Constraint-Based Agents:
A Formal Model for Agent Design

Alan K. Mackworth (mack@cs.ubc.ca)
Lab for Computational Intelligence, Department of Computer Science, University of British Columbia, Vancouver, B.C. V6T 1Z4, Canada

Ying Zhang (yzhang@parc.xerox.com)
Systems and Practices Lab, Xerox Palo Alto Research Center, Palo Alto, CA 94304, USA

Abstract.
Formal models for agent design are important for both practical and theoretical reasons. The Constraint-Based Agent (CBA) model includes a set of tools and methods for specifying, designing, simulating, building, verifying, optimizing, learning and debugging controllers for agents embedded in an active environment. The agent and the environment are modelled symmetrically as, possibly hybrid, dynamical systems in Constraint Nets. This paper is an integrated presentation of the development and application of the CBA framework, emphasizing the important special case where the agent is an online constraint-satisfying device. Using formal modeling and specification, it is often possible to verify complex agents as obeying real-time temporal constraint specifications and, sometimes, to synthesize controllers automatically. In this paper, we take an engineering point of view, using requirements specification and system verification as measurement tools for intelligent systems. Since most intelligent systems are real-time dynamic systems, the requirements specification must be able to represent timed properties. We have developed timed ∀-automata for this purpose. We present this formal specification, examples of specifying requirements and a general procedure for verification. The CBA model demonstrates the power of viewing constraint programming as the creation of online constraint-solvers for dynamic constraints.

Keywords: formal specification, agent design, artificial intelligence, robot architectures, constraint-based requirements, system verification, automata

1. Introduction and Motivation

Intelligent systems are now becoming ubiquitous, both in our daily lives and in extraordinary missions, such as planetary exploration. But, in stark contrast to other engineering disciplines, there has been little effort towards developing sound, deep and useful foundations for quantitative or qualitative performance specification, measurement and evaluation. The lack of rigorous performance criteria may lead, at best, to unsatisfactory behavior in certain environments or, at worst, to catastrophic failure in life-critical circumstances. Many researchers have suggested measures of performance for intelligent systems, such

© 2001 *Kluwer Academic Publishers. Printed in the Netherlands.*

as the Turing Test [20], Newell's criteria [16] and Albus' definition of intelligence [1]. However, most of these measures are not based on formal quantitative metrics. Partly to address this need, efforts have been made to compare performance on particular tasks, such as soccer [10, 19]. However, these methods are often domain-specific therefore hard to apply to general cases.

We advocate using formal methods to specify performance requirements for intelligent systems. In this paper we present a formal model for specifying, designing, building and evaluating embedded intelligent systems: Constraint-Based Agents (CBA). Much research has been done on formal methods [6] over the last twenty years. In this paper, we explore one of the approaches, namely, using timed $\forall$-automata for specifying performance requirements for constraint-based agents. The timed $\forall$-automata model was developed in [21, 28, 33] as an extension of discrete time $\forall$-automata [15] to continuous time, with annotations for real-time. Timed $\forall$-automata are simple yet able to represent many important features of dynamic systems such as safety, stability, reachability and real-time response.

Constraint programming has evolved several powerful frameworks for building problem-solvers as constraint-satisfying devices. Primarily, these devices are offline problem-solvers. For example, the Constraint Satisfaction Problem (CSP) paradigm has evolved and matured over the last twenty-five years. The algorithms developed in the CSP paradigm were made more available and more useful when they were incorporated into the Constraint Programming (CP) language paradigms. Despite this success, however, a major challenge still facing the constraint research community is to develop useful theoretical and practical tools for the constraint-based design of embedded intelligent systems. Many applications require us to develop online constraint-satisfying systems that function in a dynamic, coupled environment [12]. An archetypal example of an application in this class is the design of controllers for sensory-based robots [24, 19, 13]. If we examine this problem we see that almost all the tools developed to date in the CSP and CP paradigms are inadequate for the task, despite the superficial attraction of the constraint-based approach. The fundamental difficulty is that, for the most part, the CSP and CP paradigms still presume a disembodied, offline model of computation.

Consider an agent coupled to its active environment as shown in Figure 1. Each is an open dynamic system in its own right, acting on, and reacting to, the other. The coupled pair form a closed system that evolves over time.

To deal with such embedded applications, we must radically shift our perspective on constraint satisfaction from the offline model in

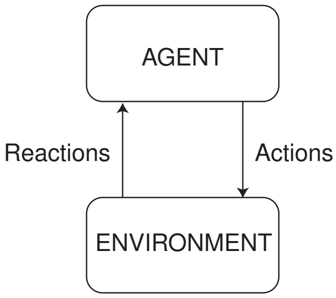

Figure 1. An agent interacting with its environment

which a solution is a function of pre-given static inputs to an online model where a solution is a temporal trace of values, resulting from a transduction of the input trace over time. Values in the agent's input trace may depend on earlier values in its output trace. In fact, the input trace for the agent is itself a causal mapping of its output trace, representing the dynamics of the environment, as shown in Figure 1.

