The Journal of Systems and Software 85 (2012) 2697–2706

Contents lists available at SciVerse ScienceDirect

The Journal of Systems and Software

journal homepage: www.elsevier.com/locate/jss

Self-control of the time complexity of a constraint satisfaction problem solver program

Yönet A. Eracar[a], Mieczyslaw M. Kokar[b,*]

[a] Department of Mechanical and Industrial Engineering, Northeastern University, Boston, MA, USA
[b] Department of Electrical and Computer Engineering, Northeastern University, Boston, MA, USA

ARTICLE INFO

Article history:
Received 9 May 2011
Received in revised form 16 May 2012
Accepted 16 May 2012
Available online 24 May 2012

Keywords:
Self-controlling software
Constraint satisfaction problem
Branch and bound algorithm
PID controller
Job scheduling problem
Fixture design problem

ABSTRACT

This paper presents the self-controlling software paradigm and reports on its use to control the branch and bound based constraint satisfaction problem solving algorithm. In this paradigm, an algorithm is first conceptualized as a dynamical system and then a feedback control loop is added to control its behavior. The loop includes a Quality of Service component that assesses the performance of the algorithm during its run time and a controller that adjusts the parameters of the algorithm in order to achieve the control goal. Although other approaches – generally termed as "self-*" – make use of control loops, this use is limited to the structure of the software system, rather than to its behavior and its dynamics. This paper advocates the analysis of dynamics of any program with control loops. The self-controlling software paradigm is evaluated on two different NP-hard constraint satisfaction and optimization problems. The results of the evaluation show an improvement in the performance due to the added control loop for both of the tested constraint satisfaction problems.

1. Introduction

In this paper we present the self-controlling software paradigm and describe its use to control the behavior of an algorithm. We classify our approach as *self-controlling software* because (1) we treat software as a dynamical system, and (2) we use a feedback control loop to continuously monitor the software performance and adjust its parameters at run time. This approach is very closely related to the approach described in Hellerstein et al. (2004), however the main distinguishing factor of self-controlling software is that the dynamics is the feature of an algorithm and not of a computer system.

The self-controlling software paradigm was first proposed in 1999 (Kokar et al., 1999). The architecture proposed in Kokar et al. (1999) included three control loops, each based on a specific control paradigm – the feedback loop (based on feedback control, Doyle et al., 1990), the adaptation loop (based on adaptive control, Åström, 1989) and the reconfiguration loop (based on reconfigurable control, Shamma, 1996). The approach was then applied to control a scheduling algorithm (Kokar et al., 2001; Xun et al., 2004; Fescioglu-Unver and Kokar, 2005) and to the tabu search algorithm (Fescioglu-Unver and Kokar, 2008, 2008, 2011). This paper describes the application of the self-controlling software paradigm to a new domain (constraint satisfaction problem) and to a new algorithm (branch and bound search).

The self-controlling software approach is closely related to the approaches in which computer systems include feedback loops in their design. Those approaches are known under many different names: *self-adaptive systems* (Laddaga, 1999; Brun et al., 2009; Badr et al., 2002; Goldman et al., 1997; Musliner, 2000), *self-managing systems*, *autonomic systems* (Blueprint, 2006), *self-aware systems* (Sztipanovits and Karsai, 1997; Karsai and Sztipanovitz, 1999; Brandozzi and Perry, 2002; Reece, 2000; Robertson and Laddaga, 2004). Collectively, all of those approaches are referred to as *self-* or *self-** systems.

The early systems in this group were not patterned upon control theory. The idea of a feedback loop as an architectural pattern was adopted by the researchers in software architectures (Shaw, 1995; Klein et al., 1999; Oreizy et al., 1999; Garlan et al., 2001; wen Cheng et al., 2002; Pezze et al., 2008) and by autonomic computing (Blueprint, 2006). However, the adoption of the control theory's feedback loop metaphor has been only partial so far. For instance, the widely cited book on the control of computer systems (Hellerstein et al., 2004) states that "it is essential that models we construct consider time", i.e., the underlying plant is a dynamical system. A more popular explanation of control theory provided by Wikipedia also states that "Control theory is an interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamical systems." However, the fact that the system being controlled is a dynamical system cannot be found in the above

* Corresponding author.
E-mail address: mkokar@ece.neu.edu (M.M. Kokar).

0164-1212/$ – see front matter © 2012 Elsevier Inc. All rights reserved.
http://dx.doi.org/10.1016/j.jss.2012.05.060

referenced architecture based adaptation literature. At the same time, this literature contributes a great deal to the methodology for designing software systems that utilize feedback loops.

Combinatorial constraint satisfaction problems (CSP) are known to be NP-hard. An algorithmic solution of all the problem instances of such problems may not exist at all. However, for under-constrained cases a solution can be found easily. On the other hand, for over-constrained cases, it may be possible to infer that a solution does not exist. The most difficult solutions to find lie in the problem subspace called the *phase transition region* (e.g., Cheeseman et al., 1991; Selman et al., 1992). It is desirable that a constraint solver algorithm recognizes such a fact and abandons the search rather than pursuing the search which has a very low probability of finding a solution. The success then depends on deciding when to quit. In case an algorithm abandons the search prematurely, the event is called a *false alarm*.

The main difficulty in optimizing such algorithms lies in the fact that the behaviors of NP-hard problems are unpredictable. One way to deal with this kind of problem is to instrument such a system with "sensors" that monitor the performance of the search for constraint satisfaction solution and react to the changes in the observed performance. It is known, however, that too quick reaction to such changes, e.g., after each step, like in greedy search, are not optimal. It may be possible that the next step in the same direction would find a feasible solution, or at least that the search space is sufficiently regular and thus the continuation of the search in the same direction would get the algorithm closer to a feasible solution.

In this paper we ask the question whether such behaviors can be interpreted as the algorithm experiencing unpredictable *disturbances*, similarly as in control theory, where the controlled process (the *plant*) is subject to such disturbances. The second question is then whether such algorithms could be treated as plants of a control system and whether they could be controlled by traditional controllers, like a PID controller (Hellerstein et al., 2004). Since in control theory a plant is a dynamical system, if we want to use traditional controllers we have to conceptualize an algorithm as a dynamical system.

