deterministic processes for the purpose of measurement.

The topology of this process network is specified by mapping output variables of a process to input variables of other processes. Additionally, each process is associated with a function that maps the current value of its state variables to a value from a well-founded domain. This value is required to decrease whenever the tester's state changes, guaranteeing termination.

Testers

An *atomic tester* T is a process having the following components:

- $T\langle i\rangle$ is a finite set of input variables;
- $T\langle o\rangle$ is a finite set of output variables;
- $T\langle state\rangle$ is a finite set of state variables (with specific initial values);
- $T\langle \rho\rangle$ is a transition function, mapping values of $T\langle i\rangle \cup T\langle state\rangle$ to values of $T\langle state\rangle \cup T\langle o\rangle$; and
- $T\langle w\rangle$, a function mapping values of $T\langle state\rangle$ to values in a well-founded order $(W, \ll)$.

Note that we represent testers as structures with named components, and we use the notation $T\langle c\rangle$ to select component c of tester T. A tester is *stateless* if it has no state variables.

The above definition of testers differs from the original notion in [10] in several ways. First, our testers are primarily intended for measuring properties of finite structures, and are not equipped with justice or compassion predicates. Second, our testers have explicit input and output variables, while those in [10] operate over streams of boolean values and have input and output defined implicitly. Third, and perhaps most importantly, the variables in our testers are not restricted to be boolean; in fact, tester variables can range over any domain including structured ones. Finally, each tester is equipped with a function $T\langle w\rangle$ used to ensure termination. Semantically, we require that, in every transition of a tester, either the valuation of its state variables remains unchanged, or the valuation changes from v to v' and $T\langle w\rangle(v') \ll T\langle w\rangle(v)$. This ensures termination.

A *tester circuit*, $\mathcal{C}$, is a collection of interconnected atomic testers. More precisely, $\mathcal{C} = \langle \mathcal{T}, I, O, \mathcal{S}\rangle$, where $\mathcal{T}$ is a set of atomic testers, $I \subseteq \cup_{T\in\mathcal{T}} T\langle i\rangle$ and $O \subseteq \cup_{T\in\mathcal{T}} T\langle o\rangle$ are the sets of inputs and outputs, respectively, of the circuit, and $\mathcal{S} \subseteq ((\bigcup_{T\in\mathcal{T}} T\langle o\rangle) \times (\bigcup_{T\in\mathcal{T}} T\langle i\rangle))$ specifies the connections between testers, by associating input variables with output variables. For convenience, we assume that input and output variables have globally unique names.

The computation in a tester circuit starts with all testers in initial states. The initial output of all testers is a special value "$\perp$". Each tester evolves synchronously, reading its inputs, and evaluating the transition function, thereby computing its next state and outputs.

CTL Model Checking with Testers

We now illustrate the use of testers, treating the model checking of CTL formulas as an instance of measurement of a given Kripke structure. In the following, we assume a standard definition of a Kripke structure K over a finite set of states S and atomic propositions P, given by $\langle S, \rightarrow, \sigma\rangle$, where $\rightarrow \subseteq S \times S$ is a transition relation, and σ is a function mapping states in S to sets of propositions. Given a Kripke structure K, let $n(K)$ denote the number of states in K, and let $d(K)$

denote the *diameter* of K, i.e., the length of the longest cycle-free path in its transition graph. A state t is a *successor* of a states s if $s \to t$.

We consider CTL formulas over a set of propositions P specified using the following syntax:

$$\varphi ::= P \mid \neg\varphi \mid (\varphi \wedge \varphi) \mid EX\varphi \mid E(\varphi U \varphi) \mid A(\varphi U \varphi)$$

Additional operators of CTL can be defined in terms of these, e.g., $\mathit{true} = \neg(p \wedge \neg p)$ for some proposition p, $EF\varphi = E(\mathit{true}\ U\ \varphi)$, $AG\varphi = \neg EF(\mathit{true}\ U\ \neg\varphi)$, and $AX\varphi = \neg EX\neg\varphi$.