Intelligent systems embedded as controllers in real or virtual systems must be designed in an online model based on various time structures: continuous, discrete and event-based. The requisite online computations, or transductions, are to be performed over various type structures including continuous and discrete domains. These hybrid systems require new models of computation, constraint satisfaction and constraint programming. To this end, we have defined constraint satisfaction as a dynamic system process that approaches asymptotically the solution set of the given, possibly time-varying, constraints [23]. Under this view, constraint programming is the creation of a dynamic system with the required property.

In this paper we present an integrated view of our approach, called Constraint-Based Agents. In the rest of this paper, we first discuss the design of embedded agents. The essence of the CBA model consists of a symmetrical coupling of an agent and its active environment. We then describe the Constraint Net model. We say that an agent is constraint-based if its behavior satisfies a specification in a well-defined constraint-based temporal logic or timed ∀-automaton. We then introduce the formal definition of timed ∀-automata and present examples of timed ∀-automata for representing performance metrics, and describe a general verification procedure for this type of requirements specification. We

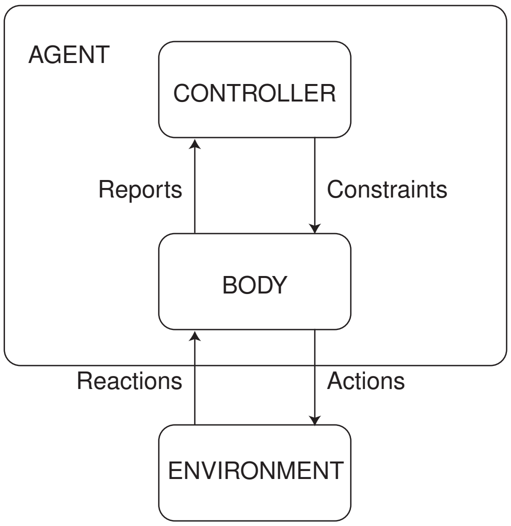

Figure 2. The structure of a constraint-based agent system

finally describe the use of the CBA framework in robot soccer and some related work, finishing with our conclusions.

2. Agents in the World

The most obvious artificial agents in the world are robots. But the CBA approach applies equally to embedded devices, pure software agents and natural animate agents. There are many ways of using a CBA model, including the embedded mode, simulation mode, verification mode, optimization mode, learning mode and design mode [17, p. 449]. The agent design problem is formidable, regardless of whether the agent is designed or modified by a human (programming), by nature (evolution), by another agent (bootstrapping), or by itself (learning). An agent is, typically, a hybrid intelligent system, consisting of a controller coupled to its body as shown in Figure 2.

The controller and the body each consist of discrete-time, continuous-time or event-driven components operating over discrete or continuous domains. The controller has perceptual subsystems that can (partially) observe the state of the body, and, through it, the state of the environment.

Parenthetically, the 'body' of an agent is simply the direct interface of the agent to its environment. The body executes actions in the environment, senses state in the environment (which may well cause state changes in the body) and reports to the controller. In the case of a robotic agent the body consists of one or more physical systems but in the case of an embedded software agent, the body is simply the software module that directly interfaces to the virtual or physical environment. Control theorists typically call the body the 'plant'. Some models do not differentiate between the body and the environment; we prefer to make that differentiation, based on the distinction between what is directly, and what is indirectly, controlled.

Agent design methodologies are evolving dialectically [10]. The symbolic methods of 'Good Old Fashioned Artificial Intelligence and Robotics' (GOFAIR) constitute the original thesis. The antithesis is reactive 'Insect AI' and control theory. The emerging synthesis, Situated Agents, has promise, but needs formal rigor and practical tools [18, 9, 7, 10, 24, 13, 14].

In 1992, Mackworth proposed robot soccer as a grand challenge problem [10] since it has the task characteristics that force us to confront the fundamental issues of agent design in a practical way for a perceptual, collaborative, real-time task with clear performance criteria. At the same time, Mackworth described the first system for playing robot soccer. Since then it has been a very productive testbed both for our laboratory [5, 19, 11, 27, 30, 29, 32] and for many other groups around the world, stimulating research toward the goal of building perceptual agents.

3. The Constraint Net Model

The Constraint Net (CN) model [25] was developed by Ying Zhang and Mackworth as a model for building hybrid intelligent systems as Situated Agents. In CN, an agent system is modelled formally as a symmetrical coupling of an agent with its environment. Even though an agent system is, typically, a hybrid dynamic system, its CN model is unitary. Most other agent and robot design methodologies use hybrid models of hybrid systems, awkwardly combining offline computational models of high-level perception, reasoning and planning with online models of low-level sensing and control.

CN is a model for agent systems software implemented as modules with I/O ports. A module performs a transduction from its input traces to its output traces, subject to the principle of causality: an output value at any time can depend only on the input values before, or at, that time. The model has a formal semantics based on the least fixpoint of sets of equations [25]. In applying it to an agent operating in a given environment, one separately specifies the behavior of the agent body, the agent control program, and the environment. The total system can then be shown to have various properties, such as safety and liveness, based on provable properties of its subsystems. This approach allows one to specify and verify models of embedded control systems. Our goal is to develop it as a practical tool for building real, complex, sensor-based agents. It can be seen as a formal development of Brooks' subsumption architecture [7] that enhances its reliability, modularity and scalability while avoiding the limitations of the augmented finite state machine approach, combining proactivity with reactivity.