In this paper we present a control theory based approach to real-time adaptation of a CSP search algorithm so that the search is directed away from the regions where solutions do not exist and, as a result of this, the probability of false alarm is minimized. We demonstrate that an algorithm *can* be treated as a dynamical system and that a carefully designed feedback loop with an appropriately tuned PID controller has a positive impact on the performance of the algorithm. However, control engineering teaches us that an inappropriate feedback loop can cause even disastrous damage. Consequently, we suggest that all the "self-" approaches to controlling software should incorporate the analysis of the dynamics of the software being controlled.

For the proof of concept, Oz (Mehl et al., 1995) was selected as the target CSP solver. It uses a branch and bound algorithm to implement search. This algorithm was controlled by our feedback control loop that was added to the Oz processing path. The proof of concept system was tested against two experimental scenarios – a job scheduling problem and a fixture design problem. For both problems, phase transition invariants were identified empirically and later used to generate hard problems that show phase transition behavior. The proof of concept system and a benchmark algorithm were executed against these hard problems. The performance of the proof of concept system and the benchmark algorithm was measured and compared for different control parameters and initial conditions. The experiments with the proof of concept system showed that the complexity of the CSP solving algorithms can be controlled, and that the hard problems (phase transitions) can be detected with low probability of false alarm, where the benchmark algorithm may abort computation abruptly against those hard problems. For not so hard problems, the proof of concept system performed as well as the benchmark algorithm. The results of these experiments provide a good indication of the usability of the proposed approach, i.e., treating a search algorithm as a *control plant* and adding a feedback loop to adjust the search during its execution.

The paper is organized as follows. In Section 2 we describe the branch and bound search algorithm. In Section 3 we formalize the branch and bound algorithm as a dynamical system. This allows us to treat this algorithm as a control plant. The description of the control system is presented in Section 4. In Section 5, we describe the two CSP problems used for the verification of our approach – a job scheduling problem and a fixture design problem. In Section 6 we show the results of experiments in which our approach was used to control the branch and bound algorithm applied to the two CSP problems. This is followed by a review of the literature pertinent to this search algorithm and some attempts to control its execution (Section 7). And finally, in Section 8, we present our conclusions and suggestions for future research.

2. Branch and bound algorithm

In the work presented in this paper, automatically generated *constraint satisfaction problem* (CSP) code (in Oz) executes a search engine (Schulte, 2002; Van Roy and Haridi, 2004), one of the system modules of the Mozart Programming system. This search engine can solve CSPs with and without objective functions using a modified version of the branch and bound algorithm.

Schulte and Smolka define the basic components of the search engine used in Oz as follows. A CSP consists of a finite set of variables, each associated with a set of possible values, and a set of constraints. A solution to a CSP is an assignment of a value to each variable from its set of values that satisfies all the constraints. A constraint satisfaction solver needs to: (1) determine whether a solution exists, (2) find one or all solutions.

Formally, a CSP can be represented by a 3-tuple $\langle V, D, C\rangle$, where $V = \{v_1, \ldots, v_n\}$ is a set of decision variables, D is *domain*, i.e., a complete mapping from V to finite sets of integers $\{I_1, \ldots, I_m\}$ – *value sets* for the decision variables, and $C = \{C_1, \ldots, C_t\}$ is a set of constraints on the decision variables (Dechter, 1992). A constraint C_i contains a subset of the variables $var(C_i) = \{v_{i_1}, \ldots, v_{i_{j(i)}}\}$ and a relation rel_i, defined on this subset: $C_i = rel_i(v_{i_1}, \ldots, v_{i_{j(i)}}) \subseteq I_{i_1} \times \cdots \times I_{i_{j(i)}}$.

In this paper, we work with finite domains. A finite domain D maps all the variables to finite subsets (intervals) of non-negative integers. In other words, for a finite domain all the sets in $\{I_1, \ldots, I_m\}$ are finite intervals of non-negative integers. A variable v_i is *determined* for given domain D if the cardinality of the value set $|D(v_i)| = 1$. Otherwise it is *non-determined*.

An *integer valuation* θ is a mapping of variables to elements of their integer value sets, written $\{v_1 \mapsto d_1, \ldots, v_n \mapsto d_n\}$, where d_i's are elements of their value sets, I_j's. A solution of a *CSP* is a valuation that satisfies every constraint in the *CSP*. Following the notation introduced in Schulte and Smolka, we write $\theta \in D$ if $\theta(v_i) \in D(v_i)$ for all $v_i \in vars(\theta)$.

A constraint C_i over variables $v_1, \ldots, v_n$ is a set of valuations such that $vars(\theta) = \{v_1, \ldots, v_n\}$.

A domain D_1 is *stronger* than a domain D_2, written $D_1 \sqsubseteq D_2$, if $D_1(v_i) \subseteq D_2(v_i)$ for all $v_i \in V$. A *propagator* is a monotonically decreasing function from domains to domains: $f(D) \sqsubseteq D$ and $f(D_1) \sqsubseteq f(D_2)$ whenever $D_1 \sqsubseteq D_2$. A propagator f is *correct* for a constraint C_i iff for all domains D

$$\{\theta \in D\} \cap C_i = \{\theta \in f(D)\} \cap C_i$$

Constraint propagation is a process that changes the problem without changing its solutions by applying propagators. Constraint

propagation reduces the search space by reducing value sets of variables, strengthening constraints, or creating new ones.

2.1. The branch and bound algorithm

Based on the definitions above, the computation steps of the constraint propagation engine are as described below. The superscript k corresponds to the iteration step of the search algorithm.