We now describe the construction of a tester circuit for checking whether a given Kripke structure is a model for a given CTL formula φ. Recall that the outputs of all testers are initially set to an undefined value $\perp$. We extend the standard Boolean operators in a symmetric way as follows: $\mathit{true} \vee \perp = \mathit{true}$, $\mathit{true} \wedge \perp = \perp$, $\mathit{false} \vee \perp = \perp$, $\mathit{false} \wedge \perp = \mathit{false}$, and $\neg\perp = \perp$.

For each formula φ and state $s \in S$, we construct a circuit $\mathcal{C}_{\varphi,s}$ based on the structure of φ, as defined below. Each circuit $\mathcal{C}_{\varphi,s}$ has a single output variable and is designed so that the final value of the output variable is true iff φ holds at state s. For CTL model checking, the circuits have no inputs, so for brevity, we omit the specification of input variables for circuits in the following construction. We express each transition relation as a set of equations (one for each state variable and output variable) in which unprimed variables represent the values of variables in the current state, and primed variables represent the values of variables in the next state. For brevity, we also omit the mapping to well-founded orders; standard arguments can be used to show termination of this calculation.

case: φ is a proposition p: Let $T_{p,s}$ be a stateless tester such that $T_{p,s}\langle i\rangle = \{\}, T_{p,s}\langle o\rangle = \{o\}$, and $T_{p,s}\langle\rho\rangle = \{o' = (p \in \sigma(s))\}$. Then $\mathcal{C}_{\varphi,s} = \langle\{T_{p,s}\}, \{o\}, \emptyset\rangle$.

case: φ is a negated formula $\neg\varphi_1$: Let $T_{\varphi,s}$ be a stateless tester such that $T_{\varphi,s}\langle i\rangle = \{i\}, T_{\varphi,s}\langle o\rangle = \{o\}$, and $T_{\varphi,s}\langle\rho\rangle = \{o' = \neg i\}$.
Let $\mathcal{C}_{\varphi_1,s} = \langle\mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$. Then $\mathcal{C}_{\varphi,s} = \langle\{T_{\varphi,s}\} \cup \mathcal{T}_1, \{o\}, \mathcal{S} \cup \{(o_1, i)\}\rangle$.

case: φ is a conjunction $\varphi_1 \wedge \varphi_2$: Let $T_{\varphi,s}$ be a stateless tester such that $T_{\varphi,s}\langle i\rangle = \{i_1, i_2\}, T_{\varphi,s}\langle o\rangle = \{o\}$, and $T_{\varphi,s}\langle\rho\rangle = \{o' = i_1 \wedge i_2\}$. Let $\mathcal{C}_{\varphi_1,s} = \langle\mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$ and $\mathcal{C}_{\varphi_2,s} = \langle\mathcal{T}_2, \{o_2\}, \mathcal{S}_2\rangle$. Then $\mathcal{C}_{\varphi,s} = \langle\{T_{\varphi,s}\} \cup \mathcal{T}_1 \cup \mathcal{T}_2, \{o\}, \mathcal{S} \cup \{(o_1, i_1), (o_2, i_2)\}\rangle$.

case: φ is an exists-next formula $EX\varphi_1$: Let state s have n successors. Let $T_{\varphi,s}$ be stateless tester with n inputs, namely, $r_1, \ldots r_n$, and one output o such that $o' = \bigvee_i r_i$. Formally, $T_{\varphi,s}\langle i\rangle = \{r_1, \ldots, r_n\}, T_{\varphi,s}\langle o\rangle = \{o\}$, and $T_{\varphi,s}\langle\rho\rangle = \{o' = \bigvee_i r_i\}$.
Let the successors of s be $t_1, t_2, \ldots, t_n$. Let $\mathcal{C}_{\varphi_1,t_j} = \langle\mathcal{T}_j, o_j, \mathcal{S}_j\rangle$ for each successor t_j.
Then $\mathcal{C}_{\varphi,s} = \langle\{T_{\varphi,s}\} \cup \bigcup_{j=1..n} \mathcal{T}_j, \{o\}, \mathcal{S} \cup \bigcup_{j=1..n}\{(o_j, r_j)\}\rangle$.