An agent situated in an environment is modelled as three machines: the agent body, the agent controller and the environment, as shown above in Figure 2. Each is modelled separately as a dynamical system by specifying a CN with identified input and output ports. The agent is modelled as a CN consisting of a coupling of its body CN and its controller CN by identifying corresponding input and output ports. Similarly the agent CN is coupled to the environment CN to form a closed agent-environment CN, as shown above in Figure 1.

The CN model is realized as an online dataflow-like distributed programming language with a formal algebraic denotational semantics and a specification language, a real-time temporal logic, that allows the designer to specify and prove properties of the situated agent by proving them of the agent-environment CN. We have shown how to specify, design, verify and implement systems for a robot that can track other robots [22], a robot that can escape from mazes and a two-handed robot that assembles objects [24], an elevator system [31] and a car-like robot that can plan and execute paths under non-holonomic constraints [27].

Although CN can carry out traditional symbolic computation online, such as solving Constraint Satisfaction Problems and path planning, notice that much of the symbolic reasoning and theorem-proving may be outside the agent, in the mind of the designer, for controller synthesis and verification. GOFAIR does not make this distinction, assuming that such symbolic reasoning occurs explicitly in, and only in, the mind of the agent.

The question "Will the agent do the right thing?" [24] is answered positively if we can:

1. model the coupled agent system at a suitable level of abstraction,

2. specify the required global properties of the system's evolution, and

3. verify that the model satisfies the specification.

In CN the modelling language and the specification language are totally distinct since they have very different requirements. The modelling language is a generalized dynamical system language. Two versions of the specification language, Timed Linear Temporal Logic [27] and Timed $\forall$-automata [23], have been developed with appropriate theorem-proving and model-checking techniques for verifying systems. In [17, Chapter 12] we describe how to build a situated robot controller using CN as realized in a logic program.

4. Constraint-Satisfying Controllers

Many agents can be designed as online constraint-satisfying devices [23, 26, 27]. A robot in this restricted scheme can be verified more easily. Moreover, given a constraint-based specification and a model of the body and the environment, automatic synthesis of a correct constraint-satisfying controller sometimes becomes feasible, as shown for a simple goal-scoring robot in [27].

As a simple example, in Figure 2 suppose the CONTROLLER is a thermostat turning on or off a furnace, the BODY, that is heating the ENVIRONMENT. The goal of the system is to make the temperature of the ENVIRONMENT, $T_E(t)$, equal to a desired temperature, $T_D(t)$. In other words the CONTROLLER of the AGENT is trying to solve the constraint $T_E = T_D$. One version of CONTROLLER correctness is established if we can prove that the (thermal) dynamics of the coupled AGENT-ENVIRONMENT system satisfy the temporal logic formula $\Diamond\Box|T_E - T_D| < \epsilon$ where $\Diamond$ can be read as 'eventually' and $\Box$ can be read as 'always'. In other words, the system will, no matter how disturbed, eventually enter, and remain within, an ϵ-neighborhood of the solution manifold of the constraint. A less restrictive form of correctness corresponds to the specification $\Box\Diamond|T_E - T_D| < \epsilon$ which is to say that the system will always return, asymptotically, to the constraint solution manifold if it should happen to leave it.

A constraint is simply a relation on the phase space of the agent system, which is the product of the controller, body and environment spaces. A controller is defined to be *constraint-satisfying* if it, repeatedly, eventually drives the system into an ϵ-neighborhood of the constraint using a constraint satisfaction method such as gradient descent or a symbolic technique.

A constraint-satisfying controller may be *hierarchical* with several layers of controller above the body, as shown in Figure 3. In this case, each layer must satisfy the constraints, defined on its state variables, appropriate to the layer, as, typically, set by the layer above. The layers below each layer present to that layer as a virtual agent body in a suitably abstract state space [27, 29]. The lower layers are, typically,

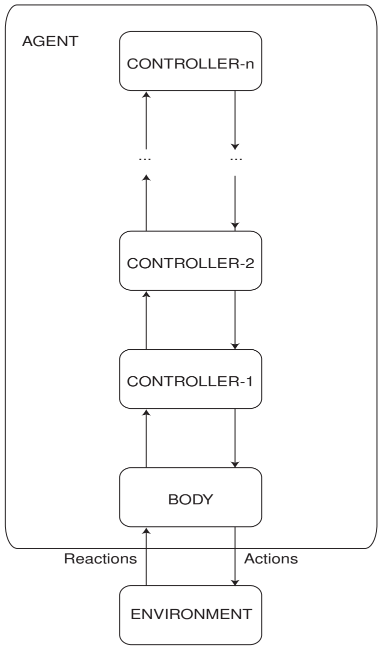

Figure 3. A hierarchical agent controller

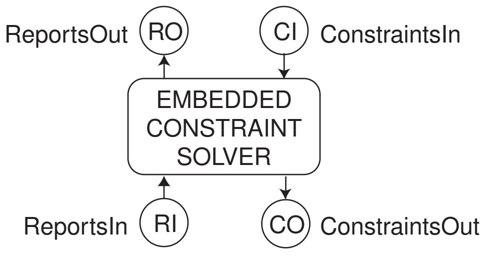

Figure 4. A layer in a constraint-based controller

reactive and synchronous (or in continous time) on continuous state spaces; the upper layers are more deliberative and asynchronous (or event-triggered) in symbolic, discrete spaces.