1 *Initialization*: Create a *computation space* $S = \langle V, D^k, C\rangle$, $k = 0$.
2 *Branch*:
  (a) *Select a non-determined decision variable*: Select a decision variable $v_i \in V$, where $|D^k(v_i)| > 1$, using a selection function, *Sel*.
  (b) *Select a split point*: Select a subset $I_i^{k,1} \subset D^k(v_i)$ of the selected decision variable v_i using a *distribution strategy* U, $I_i^{k,1} = U(D^k(v_i))$. This will partition the value set of variable v_i into $I_i^{k,1}$ and $I_i^{k,2} = D^k(v_i) \setminus I_i^{k,1}$.
  (c) *Split computation space*: Create computation spaces $S_1 = \langle V, D^{k,1}, C\rangle$, $S_2 = \langle V, D^{k,2}, C\rangle$, where:
    $D^{k,1} = D^k$, except for v_i, where $D^{k,1}(v_i) = I_i^{k,1}$
    $D^{k,2} = D^k$, except for v_i, where $D^{k,2}(v_i) = I_i^{k,2}$
    $k \leftarrow k + 1$.
  (d) Push S_2 on the computation space stack; $S \leftarrow S_1$.
3 *Bound*: Apply propagators, f, to domain, D^k, to narrow the value sets of the decision variables in space S: $D^k \leftarrow f(D^k)$.
4 *Check the status of the computation space S*:
  (a) If all the decision variables are determined ($|D^k(v_i)| = 1$ for all v_i), report the variable assignments $\theta(v_i) \in D^k(v_i)$ for all v_i, discard the current computation space, S, and go to Step 5.
  (b) If there are possible solutions left that are consistent with the constraints in C ($|D^k(v_i)| > 1$) for at least one v_i, go to Step 2.
5 *Backtracking*: If the computation space stack is not empty, pop the computation space S from the stack and go to Step 2. Otherwise stop.

3. Formalization of the CSP search algorithm as a dynamical system

Since the concept of *dynamical system* is not widely used in the self-adapting computer systems community, we first introduce a general definition of this concept and then provide a dynamical system conceptualization of the branch and bound algorithm.

3.1. General dynamical systems

Intuitively, a dynamical system is a system whose past can be summarized by its *state*, which in turn changes, or evolves, from one moment of time to the next. The evolution is the function of both the current state and the *input* applied to the system. The *output* of a dynamical system depends on the system's state.

Formally, a discrete-time, time-invariant general dynamical system (Padulo and Arbib, 1974) *GDS* is an 8-tuple

$$GDS = (T, X, W, Q, P, f, g, \leq) \quad (1)$$

where T is the time set with an order relation $\leq$; X and W are the input and output sets, respectively; Q is the set of inner states $q \in Q$; P are the input processes (sequences of inputs, X), $p : T \to X$, $p \in P$; f is the (local) state transition function, $f : Q \times P \to Q$; g is the output function, $g : Q \to W$. The state transition function f of a dynamical system must satisfy the properties of consistency ($f(q_0, \emptyset) = q_0$) and causality ($f(q_0, p) = f(q_0, p')$, if $p = p'$). In words, the consistency condition means that the system does not change state within a given time moment; the causality means that whenever the system is started in a given initial state and two sequences are applied to it such that they are equal (for the same time set covered by the sequences), the system will end up in the same state. Fig. 1 represents the structure of a general, discrete time dynamical system. Note that this figure uses the Z domain, rather than the time domain representation (cf. Hellerstein et al., 2004).

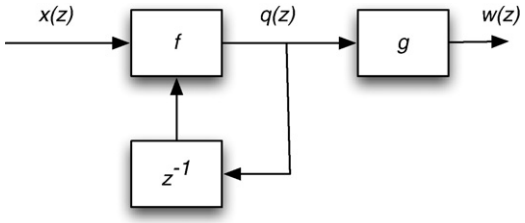

Fig. 1. The structure of a discrete time dynamical system (z^{-1} represents a unit delay).

3.2. Modeling the branch and bound algorithm as a dynamical system

To conceptualize a system as a dynamical system, one needs to identify all the elements shown in Eq. (1) above. Moreover, it is expected that the state transition function will satisfy the consistency and causality properties. To achieve this goal we had to analyze various options. In the end, we decided that the branch and bound search algorithm used in this paper can be formulated as a dynamical system in the following way:

- The time, T, is the set of non-negative integers representing the computation steps, k.
- The input space, X, is the set of subsets of the domain value sets, $I_i \subset D^k(v_i)$.
- The output set, W, is the set of non-negative integers representing the number of non-determined decision variables.
- The state space, Q, is the set of computations, S.
- The input processes, P, are the time sequences of values of the distribution strategy, U of the branch and bound algorithm.
- The state transition function $f : Q \times X \to Q$ is given by the branch and bound algorithm (see Section 2.1), with the initial state given by Step 1.
- The output function, g, is the function that returns the number that is equal to the number of variables that do not have a unique assignment of values in the current computation space, S, i.e., the number of variables $v_i \in V$, for which $|D^k(v_i)| > 1$.

4. The feedback control loop

The self-controlling software approach requires to identify measurable inputs to the plant and classify them as *control inputs*, which impact the plant's behavior. In addition, this approach assumes the existence of *disturbances* which alter the plant's behavior unpredictably. Moreover, it includes a *controller* subsystem for changing the values of the control inputs to the plant, and adds, if necessary, a *quality of service* (QoS) subsystem for computing feedback. In a *feedback loop*, at each (time) step the controller computes the control inputs for the plant based on the *goal* (e.g., a *set point*) received as an external input and feedback received from the QoS subsystem.