case: φ is an exists-until formula $E(\varphi_1 U \varphi_2)$: Let state s have n successors. Let $T_{\varphi,s}$ be a tester with $2 + n$ inputs, namely, $i_1, i_2, r_1, \ldots r_n$, one output

o, and one state variable x initialized to $d(K)$, the diameter of the Kripke structure. The tester is such that at each step after i_1 and i_2 get non-$\perp$ values, x is decremented, and the output o' is computed as $i_2 \vee (i_1 \wedge (\bigvee_{j=1..n} r_j))$. If x reaches 0 and the output is $\perp$, then it is set to *false*. Formally,

$$T_{\varphi,s}\langle i\rangle = \{i_1, i_2, r_1, \ldots, r_n\}$$
$$T_{\varphi,s}\langle o\rangle = o$$
$$T_{\varphi,s}\langle state\rangle = \{x = d(K)\}$$
$$T_{\varphi,s}\langle \rho\rangle = \left\{ \begin{array}{l} x' = \begin{cases} \text{if } i_1 = \perp \vee i_2 = \perp \text{ then } x \\ \text{else if } x > 0 \text{ then } x - 1 \text{ else } 0 \end{cases} \\ o' = \text{ if } x = 0 \wedge v = \perp \text{ then } \mathit{false} \text{ else } v \\ \qquad \text{where } v = i_2 \vee (i_1 \wedge (\bigvee_{j=1..n} r_j)) \end{array} \right\}$$

Let $\mathcal{C}_{\varphi_1,s} = \langle \mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$ and $\mathcal{C}_{\varphi_2,s} = \langle \mathcal{T}_2, \{o_2\}, \mathcal{S}_2\rangle$. Let s have n successors, namely $t_1, t_2, \ldots, t_n$. Let $\mathcal{C}_{\varphi,t_j} = \langle \hat{\mathcal{T}}_j, \{\hat{o}_j\}, \hat{\mathcal{S}}_j\rangle$ for each successor t_j. Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{T}}_j$ and $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{S}}_j$.
Then $\mathcal{C}_{\varphi,s} = \langle \mathcal{T} \cup \{T_{\varphi,s}\}, \{o\}, \mathcal{S} \cup \{(o_1, i_1), (o_2, i_2)\} \cup \bigcup_{j=1..n} \{(\hat{o}_j, r_j)\}\rangle$.

case: φ is an always-until formula $A(\varphi_1 U \varphi_2)$: This is similar to the exists-until case above, except that it performs a conjunction (instead of a disjunction) over the results r_i from successor states. Let state s have n successors. Let $T_{\varphi,s}$ be a tester with $2 + n$ inputs, namely, $i_1, i_2, r_1, \ldots r_n$, one output o, and one state variable x initialized to $d(K)$, the diameter of the Kripke structure. Let

$$T_{\varphi,s}\langle i\rangle = \{i_1, i_2, r_1, \ldots, r_n\}$$
$$T_{\varphi,s}\langle o\rangle = o$$
$$T_{\varphi,s}\langle state\rangle = \{x = d(K)\}$$
$$T_{\varphi,s}\langle \rho\rangle = \left\{ \begin{array}{l} x' = \begin{cases} \text{if } i_1 = \perp \vee i_2 = \perp \text{ then } x \\ \text{else if } x > 0 \text{ then } x - 1 \text{ else } 0 \end{cases} \\ o' = \text{ if } x = 0 \wedge v = \perp \text{ then } \mathit{false} \text{ else } v \\ \qquad \text{where } v = i_2 \vee (i_1 \wedge (\bigwedge_{j=1..n} r_j)) \end{array} \right\}$$