A typical layer in a hierarchical controller is shown in Figure 4.

Each layer has two external inputs: the trace of constraint requests coming from above $ConstraintsIn$ (CI) and the reports coming from below $ReportsIn$ (RI). Its two outputs are its reports to the level above $ReportsOut$ (RO) and its constraint requests to the level below

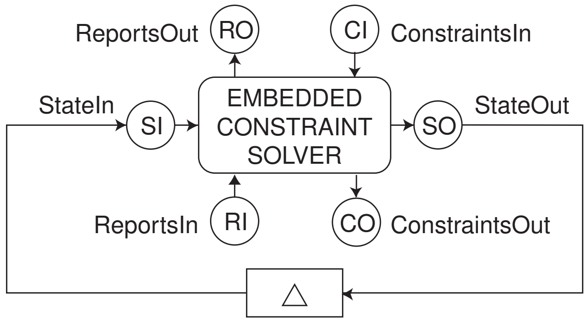

Figure 5. A layer with state in a constraint-based controller

ConstraintsOut (CO). These traces arise from *causal transductions* of the external inputs:

$$CO = C_t(CI, RI) \tag{1}$$
$$RO = R_t(CI, RI) \tag{2}$$

If the constraint-solver can be represented as a state-based solver then the layer may be represented as shown in Figure 5.

Here, for simplicity, a discrete-time state-based layer is shown. It produces an extra output *StateOut* (SO) that is consumed as an extra input *StateIn* (SI) after a unit delay (Δ). In this case the behavior of the layer may be represented by computing the values of each of the three outputs as a *transliteration* (function) of the current values of the three inputs:

$$CO(t) = C_f(CI(t), RI(t), SI(t)) \tag{3}$$
$$RO(t) = R_f(CI(t), RI(t), SI(t)) \tag{4}$$
$$SO(t) = S_f(CI(t), RI(t), SI(t)) \tag{5}$$
$$SI(t+1) = SO(t) \tag{6}$$

5. Timed ∀-automata

In this section we present a formal model for specifying the behavior of constraint-based agents, timed ∀-automata. In general, there are two uses of automata: first, to describe computations, for example, input/output state automata, and second, to characterize a set of sequences, for example, regular grammars/languages. Examples of the first category are mostly deterministic and examples of the second category are mostly non-deterministic. However, all the original automata work is based on discrete time steps/sequences. Approaches to extending automata to continuous time have been explored in the hybrid systems community [2, 3, 8]. The timed $\forall$-automata model that we developed belongs to the second category, i.e., *non-deterministic finite* state automata specifying behaviors over *continuous* time. The discrete time version of $\forall$-automata was originally proposed as a formalism for the specification and verification of temporal properties of concurrent programs [15].

5.1. Syntax

Syntactically, a [timed] $\forall$-automaton is defined as follows.

DEFINITION 1 (Manna and Pnueli, 1987). *A $\forall$-automaton A is a quintuple (Q, R, S, e, c) where Q is a finite set of automaton-states, $R \subseteq Q$ is a set of recurrent states and $S \subseteq Q$ is a set of stable states. With each $q \in Q$, we associate an assertion $e(q)$, which characterizes the entry condition under which the automaton may start its activity in q. With each pair $q, q' \in Q$, we associate an assertion $c(q, q')$, which characterizes the transition condition under which the automaton may move from q to q'.*

R and S are generalizations of accepting states. We denote by $B = Q - (R \cup S)$ the set of non-accepting (bad) states. Let $\mathcal{R}^+$ be the set of non-negative real numbers representing time durations.

DEFINITION 2. *A timed $\forall$-automaton is a triple (A, T, τ) where A is a $\forall$-automaton, $T \subseteq Q$ is a set of timed automaton-states and $\tau : T \cup B \to \mathcal{R}^+ \cup \{\infty\}$ is a time function.*

One of the engineering advantages of using automata as a specification language is its graphical representation. It is useful and illuminating to represent timed $\forall$-automata by diagrams. A timed $\forall$-automaton can be depicted by a labeled directed graph, where automaton-states are depicted by circle nodes and transition relations by directed arcs. In addition, each automaton-state may have an entry arc pointing to it. Each recurrent state is depicted by a diamond and each stable state is depicted by a square, inscribed within a circle. Nodes and arcs are labeled by assertions as follows. A node or an arc that is left unlabeled is considered to be labeled with **true**. Furthermore,

1. if an automaton-state q is labeled by ψ and its entry arc is labeled by ϕ, the entry condition $e(q)$ is given by $e(q) = \psi \wedge \phi$; if there is no entry arc, $e(q) = \textbf{false}$, and

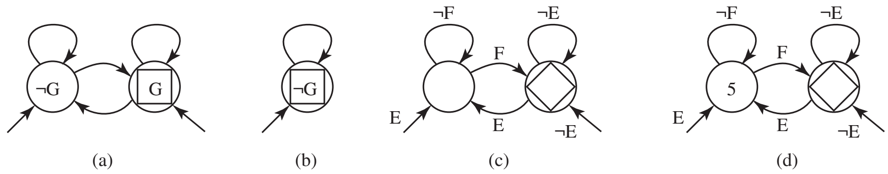

Figure 6. Examples of timed ∀-automata

2. if arcs from q to q' are labeled by ϕ_i, $i = 1...n$, and q' is labeled by ψ, the transition condition $c(q, q')$ is given by $c(q, q') = (\phi_1 \vee ... \vee \phi_n) \wedge \psi$; if there is no arc from q to q', $c(q, q') = \mathbf{false}$.