The first decisions that need to be made for the control of the branch and bound algorithm is what should be the control variable and what should be the measure that would give indications of whether the search algorithm is moving in the right direction. Intuitively, it is desirable that the search algorithm reaches the regions of the search space where feasible solutions exist, without traversing the entire search tree. Towards this aim, we need to identify a set of search parameters that impact the direction of search and a

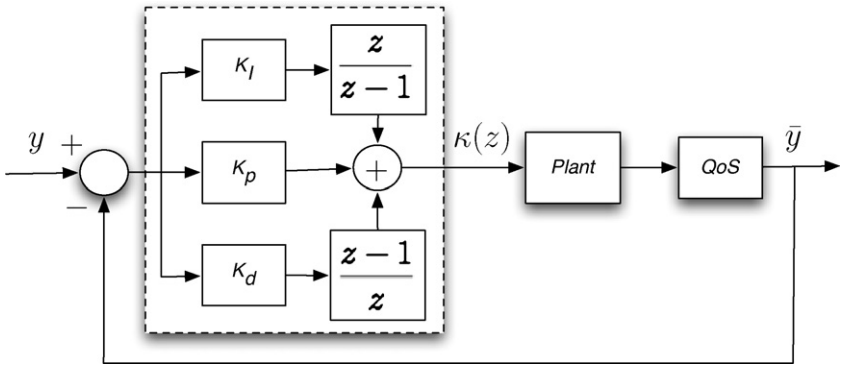

Fig. 2. Control diagram (dashed box represents the PID controller).

measure that can indicate whether the search is moving towards the regions with concentrated feasible solutions.

For the branch and bound algorithm described in Section 2.1, the selection function, *Sel*, that picks the next decision variable, v_i, to be distributed among the non-determined decision variables at the current decision node has the most impact on the direction of search. Two of the widely used selection functions in the branch and bound algorithm are (Haralick and Elliott, 1980): (1) *MaximumSuspensions*, the function that selects the non-determined decision variable that participates in most of the constraints and (2) *MinimumDomain*, the function that selects the non-determined decision variable with the smallest current domain, $D^k(v_i)$.

The second factor that impacts the search direction is the distribution strategy, $U(D^k(v_i))$, which splits the domain of the selected variable, v_i, into two subsets. The effect of U is local relative to the selection function *Sel*. However, U causes changes in the speed of search in addition to the direction. If U returns a singleton subset, then the search becomes a depth-first search and the vertical progress in the search tree gets faster. On the contrary, if U returns a multiple-element subset of $D^k(v_i)$, then the search becomes a breadth-first search and the vertical progress slows down.

In our case, all the domains are intervals of integers. The split thus can be defined by a real number $\kappa \in [0, 1]$. The distribution function then becomes:

$$U(D^k(v_i)) = I_i^{k,1} = \{x | x \leq (l + \kappa(u - l)) \quad (2)$$

where l and u are the lower and upper bounds of $I_i^{k,1}$, respectively. The value of $\kappa = 0$ results in $U(D^k(v_i))$ returning a singleton set containing the lower bound, while the value of 1 returning the upper bound of the interval.

Experimentation showed (Freuder, 1982) that *MaximumSuspensions* as the selection function and returning a single integer in the distribution function results in faster search. Therefore, in our experiments, we used the *MaximumSuspensions* function. In the case of a tie, the *MinimumDomain* function was used to break the tie. In summary, κ was used as the controlled variable for the branch and bound algorithm.

Next, a feedback variable is defined. Feedback is used to modify the control input in order to steer search towards the sections of the search tree where the percentage of feasible solution leaves is higher than the percentage of non-solution leaves. Consequently, we defined the feedback variable, $\bar{y}$, which is computed by the QoS function as shown in Eq. (3), where S is the number of feasible solution leaves and N is the number of non-solution leaves visited in the current control cycle. Feedback can take any value between 1.0 (all solutions found were feasible) and −1.0 (no feasible solutions found).

$$\bar{y} = \frac{S - N}{S + N} \quad (3)$$

When search enters a phase transition region, the value of $\bar{y}$ starts moving towards −1 (since the algorithm cannot find any feasible solutions). At this point the search has entered the region of high concentration of non-feasible solutions and thus the probability that there are feasible solutions that are easy to find is rather low. Thus this metric has an indirect effect on the probability of false alarms.

Fig. 2 demonstrates the Proportional-Integral-Derivative (PID) control used in our experiments. The figure uses the (typical) representation of the control action in the Z domain. Eq. (4) details the calculation of the control variable κ in the time domain. All the variables involved in the equation are indexed by the step, i.e., the index of $k-1$ refers to the value of the variable in the previous step, while k refers to the current value of a given variable. Here y is the control *goal*, $\Delta = y - \bar{y}$ is the control *error*, K_p and K_d are the proportional and derivative *control constants*, respectively (the integral constant was equal to 1):

$$\kappa(k) = K_p\Delta(k) + \kappa(k-1) + K_d(\Delta(k) - \Delta(k-1)) \quad (4)$$

This controller directs the search towards the areas where the value of S is higher and/or the value of N is lower. However, search should be terminated when the control does not seem to be efficient; this may be an indication that the search entered a phase transition region. The *termination point* is defined as the number of decision points visited since the last *significant event*. Significant event is defined as either the beginning of the search or the detection of a solution or the detection of a non-solution node. At every significant event point, the number of decision points reached since the last significant event is compared against the termination point. In case a solution is found, the count of decision points is reset. When the search reaches a termination point it returns a `Done` message and the search is terminated.

The termination point parameter is given to the CSP solver at the beginning of search. The number of decision points is computed in Step 2c in the branch and bound algorithm. The termination point condition is checked in Step 5 of the branch and bound algorithm.

The API of the selected Oz CSP solver had to be slightly modified to accept the control variable κ as a dynamic input and to provide access to vital search statistics. The Oz search statistics – the number of decision points, the number of feasible solutions and the number of non-solutions visited in the last control cycle, and the depth of the current decision point – had to be accessed in order to compute $\bar{y}$ (Eq. (3)) and to compute the termination point condition. This required the modification of the `Search.oz` file in the Mozart system library.