Let $\mathcal{C}_{\varphi_1,s} = \langle \mathcal{T}_1, \{o_1\}, \mathcal{S}_1\rangle$ and $\mathcal{C}_{\varphi_2,s} = \langle \mathcal{T}_2, \{o_2\}, \mathcal{S}_2\rangle$. Let s have n successors, namely $t_1, t_2, \ldots, t_n$. Let $\mathcal{C}_{\varphi,t_j} = \langle \hat{\mathcal{T}}_j, \hat{o}_j, \hat{\mathcal{S}}_j\rangle$ for each successor t_j. Let $\mathcal{T} = \mathcal{T}_1 \cup \mathcal{T}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{T}}_j$ and $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2 \cup \bigcup_{j=1..n} \hat{\mathcal{S}}_j$.
Then $\mathcal{C}_{\varphi,s} = \langle \mathcal{T} \cup \{T_{\varphi,s}\}, \{o\}, \mathcal{S} \cup \{(o_1, i_1), (o_2, i_2)\} \cup \bigcup_{j=1..n} \{(\hat{o}_j, r_j)\}\rangle$.

3 Example

We illustrate the use of testers by measuring the weighted branching structure of an autonomous robot with respect to a desired CTL property. The movement of the robot within its environment is expressed with the finite, weighted Kripke structure $K = (S, s_1, \rightarrow, \sigma, c, d)$ shown in Figure 1(a). State s_1 is marked by σ as initial, and states s_5 and s_6 are marked by σ as towing. Each transition $\rightarrow_{ij}$ of K is annotated with two weights. The first weight, c_{ij}, is the energy consumed (as a percentage of a fully charged battery) along $\rightarrow_{ij}$, i.e., when moving from state s_i to state s_j. The second weight, d_{ij}, is the distance traveled (in meters) along $\rightarrow_{ij}$.

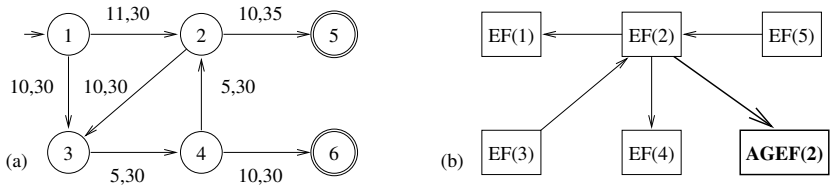

Fig. 1. (a) Weighted Kripke structure associated with the movement of a robot. On each edge, the first weight represents the energy consumed on that edge, and the second weight represents the distance traveled on that edge. (b) Communication structure of the tester $EF(2)$. This tester receives horizontal messages from the testers $EF(5)$ and $EF(3)$, sends horizontal messages to the testers $EF(1)$ and $EF(4)$, and vertical messages to the tester $AGEF(2)$.

We would like to check and measure the following property φ: *Whenever the robot reaches a state where its overall battery consumption* $bc > 80$*, there is a towing state within a distance* $td < 100$. Let ts be an atomic proposition that is true in towing states. Then, φ can be written in a CTL-like logic as:

$$\varphi \doteq AG_{bc>80}\ EF_{td<100}\ ts$$

Although φ seems to capture our intuition, we encounter difficulties as soon as we seek to measure it, due to the inherent nondeterminism in CTL formulas. In fact, we are not interested in any towing distance $td<100$ from a state, but in its *minimal* towing distance. Hence, the property we would like to measure can be more precisely stated as: *Whenever the robot reaches a state where its overall battery consumption* $bc>80$*, there is a* minimal *towing distance* $td<100$.

Testers

In order to capture, and more importantly, to properly *measure* such properties in all their generality, we associate with each temporal operator, and each state of the weighted Kripke structure K, a tester. These testers communicate in a data-flow fashion both horizontally, with the testers of the same kind, and vertically, with the testers corresponding to the enclosing operator.