A T-state is denoted by a nonnegative real number indicating its time bound. Some examples of timed ∀-automata are shown in Figure 6.

5.2. Semantics

Semantically, each assertion denotes a constraint defined on a domain of interest. Let D be a domain of interest; D can be finite, discrete, or continuous, or a cross product of a finite number of domains. Physically, D can represent, for example, velocities, distances, torques, sentences, commands or a combination of the above. A *constraint* C defined on D is a subset of $D, C \subseteq D$. Physically, a constraint represents a certain relation on a domain, such as a relation between external environment stimuli and an agent's internal knowledge representation, or, a relation between the agent's internal states and its actions, or, the relation between the current and next state. An element d in domain D *satisfies* constraint C, if and only if $d \in C$.

The semantics of a timed ∀-automaton is defined as follows. Let $\mathcal{T}$ be a time domain, which can be continuous, for example, $\mathcal{R}^+$. First, let us define runs of ∀-automata. Let $A = (Q, R, S, e, c)$ be a ∀-automaton and $v : \mathcal{T} \to D$ be a function of time, a *trace* of the agent-environment system. A *run* of A over v is a function $r : \mathcal{T} \to Q$ satisfying:

1. *Initiality*: $v(0) \in e(r(0))$;

2. *Consecution*:

   a) *Inductivity*: $\forall t > 0, \exists q \in Q, t' < t, \forall t'', t' \leq t'' < t, r(t'') = q$ and $v(t) \in c(r(t''), r(t))$ and

   b) *Continuity*: $\forall t, \exists q \in Q, t' > t, \forall t'', t < t'' < t', r(t'') = q$ and $v(t'') \in c(r(t), r(t''))$.

When $\mathcal{T}$ is discrete, the two conditions in *Consecution* reduce to one, i.e., $\forall t > 0, v(t) \in c(r(pre(t)), r(t))$ where $pre(t)$ is the previous time point of t.

If r is a run, let $Inf(r)$ be the set of automaton-states appearing infinitely many times in r, i.e., $Inf(r) = \{q | \forall t \exists t' \geq t, r(t') = q\}$. A run is called *accepting* if and only if

1. $Inf(r) \cup R \neq 0$, that is, some of states appearing infinitely many times in r belong to R, or

2. $Inf(r) \subseteq S$, that is, all the states appearing infinitely many times in r belong to S.

For a timed $\forall$-automaton, in addition for a run to be accepting, it has to satisfy time constraints. Let $I \subseteq \mathcal{T}$ be a time interval and $|I|$ be the time measurement, and let $r|I$ be a segment of r over time interval I. A run satisfies time constraints if and only if:

1. *Local*: For any $q \in T$ and any time interval I, if $r|I$ is a segment of consecutive states of q, then $|I| \leq \tau(q)$;

2. *Global*: For any time interval I, if $r|I$ is a segment of consecutive states of $B \cup S$, then $\int_I \chi_B(r(t))dt \leq \tau(B)$, where $\chi_B : Q \to \{0, 1\}$ is the characteristic function for the set B.

DEFINITION 3. *A timed $\forall$-automaton $TA = (A, T, \tau)$ accepts a trace v, if and only if*

1. all runs are accepting for A; and

2. all runs satisfy the time constraints.

With the semantics defined, we can infer that, for the timed $\forall$-automata in Figure 6, (a) specifies the behavior of reachability, i.e., eventually the system should satisfy constraint G, (b) specifies the behavior of safety, i.e. constraint G is never satisfied, (c) specifies the behavior of bounded response, i.e., whenever constraint E is satisfied, constraint F will be satisfied within bounded time and (d) specifies the behavior of real-time response, i.e., whenever constraint E is satisfied, constraint F will be satisfied within 5 time units.

6. Examples of Performance Specification

Timed $\forall$-automata are a simple yet powerful model for the specification of behaviors of dynamic systems, since it integrates constraint specification with timed dynamic behavior specification.

6.1. Examples of Constraint Specification

Constraint specification alone can specify many performance metrics. As we said, constraints can specify relations between external environment stimuli and an agent's internal knowledge representation, or between an agent's internal states and its actions, or between the current and next states. Constraints can be finite, discrete or continuous, or any combination of the above. Constraints can be linear, nonlinear, equalities or inequalities. Moreover, constraints can also specify optimal conditions or optimality with extra constraints, or combinations of multiple optimal criteria and additional constraints.

Considering the following examples for specifying constraints:

1. *Inequality*: $f(x) \leq 0$ where x is a vector of variables and f is a vector of functions.
2. *Optimality*: $min|f(x)|$ where $|x|$ is a norm for x.
3. *Negation*: $x \neq y$.
4. *Constrained Optimality*: $min|f(x)|$ given $g(x) \leq 0$.
5. *Robustness*: Let $f(x)$ be a set of output functions with x as inputs. The robustness can be represented by its Jacobian J. There are many ways to state an optimal condition for robustness. One method is to minimize $|w|$ where w is the vector of diagonal elements of W in the singular value decomposition of $J = UWV^T$.