5. Verification and demonstration

An experimental system was implemented and tested on two scenarios to evaluate the approach specified in this paper; a job scheduling problem (see Section 5.1) and a fixture design utility problem (see Section 5.2). Problem formulations were specified using the UML/OCL representation. The formulations were automatically translated to the selected Constraint Satisfaction Problem (CSP) language, Oz, and then used by the system to find constraint satisfying solutions. The goal was to provide a proof-of-concept of the development of such a self-controlling constraint satisfaction solving program that (1) was applicable to various constraint satisfaction problems and (2) had the ability to control its own complexity by controlling the search direction and detecting phase transition regions to terminate the search in order to minimize the probability of false alarms. The first requirement was satisfied by demonstrating that the same system can be used in two different problem domains. The second requirement was satisfied by testing the experimental system on both hard and easy problem instances in order to show its ability to detect phase transition regions. In each of the scenarios, the phase transition phenomenon was investigated and an understanding of the model for phase transition was developed. These models were then used to generate test data for the experimental system. Known benchmarks implemented in Mozart Programming System version 1.3.1 were used as a reference.

5.1. The job scheduling example

This example deals with a job scheduling problem. There are a number of jobs, each consisting of one or more tasks. Jobs have due dates by which all the tasks for the job need to be completed. Tasks have prerequisites, i.e., sets of tasks that need to be completed before they can start. Each task requires a single resource and takes a certain time to be completed. There is only one of each resource type available and a resource can not be shared by more than one task at the same time. In our experiments, the ultimate goal was to find solutions that not only satisfy some constraints, but also are *good* with respect to two optimization goals. However, this aspect is outside of the scope of this paper and thus we focus only on the constraint satisfaction of this problem.

To formalize the constraints used in the job scheduling experiment, we introduce the following notation:

- Job – the set of all jobs
- $Task$ – the set of all tasks (the union of tasks for all the jobs)
- $Resource$ – the set of all resources
- $Times \subset Int$ – time interval
- $dur : Task \rightarrow Times$ – a function that returns the duration of a given task
- $preTask : Task \rightarrow 2^{Task}$ – a function that returns the set of prerequisite tasks for a given task
- $due : Job \rightarrow Times$ – a function that returns the due date or deadline for the completion of a given job
- $res : Task \rightarrow Resource$ – a function that maps a task to a resource (to simplify the problem, we assume that a task depends on a single resource)
- $start : Task \rightarrow Times$ – a function that returns the starting date for a given task (in our problem, starting dates for the tasks are the decision variables)
- $task : Job \rightarrow 2^{Task}$ – tasks that are part of a given job.

We use the *dot* notation to represent values of functions. For instance, we say $T.start$ to denote $start(T)$ for a given task T.

Constraint 1: All tasks that are prerequisites for task T should end before T can start:

$$\forall T \in Task \, \forall P \in T.preTask \cdot P.start + P.dur \leq T.start \tag{5}$$

Constraint 2: All jobs must finish by their deadlines:

$$\forall J \in Job \cdot \max_{T \in J.task} (T.start + T.dur) \leq J.due \tag{6}$$

Constraint 3: A resource cannot be used by more than one task concurrently:

$$\begin{aligned} &\forall T_1, T_2 \in Task \cdot T_1.start \leq T_2.start \,\& \\ &T_2.start \leq T_1.start + T_1.dur \Rightarrow T_1.res \neq T_2.res \end{aligned} \tag{7}$$

5.2. The fixture design example

The second experiment dealt with a scenario from the domain of testing of electronic circuitry boards in which units under test (UUT) are tested using a functional board tester (FBT). A UUT has a number of *pins* (e.g., 32). For testing, each pin needs to be connected to a *channel*, which is part of a *card* in the FBT. The FBT has a menu of cards, each with a set of *capabilities* associated with particular channels. To test a UUT, each pin's *requirements* need to be matched by a channel's capability. Examples of pin requirements are the digital timing (e.g., 10 MHz, 25 MHz, 50 MHz), the voltage levels that are used (e.g., [+5 V,−2 V], [+10 V,−10 V], [+15 V,−15 V], [+32 V,−32 V]) and the analog capabilities that are required (e.g., digital multi meter (DMM) input high/input low, function generator (FG) output, timer counter (TC) start/stop/trigger, digital storage oscilloscope (DSO) channel). The description of the configuration of the tester contains data on the number and the types of the channel cards and their positions within the chassis, as well as capabilities of their channels.

A testing facility is interested in keeping the overall cost of testing as low as possible. On the one hand, the goal is to minimize the number of channel cards in order to minimize the capital cost. On the other hand, the goal is to minimize the operational cost by using channels that are cheaper to use during testing (e.g., a 25 MHz channel is less expensive than a 50 MHz channel). The result of this optimization is a mapping between the UUT pins and the channels of the channel cards, e.g., UUT pin 1 will be connected to channel 2 of the third channel card, and so on. Again, in this paper we are focusing only on the constraint satisfaction aspect of this problem.

To formalize the constraints used in the fixture design experiment, we introduce the following notation:

- $Channel$ – the set of all channels on all of the channel cards
- Pin – the set of all pins
- $PinRequirement$ – the set of all pin requirements
- $Capability$ – the set of all channel capabilities
- $channel : Pin \rightarrow Channel$ – function that assigns channels to pins (wirings)
- $requirement : Pin \rightarrow 2^{PinRequirement}$ – function that assigns a set of requirements to a given pin
- $capability : Channel \rightarrow 2^{Capability}$ – function that assigns a set of capabilities to a given channel
- $card : Channel \rightarrow ChannelCard$ – function that assigns a channel card to a channel
- $mapping : PinRequirement \rightarrow Capability$ – function that assigns a channel capability to a pin requirement (the capability must satisfy the pin requirement)

Constraint 1: Each channel can be assigned to only one pin.

$$\forall p_1, p_2 \in Pin \cdot p_1.channel = p_2.channel \Rightarrow p_1 = p_2 \tag{8}$$

Constraint 2: Only the pins with requirements should be assigned to channels.

$$\forall p \in Pin \cdot P.requirement = \emptyset \Rightarrow P.channel = \emptyset \quad (9)$$

Constraint 3: Channel capabilities must cover pin requirements.

$$\forall p \in Pin \cdot p.requirement.mapping \subseteq p.channel.capability \quad (10)$$

6. Evaluations

6.1. Experiment planning and generation of test cases

The test cases were prepared in such a way as to cover the following types of cases: (1) where solutions exist and are easy to find (under-constrained); (2) where solutions do not exist and it is easy to find out about this (over-constrained); (3) phase transition regions where solutions exist; (4) phase transition regions where solutions do not exist. In total, 320 test cases were generated: 200 cases for the job scheduling problem and 120 for the fixture design problem.