For example, the communication structure of the EF tester associated with state s_2 is shown in Figure 1(b). This tester receives messages from the EF testers associated with its successor states s_3 and s_5 in K, and sends messages to the EF testers associated with its predecessor states s_1 and s_4 in K. It also sends messages to the tester $AGEF$ of its enclosing formula at state s_2.

As the example indicates, the communication structure of a tester is completely defined by the structure K, the formula φ we intend to measure, and the state s with which the tester is associated. A tester accumulates (folds) path information in K starting at (or ending in) s and passes it to the other testers. In order to do this, the tester requires the following information: (1) a procedure for folding information along a single path, (2) a procedure for folding the information from multiple paths, and (3) a termination condition.

A natural way to provide such information is through a complete, idempotent dioid structure $D = (+, \times, 0, 1)$, where $+$ is an additive, commutative and idempotent monoid, with neutral element 0, and $\times$ is a multiplicative monoid, with neutral element 1. In this setting, (1) is taken care of by multiplication, (2) is taken care of by the addition, and (3) is taken care of by the completeness of the dioid. In some cases, however, one would like to fold information using other operations, such as averaging. In such situations, an explicit termination condition, which we refer to as a "stopping criterion", has to be provided, as the completeness of the dioid does not suffice. Stopping criteria can also be used to speed up the computation.

For AF and AG formulas, an operator folding the information computed in all their satisfying states must also be provided. To make the above discussion more precise, let us define the testers associated with φ.

The *EF* Tester

Consider first the EF tester associated with the subformula $EF_{td<100}\, ts$ at state s_4. In this state, we have two paths that satisfy the constraint $td < 100$: the path $p_1 = s_4 s_6$ with associated towing distance $td_1 = 30$, and the path $p_2 = s_4 s_2 s_5$ with associated towing distance $td_2 = 65$.

The towing distances td_1 and td_2 are computed by folding the weights along the paths p_1 and p_2, respectively, in an additive fashion. Moreover, since we are only interested in the shortest path, we fold the information among different paths by taking their minimum. Hence, the idempotent dioid structure we are interested in is $D = (min, +, \infty, 0)$. The stopping criterion, $\psi \doteq (td \geq 100)$ is not necessary for convergence, but it speeds up the computation, and helps compute the measure for the entire CTL formula φ. The atomic proposition we are passing to the EF tester as a parameter is simply $p \doteq ts$.

Given the above considerations, the general specification of an $EF^D_\psi\, p\,(s, K)$ tester can be given once and for all as below. We assume that the communication structure is automatically compiled in EF from K and φ. For readibility of the tester, we instantiate p, ψ and D in its definition.

$EF^{(min,+,\infty,0)}_{td \geq 100}\, ts\,(s, K)$
{
  init: `td = (`$\sigma(s) = ts$`) ? 0 :` ∞
  stop: `td` ≥ 100
  transition:
    `td =` $min_{t \in (s \to t)}$ `(td, receiveEF(`t`)` $+ d_{st}$`))`
    `sendEF(`t`)`$_{t \in (t \to s)}$ `= td`
    `sendAGEF(`s`) = (td` ≥ 100`) ? (`b`=F,` td`=td) : (`b`=T,` td`=td);`
}

Note that this section uses a more concise and less formal notation for testers, including a send-receive notation for communication. For example, `sendEF(`t`) = td` denotes sending the value of `td` to the tester for the EF formula at state t, and `receiveEF(`t`)` denotes the value received from the EF tester at state t. This

notation can be translated straightforwardly into the more formal notation in Section 2. We also use tuples with named fields as communication messages.

The AG Tester

The tester associated with the $AG_{bc>80}$ property can be defined independently of the rest of the subformulas in φ. The purpose of this tester is to compute the battery consumption in a forward fashion, starting from the initial state.

We would like to stop the computation of the battery consumption at a given state s, as soon as we arrive (for the first time) at s with $bc>80$. Moreover, if we arrive at s along two different paths with values bc_1 and bc_2, we would like to consider only the maximum of bc_1 and bc_2. Hence, the dioid structure we are interested in has the form $D=(max,+,-\infty,0)$. The termination condition in this case is $\psi \doteq (bc>80)$.