6.2. Examples of $\forall$-Automata

With automata, timed dynamic behaviors can be specified. Here is a set of examples for specifying performance using timed $\forall$-automata, as shown in Figure 6:

1. Let G be a constraint that the distance between the robot and its desired position is less than some constant value. Then Figure 6(a) specifies that the robot will eventually arrive at its desired position.

2. Let G be a constraint that the error of a learning algorithm is less than a desired tolerance. Then Figure 6(a) specifies that the learning will eventually converge. If we let the state of $\neg G$ in Figure 6(a) be a timed state with time bound t, it further specifies that the learning will be done within time t.

3. Let G be a constraint that the distance between the robot and obstacles is less than some constant value. Then Figure 6(b) specifies that the robot will never hit any obstacle. If G denotes that the current memory usage is over the limit, Figure 6(b) specifies that the memory usage at any time is within its limit.

4. Let E be an external stimulus and F be a response. Then Figure 6(c) specifies that there is a response after the stimulus within bounded time. Figure 6(d) specifies that such a response occurs within 5 time units.

Even though timed $\forall$-automata are powerful, still they are not able to represent all forms of performance metrics. For example, optimal performance over time $min \int f(t)dt$ is not specifiable with timed $\forall$-automata. This form is mostly used for characterizing energy, efficiency or overall errors. Furthermore, specifications with probability behaviors are not included either. However, it is possible to add probability, for example, instead of "all runs" must be accepting and satisfying time constraints, we can say "$x\%$ of the runs" must be accepting and satisfy the timing constraints.

6.3. Performance Comparisons

Note that a requirements specification defines what the system should do, not how the system should be organized – its architecture. For example, behavior-based control [1, 7] (which is arbitration-based or a horizontal hierarchy) has a different form of architecture from function-based control [4] (which is abstraction-based or a vertical hierarchy); model-based systems have a different form of architecture than learning-based systems, event-driven systems differ from time-driven systems. Different systems with different architectures can still be compared based on the behavioral interface under the formal performance specification. For example, given a set of requirements specifications Rs, suppose system A satisfies a subset $As \subseteq Rs$ and system B satisfies a subset $Bs \subseteq Rs$. If $As \subseteq Bs$, system A is no better than system B with respect to requirements Rs. Similarly, if system A satisfies requirement α and system B satisfies requirement β and if $\alpha \to \beta$, system A is no worse than system B with respect to the requirements.

However, a requirements specification does not define a metric on architectures. The measurement of performance should come from the customer's point of view, but the measurement of architecture should come from the developer's point of view, considering factors such as design time, debug time, upgrade time, modularity and the percentage of re-usable components.

7. System Verification

For most dynamic systems, stability or convergence is the most important property that needs to be verified. For example, we can verify that equation $dx/dt = 0$ satisfies the property of the $\forall$-automaton in Figure 6(a) with G as $|x'| \leq \epsilon$ for any positive number ϵ. The most commonly used method for the verification of such properties is the use of Lyapunov functions. We developed a formal method based on model-checking, that generalizes Lyapunov functions [21, 28]. This method is automatic if the domain of interest is finite discrete and time is discrete [21].

The details of the model-checking method are out of the scope of this paper. The basic principle is to first find a set of invariants, each associated with an automaton-state in the timed $\forall$-automaton. Then, find a set of Lyapunov functions, which are non-increasing in stable states and decreasing in bad states. Finally, find a set of local and global timing functions, where local timing functions are decreasing in timed states and global timing functions, like Lyapunov functions, are non-increasing in stable states and decreasing in bad states, in addition to having bounded values.

8. Robot Soccer Players

The CBA framework has also been motivated, developed and tested by application to the experimental challenge of designing, building and verifying controllers with perceptual systems for robot soccer players with both off-board and on-board vision systems.

In the Dynamo (Dynamics and Mobile Robots) project in our laboratory, we have experimented, since 1991, with multiple mobile robots under visual control. The Dynamite testbed consists of a fleet of radio-controlled vehicles that receive commands from a remote computer. Using our custom hardware and a distributed MIMD environment, vision programs are able to monitor the position and orientation of each robot at 60 Hz; planning and control programs generate and send motor commands at the same rate. This approach allows umbilical-free behavior and very rapid, lightweight fully autonomous robots. Using this testbed we have demonstrated various robot tasks [5], including playing soccer [19] using a 2-layer deliberative/reactive controller architecture.

One of the Dynamo robots, Spinoza, is a self-contained robot consisting of an RWI base with an RGB camera on a pan-tilt platform mounted as its head and a trinocular stereo camera in its base. As an illustration of these ideas, consider the task for Spinoza of repeatedly finding, tracking, chasing and kicking a soccer ball, using the pan-tilt camera. After locating the moving ball Spinoza is required to track it, move to within striking distance of the ball and strike it. The available motor commands control the orientation of the base, the forward movement of the base, and the pan and tilt angles of the camera. The parameters can be controlled in various relative/absolute position modes or rate mode. The available rate of pan substantially exceeds the rate of base rotation. A hierarchical constraint-based active-vision controller, using prioritized constraints and constraint arbiters, can be specified for Spinoza that will, repeatedly, achieve and maintain (or re-achieve) the desired goal subject to safety conditions such as staying inside the soccer field, avoiding obstacles and not accelerating too quickly. If the dynamics of Spinoza and the ball are adequately modelled by the designer then this constraint-based vision system will be guaranteed to achieve its specification.