The generation of test cases in the phase transition region was the result of both intuitions about the problem and extensive experimentation. Here we present the main ideas of this problem. The main intuition behind the phase transitions is that they are located where the problem is neither over nor under constrained, nevertheless it is "closer" to the over constrained region. Consequently, our goal was to find the cases where it is very difficult to satisfy all constrains of the problem.

For the job scheduling problem, we first looked for a *critical resource*, i.e., a resource that is needed for the largest number of tasks. The tasks that require this resource are then in the *critical task set*. One of the parameters that makes the constraint satisfaction difficult is the duration of all of the tasks in the critical task set, including the tasks that are the prerequisites to the tasks in the set. Tasks are part of jobs and jobs have deadline requirements. The second parameter that makes the satisfaction problem difficult is the due dates of the jobs, in particular of the job that has the earliest deadline. Thus, intuitively, the ratio of the duration to the due date, as described above, is the parameter that indicates where the phase transition region is. Larger values of duration make the satisfaction problem more difficult, but then larger values for deadlines can compensate for the difficulty. In particular, the value of the ratio equal to 1 is the boundary between satisfiable and unsatisfiable cases. If the ratio is greater than 1, the problem does not have a solution. So in summary, the hard cases are in the region where the ratio is slightly smaller than 1. Our experiments have confirmed this intuition, i.e., the test cases with the values of the ratio close to 1 were in the phase transition region. A test case generation program was implemented based upon these principles and the test cases generated by this program were used for testing.

A similar approach was employed to generate test cases for the fixture design problem. First, we identified a *critical requirement* parameter. It is the requirement that is associated with a *critical pin set* (i.e., the largest pin set that shares the same requirement). Since requirements need to be covered by the channel card capabilities, when the size of the critical pin set is larger than the number of channels on a single channel card that have this specific capability, this requirement is not satisfiable. This leads to the conclusion that the critical phase transition parameter is the ratio of the size of the critical pin set to the number of channels with a matching capability in a single channel card. When this ratio is greater than 1, there is no solution. Again, our experiments confirmed this intuition. A program to generate test cases based upon this analysis was written and the generated test cases were used for the evaluation of the approach presented in this paper.

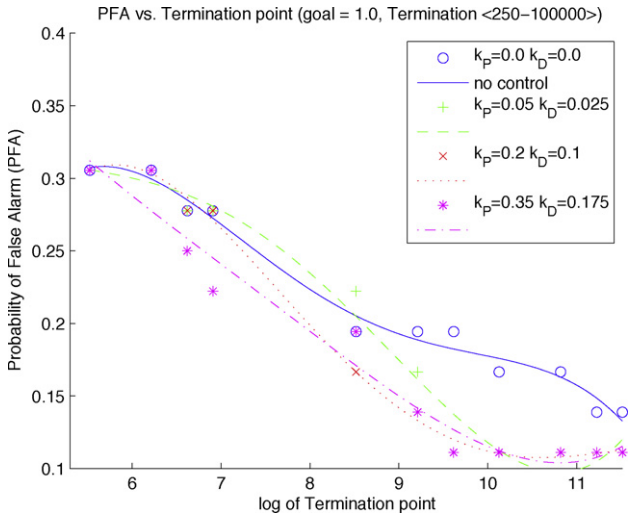

Fig. 3. Job scheduling experiment: PFA as a function of termination point and varying control parameters.

6.2. Evaluation metrics

The main goal of the controller for the branch and bound algorithm is to increase the efficiency of the algorithm in terms of both detecting solutions that satisfy the constraints of a given problem and recognizing phase transitions. As described earlier in the paper, the algorithm directs the search for solutions towards the areas of the search space where solutions are more likely to be found. It also has a termination point parameter that indicates a phase transition region and causes the search to quit. To assess the quality of the controller, we focused on the metric of *probability of false alarm* (*PFA*). *PFA* is defined as the ratio of *false positives* (*FP*), i.e., the number of cases where the algorithm wrongly declares a phase transition, to the number of all cases:

$$PFA = \frac{FP}{S+N} \quad (11)$$

where S is the number of cases for which solutions exist (or the space can be searched exhaustively) and N is the number of cases where a solution does not exist (or at least it is not known to exist).

6.3. Analysis of control: job scheduling problem

In the experiments with the controller two aspects were addressed: whether the control parameters could be modified in order to improve the performance of the controller and whether the controller performed well with respect to a benchmark CSP solver. Here we discuss just the first aspect in the context of the job scheduling problem.

To assess the sensitivity of the control algorithm to the change of control parameters we ran a number of experiments in which we modified the control parameters and observed the value of the PFA. In Fig. 3 we show the results of the experiments with different settings of the proportional and derivative parameters K_p and K_d. These plots allow us to make two observation. First, it is clear that the number of false alarms is much higher when control is not used. When search without control reaches the value of ten thousand iterations without finding a solution and quits, in about 20% of cases it is a false alarm. In a similar situation, when a search with control quits after the same number of iterations, the percentage of false alarms is almost null.