The intuition is as follows. In structure K of Figure 1(a), there is no simple path ending in a state with $bc>80$. However, looping sufficiently many times within the cycle $s_2s_3s_4s_2$ increases the battery consumption until we arrive at s_3 with $bc=81$, and thereafter in s_4, s_2, and s_6 with bc equal to 86, 91 and 96, respectively. Hence, we can define (once and for all) the AG tester as below. Again, for readability, we instantiate the parameters ψ and D.

$AG_{bc>80}^{(max,+,-\infty,0)}(s,K)$
{
  init: `bc =` $(\sigma(s)=init)$ `? 0 :` $-\infty$
  stop: `bc` >80
  transition:
    if `(bc` ≤ 80`)` **then**
      `bc =` $max_{t\in(t\to s)}$ `(bc, receiveEF(`t`)` $+c_{ts}$`))`
    `sendAG(`t`)`$_{t\in(s\to t)}$ `= bc`
    `sendAGEF(`s`) = bc` >80 `? (`b`=T,` bc`=bc) : (`b`=F,` bc`=bc)`
}

The AGEF tester.

The $AGEF$ tester at state s collects the information from its $AGEF$ peers, and from the AG and the EF testers at state s. Then, in conjunction with its $AGEF$ peers, it checks and measures the top-level formula φ.

For each state where if $bc>80$ holds then it is also the case that $td<100$ holds (which is the formal requirement in φ), we would like for our example that $AGEF(s)$ first compute the linear combination $lc=0.6\times bc+0.4\times td$. We would then like it to back propagate this information so that the initial state will contain the average of all such linear combinations.

The main problem in the back propagation is the cycle $s_2s_3s_4s_2$: simply sending lc to the $AGEF$ testers of all its predecessor states would result in double counting. Assuming the existence of a linear order on the states, with the initial state as minimal, we therefore send the lc information to only a single predecessor, the one which is minimal in the state ordering.

To ensure termination as well as proper property checking and measuring, each $AGEF$ tester sends a tuple $(bb, ns, lcSum, done)$. The value bb is true when the state satisfies $\neg(bc > 80) \vee (td < 100)$, and all of its greater $AGEF$ successors do. The value $lcSum$ is the sum of the lc value of the current $AGEF$ state and the lc values of its greater successors. The value of ns is the number of states whose lc values are summed in $lcSum$. The value of $done$ is true when all the greater $AGEF$ successors are done; this value is also used as the stopping condition of the $AGEF$ tester. Note that the average of the lc values can always be computed as $lcSum/ns$; for brevity, this calculation is omitted from the pseudocode for the $AGEF$ tester.

The structure $D = (avg, lin(0.6, 0.4), 0)$ contains the commutative monoid for avg, and the linear-combination operator (with its associated weights). One therefore needs in addition a state ordering, and the termination condition $done$. Given all these considerations, the specification of the $AGEF$ tester is as follows:

$AGEF_{done}^{(avg,lin(0.6,0.4),0)}(s, K)$
{
  init:
    bb = (receiveAG(s).b $\Rightarrow$ receiveEF(s).b)
    lcSum = bb ? lin(0.6,0.4)(receiveAG(s).bc, receiveEF(s).td): 0
    ns = 1
    done = F
  stop: done
  transition:
    alreadyDone = done
    done = $\wedge_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$done$)
    if (done $\wedge$ $\neg$alreadyDone) **then**
      sb = $\wedge_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$bb$)
      bb = bb $\wedge$ sb
      ns = ns + $sum_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$ns$)
      lcSum = lcSum + $sum_{t\in(s\to t)\wedge(t>s)}$(receiveAGEF($t$).$lcSum$)
    predecessor = $min_{t\in(t\to s)\wedge(t<s)}$($t$)
    sendAGEF(predecessor) = (bb = bb, ns = ns, $lcSum$ = lcSum, $done$ = done)
}

To simplify the pseudo-code, we assume that the receives from the AG and EF testers in the **init** section block the $AGEF$ tester until the computations of the AG and EF testers have terminated, because we want the $AGEF$ tester to compute with the final values from those testers. This implicit synchronization can be implemented explicitly using additional variables.