Yu Zhang and Mackworth have extended these ideas to build 3-layer constraint-satisfying controllers for a complete soccer team [32]. The controllers for our softbot soccer team, UBC Dynamo98, are modelled in CN and implemented in Java, using the Java Beans architecture [29]. They control the soccer players' bodies in the Soccer Server developed by Noda Itsuki for RoboCup. These experiments provide evidence that the constraint-based CN approach is a clean and practical design framework for perceptual robots.

9. Related Work and Conclusion

Much work has been done in formal approaches to system specification and verification [2, 3, 8, 15]. In general, there are two schools. One is to develop a uniform specification for both systems and their requirements; the other is to use two different specifications, one for systems and one for requirements. The advantage of the former is that the same formal approach can apply to both system synthesis and system verification. However, in most cases, if the specification language is powerful for both systems and requirements, the synthesis or verification tasks become hard. We advocate the latter approach, that is, using timed ∀-automata for requirements specification and using Constraint Nets [21, 28, 25] for system modeling. Control synthesis [21, 27] and verification [21, 23, 24, 28, 32] are also studied in this framework.

The Constraint-Based Agent approach is a framework for the specification, design, analysis, implementation and validation of artificial and natural agent systems. It requires a new model of online and embedded computation for Constraint Programming such as Constraint Nets.

In this paper, we have also shown how to use formal methods to specify the performance metrics of intelligent systems, with timed ∀-automata as an example. The advantage of formal methods over other methods lies in their precision and generality. Timed ∀-automata, with its graphical depiction and constraint specification, is a simple yet powerful formalism for specifying many properties of dynamic systems.

10. Acknowledgments

We are grateful to Rod Barman, Cullen Jennings, Stewart Kingdon, Jim Little, Valerie McRae, Don Murray, Dinesh Pai, David Poole, Michael Sahota, Vlad Tucakov and Yu Zhang for help with this. This work is supported, in part, by the Natural Sciences and Engineering Research Council of Canada and the Institute for Robotics and Intelligent Systems Network of Centres of Excellence. Alan Mackworth holds a Canada Research Chair in Artificial Intelligence.

References

1. Albus, J. S.: 1991, 'Outline for a Theory of Intelligence'. In: *IEEE Transactions on Systems, Man, and Cybernetics*, Vol. 21. IEEE, pp. 473–509.
2. Alur, R., C. Courcoubetis, T. A. Henzinger, and P. Ho: 1993, 'Hybrid Automata: An Algorithmic Approach to the Specification and Verification of Hybrid Systems'. In: R. L. Grossman and A. Nerode and A. P. Ravn and H. Rischel (ed.): *Hybrid Systems*, No. LNCS 736. Springer-Verlag, pp. 209–229.
3. Alur, R. and D. Dill: 1990, 'Automata for Modeling Real-Time Systems'. In: M. S. Peterson (ed.): *ICALP90: Automata, Languages and Programming*, No. LNCS443. Springer-Verlag, pp. 322–335.
4. Arkin, R. C.: 1998, *Behavior-Based Robotics*. Cambridge, MA: The MIT Press.
5. Barman, R. A., S. J. Kingdon, J. J. Little, A. K. Mackworth, D. K. Pai, M. Sahota, H. Wilkinson, and Y. Zhang: 1993, 'Dynamo: Real-time experiments with multiple mobile robots'. In: *Intelligent Vehicles Symposium*. Tokyo, pp. 261–266.
6. Bowen, J. P.: 2001, 'Formal Methods Web Site', `http://www.afm.sbu.ac.uk/`.
7. Brooks, R. A.: 1991, 'Intelligence without reason'. In: *IJCAI-91*. Sydney, Australia, pp. 569–595.