The second observation is that the controller is relatively insensitive to the values of the control parameters. Although the controller with the smaller values of $K_p = 0.05$ and $K_d = 0.025$

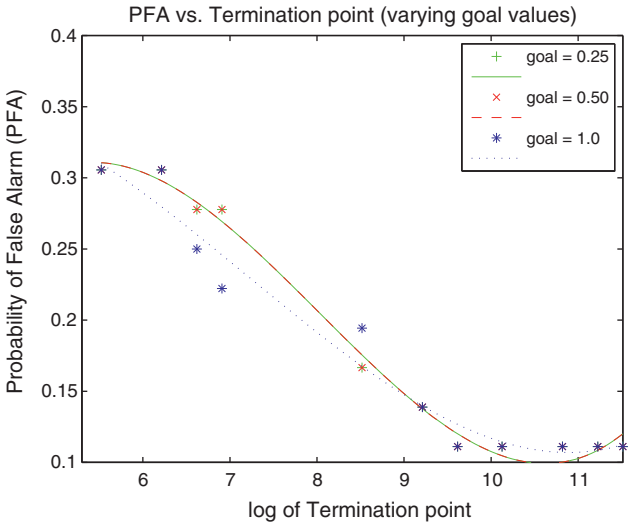

Fig. 4. Effects of changing goal values on PFA.

performed worse in terms of generated PFAs, the other two controllers with the higher values of K_p and K_d performed at about the same level of accuracy of generating false alarms.

Fig. 4 shows the impact of the selection of the value for the goal of the controller (y). As can be seen from this figure, this value did not have too much impact on the performance of the branch and bound search for this particular problem. The performance for the higher value of $y = 1$ was a bit better, especially for smaller values of the terminating point parameter, as compared to that of the smaller value of $y = 0.5$.

And finally, in Fig. 5 we show the impact of the selection of the initial value of the control variable, κ_0. These plots show a clear dominating performance of the control with the initial value of $\kappa_0 = 0$ over the selection of $\kappa_0 = 0.5$.

6.4. Analysis of control: fixture design problem

The results for the fixture design problem were similar to the results for the job scheduling problem. Again, as shown in Fig. 6, the performance of the search algorithm without control resulted in higher values of PFA. The variation in the values of K_p and K_d

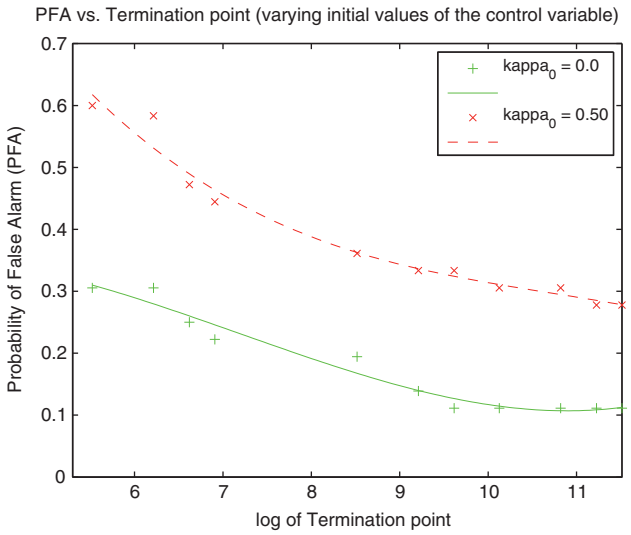

Fig. 5. Effects of changing the initial value κ_0 of the control variable on the values on PFA.

Table 1
Summary of types of test cases.

Problem	Total	S/S	N/N	S/PT	N/PT	PT/PT
Job scheduling	200	102	20	22	0	56
Fixture design	120	36	59	12	4	65

parameters did not have a significant effect. The selection of the initial value had a high impact on the performance of search, but we did not show a separate figure for this aspect.

6.5. Comparisons with a benchmark

To assess the performance of the controller in general, we used the known benchmark programs available in the Mozart Programming System, version 1.3.1. Both the benchmark programs and the code generated by our system were run on all of these cases. Since the two types of programs were developed using different approaches, their behaviors were different. The Oz benchmark programs were designed to search for at least one solution to the CSP problem indefinitely. In case it could not find a solution and exhausted its memory limits, it would generate a memory fault event and quit. Our program, on the other hand, was designed to detect that a given problem is in the phase transition region and then quit if the termination point condition (as defined in Section 4) was met. If it were able to find a solution before the termination point, it would provide the best solution found so far. Since the problem of detecting phase transitions is hard, our program would report a phase transition prematurely, when in fact it was not the case. Thus our program's behavior, at times, resulted in *false alarms*.

To compare the performance of the two systems we classified the results into the following five types:

S/S: A solution exists and both programs find at least one solution.
N/N: A solution does not exist and both programs recognize this fact.
S/PT: A solution exists, but our program reports a phase transition (false alarm).
N/PT: A solution does not exist; the benchmark runs until fault, our program recognizes a phase transition.
PT/PT: It is not known whether a solution exists (it is a hard case); the benchmark runs until fault, our program recognizes a phase transition.

The results of experiments are summarized in Table 1. This table shows the distribution of the test cases for each of the above described classes for the two constraint satisfaction problems. Column 2 shows the number of test cases for each of the two problems. Column 3 shows for how many cases both programs were able to find a solution. Column 4 shows in how many cases the two programs concluded that a solution did not exist. The *S/PT* column shows in how many cases the benchmark programs were able to find a solution, while our program generated a false alarm. Column *N/PT* indicates how many of the test cases did not have a solution. Our program was able to recognize this and thus terminated gracefully, while the benchmark programs ran until they generated faults. Column *PT/PT* shows the counts of the cases where the programs behaved in the same way as in the former case. Those were the *hard* cases where it is not known whether a solution exists or not.