This tester has the property that the formula φ is true, provided that the $AGEF$ tester of state s_1 is done and its Boolean value bb is true. In that case, the desired average is $lcSum/ns$.

Implementation and results.

We implemented the robot example in MATLAB to gain experience with the behavior and performance of these testers. We used centralized data structures

for initial-prototyping purposes. As future work, we plan to develop a distributed implementation.

In Table 1, we present the results obtained from the testers. One can easily check that all boolean and real values are correctly computed and propagated to a tester's neighbors. The final result is the tuple $(T, 6, 337.0)$ computed by the $AGEF$ tester for the initial state s_1. These values convey the fact that the property does hold for the weighted Kripke structure K of Figure 1(a) and yields the desired average as $337.0/6 = 56.17$.

	EF Testers		AG Testers		$AGEF$ Testers		
	b	td	b	bc	bb	ns	$lcSum$
s_1	T	65	F	0	T	6	337.0
s_2	T	35	T	91	T	2	117.2
s_3	T	60	T	81	T	3	193.8
s_4	T	30	T	86	T	2	121.2
s_5	T	0	T	81	T	1	48.6
s_6	T	0	T	96	T	1	57.6

Table 1. Results obtained from the testers for the robot example

References

1. Shaull Almagor, Udi Boker, Orna Kupferman: What's Decidable about Weighted Automata? LNCS 6996, Springer-Verlag, ATVA 2011: 482-491.
2. Rajeev Alur, Kousha Etessami, Salvatore La Torre, Doron Peled: Parametric temporal logic for "model measuring". ACM Trans. Comput. Log. 2(3): 388-407 (2001).
3. Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, Yunshan Zhu: Bounded Model Checking. Vol. 58 of Advances in Computers, Academic Press, 2003.
4. Arindam Chakrabarti, Krishnendu Chatterjee, Thomas A. Henzinger, Orna Kupferman, Rupak Majumdar: Verifying Quantitative Properties Using Bound Functions. LNCS 3725, Springer-Verlag, CHARME 2005: 50-64.
5. Krishnendu Chatterjee, Laurent Doyen, Thomas A. Henzinger: Expressiveness and Closure Properties for Quantitative Languages. LNCS Logical Methods in Computer Science 6(3) (2010).
6. Edmund M. Clarke, E. Allen Emerson: Design and Synthesis of Synchronization Skeletons Using Branching-Time Temporal Logic. LNCS 131, Springer-Verlag, Logic of Programs 1981: 52-71.
7. E. Allen Emerson, Edmund M. Clarke: Characterizing Correctness Properties of Parallel Programs Using Fixpoints. LNCS 85, Springer-Verlag ICALP 1980: 169-181.
8. Peter Faymonville, Bernd Finkbeiner, Doron Peled: Monitoring Parametric Temporal Logic. VMCAI 2014: 357-375.
9. Vojtech Forejt, Marta Z. Kwiatkowska, Gethin Norman, David Parker: Automated Verification Techniques for Probabilistic Systems. LNCS 6659, Springer-Verlag, SFM 2011: 53-113.

10. Yonit Kesten, Amir Pnueli, Li-on Raviv: Algorithmic Verification of Linear Temporal Logic Specifications. ICALP 1998: 1-16.
11. Amir Pnueli, Aleksandr Zaks: PSL Model Checking and Run-Time Verification Via Testers. FM 2006: 573-586.
12. Jean-Pierre Queille, Joseph Sifakis: Iterative Methods for the Analysis of Petri Nets. LNCS 51, Springer-Verlag, Selected Papers from the First and the Second European Workshop on Application and Theory of Petri Nets 1981: 161-167.