8. Henzinger, T. A., Z. Manna, and A. Pnueli: 1991, 'Timed Transition Systems'. In: J. W. deBakker and C. Huizing and W. P. dePoever and G. Rozenberg (ed.): *Real-Time: Theory in Practice*, No. LNCS 600. Springer-Verlag, pp. 226–251.
9. Lavignon, J. and Y. Shoham: 1990, 'Temporal Automata'. Technical Report STAN-CS-90-1325, Stanford University, Stanford, CA.
10. Mackworth, A. K.: 1993, 'On Seeing Robots'. In: A. Basu and X. Li (eds.): *Computer Vision: Systems, Theory, and Applications*. Singapore: World Scientific Press, pp. 1–13.
11. Mackworth, A. K.: 1996, 'Quick and Clean: Constraint-Based Vision for Situated Robots'. In: *IEEE Int'l. Conf. on Image Processing*. Lausanne, Switzerland, pp. 789–792.
12. Mackworth, A. K.: 1997, 'Constraint-Based Design of Embedded Intelligent Systems'. *Constraints* **2**(1), 83–86.
13. Mackworth, A. K.: 1999, 'The Dynamics of Intelligence: Constraint-Satisfying Hybrid Systems for Perceptual Agents'. In: *Hybrid Systems and AI: Modeling, Analysis and Control of Discrete and Continuous Systems*. Stanford, CA, pp. 210–214.
14. Mackworth, A. K.: 2000, 'Constraint-Based Agents: The ABCs of CBAs'. In: *Proc. 6th Int. Conf. on Principles and Practice of Constraint Programming – CP2000*. Springer LNCS 1894, Singapore, pp. 1–10.
15. Manna, Z. and A. Pnueli: 1987, 'Specification and Verification of Concurrent Programs by ∀-automata'. In: *Proc. 14th Annual ACM Symposium on Principles of Programming Languages*. pp. 1–12.
16. Newell, A.: 1982, 'The Knowledge Level'. *Artificial Intelligence* **18**((1)), 87–127.
17. Poole, D. L., A. K. Mackworth, and R. G. Goebel: 1998, *Computational Intelligence: A Logical Approach*. New York: Oxford University Press.
18. Rosenschein, S. J. and L. P. Kaelbling: 1986, 'The synthesis of machines with provable epistemic properties'. In: Joseph Halpern (ed.): *Proc. Conf. on Theoretical Aspects of Reasoning about Knowledge*. Los Altos, CA: Morgan Kaufmann, pp. 83–98.
19. Sahota, M. and A. K. Mackworth: 1994, 'Can Situated Robots Play Soccer?'. In: *Proc. Artificial Intelligence 94*. Banff, AB, pp. 249–254.
20. Turing, A. M.: 1963, 'Computing machinery and intelligence'. In: E. Feigenbaum and J. Feldman (eds.): *Computers and Thought*. New York, NY: McGraw-Hill, pp. 11–35.
21. Zhang, Y.: 1994, 'A Foundation for the Design and Analysis of Robotic Systems and Behaviors'. Ph.D. thesis, University of British Columbia, Vancouver, British Columbia.
22. Zhang, Y. and A. K. Mackworth: 1992, 'Modeling behavioral dynamics in discrete robotic systems with logical concurrent objects'. In: S. G. Tzafestas and J. C. Gentina (eds.): *Robotics and Flexible Manufacturing Systems*. Elsevier Science Publishers B.V., pp. 187–196.
23. Zhang, Y. and A. K. Mackworth: 1994a, 'Specification and Verification of Constraint-Based Dynamic Systems'. In: A. Borning (ed.): *Principles and Practice of Constraint Programming*, No. 874 in Lecture Notes in Computer Science. Springer-Verlag, pp. 229 – 242.
24. Zhang, Y. and A. K. Mackworth: 1994b, 'Will The Robot Do The Right Thing?'. In: *Proc. Artificial Intelligence 94*. Banff, AB, pp. 255–262.
25. Zhang, Y. and A. K. Mackworth: 1995b, 'Constraint Nets: A Semantic Model for Hybrid Dynamic Systems'. *Theoretical Computer Science* **138**, 211 – 239.

26. Zhang, Y. and A. K. Mackworth: 1995c, 'Constraint Programming in Constraint Nets'. In: V. Saraswat and P. Van Hentenryck (ed.): *Principles and Practice of Constraint Programming*. Cambridge, MA: The MIT Press, Chapt. 3, pp. 49–68.
27. Zhang, Y. and A. K. Mackworth: 1995d, 'Synthesis of Hybrid Constraint-Based Controllers'. In: P. Antsaklis, W. Kohn, A. Nerode, and S. Sastry (eds.): *Hybrid Systems II*, Lecture Notes in Computer Science 999. Springer Verlag, pp. 552 – 567.
28. Zhang, Y. and A. K. Mackworth: 1996, 'Specification and verification of hybrid dynamic systems with timed ∀-automata'. In: R. Alur and T. A. Henzinger and E. D. Sontag (ed.): *Hybrid Systems III, Verification and Control*. Springer, pp. 587–603.
29. Zhang, Y. and A. K. Mackworth: 1998a, 'A Constraint-based Controller for Soccer-playing Robots'. In: *Proceedings of IROS '98*. Victoria, BC, Canada, pp. 1290 – 1295.
30. Zhang, Y. and A. K. Mackworth: 1998b, 'Using Reactive Deliberation for Real-time Control of Soccer-playing Robots'. In: H. Kitano (ed.): *RoboCup-97: Robot Soccer World Cup 1*. pp. 508–512.
31. Zhang, Y. and A. K. Mackworth: 1999a, 'Modelling and Analysis of Hybrid Systems: An Elevator Case Study'. In: H. Levesque and F. Pirri (eds.): *Logical Foundations for Cognitive Agents*. Berlin: Springer, pp. 370–396.
32. Zhang, Y. and A. K. Mackworth: 1999b, 'A Multi-level Constraint-based Controller for the Dynamo98 Robot Soccer Team'. In: Minoru Asada and Hiroaki Kitano (ed.): *RoboCup-98: Robot Soccer World Cup II*. Springer, pp. 402–409.
33. Zhang, Y. and A. K. Mackworth: 2000, 'Formal Specification of Performance Metrics for Intelligent Systems', In: *Proc. Workshop on Performance Metrics for Intelligent Systems*. National Institute of Standards and Technology, Washington, DC, 2000, (5 pp.), (to appear).