To summarize, the main purpose of these tests was to show the superiority of our program over the benchmark programs in terms of the capability to quit the search when a hard case (a phase transition region) is encountered. This fact can be clearly seen from column *PT/PT*. However, we also had to show that our program performed well on the easy cases of an NP-complete problem, i.e., that

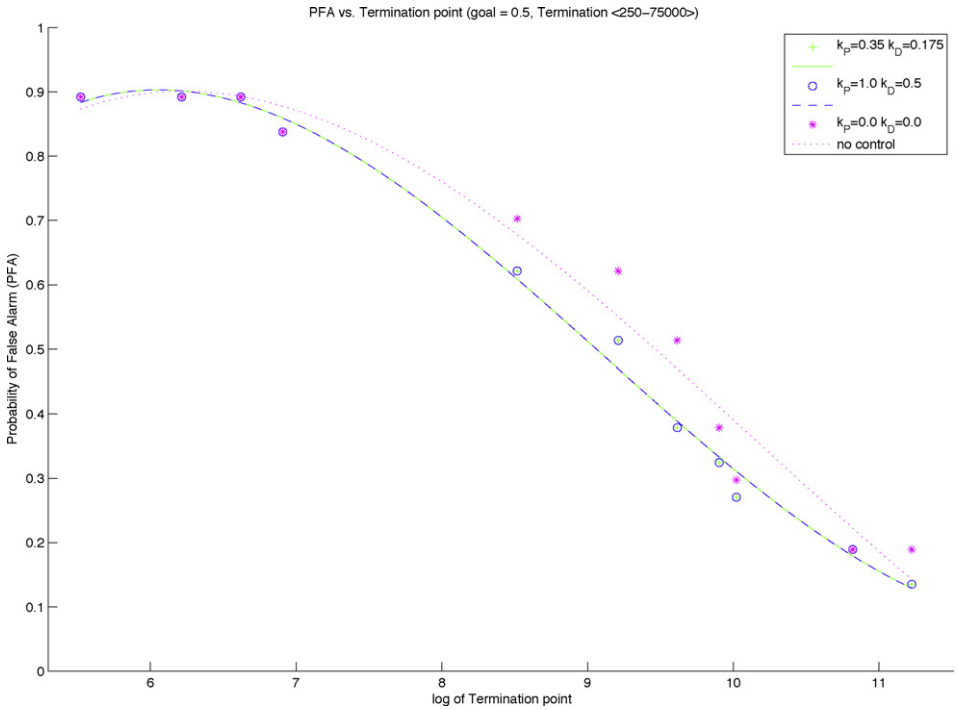

Fig. 6. Fixture design experiment: PFA as a function of termination point and varying control parameters.

it was able to find solutions in under-constrained cases. Its behavior was inferior for some of the harder cases (column *S/PT*). However, this was compensated by a significantly better performance of our program for the hard cases (columns *N/PT* and *PT/PT*) for which the benchmark programs simply ran until generating faults.

7. Literature review

Literature related to self-* systems was briefly overviewed in Section 1. Here we focus our attention on the CSP solving algorithms. In our research, first, we had to answer the question whether such algorithms do need improvement. Second, we had to find out whether self-adaptive software methods in general, and control theory in particular, have been used to control CSP solvers. The answer to the former question was affirmative and to the second negative. Below we summarize some of our findings.

The classical CSP framework was introduced formally at the beginning of the 70s (Montanari, 1974). In Montanari and Rossi (1996), Rossi and Montanari give a brief history of the studies on CSP. The general constraint satisfaction problem is of high-complexity (*NP*-complete or worse) and there are no efficient algorithms to solve it. Therefore, one of the main research topics has been finding fast preprocessing algorithms that can make the search for a solution efficient in important practical cases.

A widely used method for solving the CSP is *backtracking* (Korf, 2010). In this method, variables are instantiated sequentially. As soon as all the variables relevant to a constraint are instantiated, the validity of the constraint is checked. If a partial instantiation violates any of the constraints, backtracking is performed to the most recently instantiated variable that still has alternatives available. However, the backtracking method suffers from thrashing (Gaschnig, 1979); i.e., search in different parts of the space keeps failing for the same reasons. The cause of thrashing is referred as to as lack of *arc consistency* (cf. Kumar, 1992). Algorithms for eliminating arc-inconsistency are only a subset of *preprocessing algorithms* that are used to increase the efficiency of the search algorithms. Some of such preprocessing algorithms are described in Fikes (1970), Mackworth (1977), Montanari (1974), Freuder (1978), Dechter (1990), Freuder (1988), and Waltz (1975). Other significant algorithms that improve on the classical backtracking algorithm are *intelligent backtracking* (Gueret et al., 2000), *dynamic backtracking* (Ginsberg and McAllester, 1994), and *backjumping* (Dechter, 1990). None of these algorithms use a control feedback loop built on the principles of control theory.

Improvements to the search algorithms have been investigated using the standard depth-first and breadth-first search as the base. Breadth-first search may require too much space and depth-first search may spend too much time without finding any feasible solutions. Iterative-deepening depth-first (IDDF) search (Korf, 1985) runs a depth-limited search repeatedly, increasing the depth limit at each iteration until it reaches the shallowest goal state. IDDF search combines depth-first search space efficiency and breadth-first search completeness. Iterative-Deepening A* (IDA*) (Korf, 1985) combines IDDF with A* (Dechter and Pearl, 1985), which uses a best-first search and finds the least-cost path from a given initial node to one goal node. Later in Korf (1997), Korf modified IDA* with a lower-bound heuristic function based on memory-based lookup tables, called "pattern databases" (Culberson and Schaeffer, 1996).

Combinatorial CSPs are a subcategory of the general CSPs where the values of some or all of the decision variables are restricted to be integer. The word "combinatorial" is used to stress the fact that only a finite number of alternative feasible solutions exist. The branch and bound algorithm is the most widely used solver for combinatorial CSPs. Mitten (1970) and de Bruin et al. (1988) characterize a branch and bound algorithm by four rules: (1) a *branching rule* defining how to split a problem to sub-problems, (2) a *bounding rule* defining how to compute a lower bound on the optimal solution of a sub-problem, (3) a *selection rule* defining which sub-problem to branch from next, and (4) an *elimination rule* stating how to recognize and eliminate sub-problems which cannot yield an optimal solution to the original problem.

Significant research has been done on efficient branching rules that will improve the performance of a branch and bound algorithm. Branching heuristics that choose the most "constrained" decision variable to get away from the over-constrained regions

had been proved to be very effective in search (Haralick and Elliott, 1980; Beck et al., 2004; Freuder, 1982; Gent et al., 1999). Our paper focuses on the branch and bound algorithm and many of the ideas known in the subject literature have been useful to us in selecting control variables and developing the controller.

Theoretical evaluation of constraint satisfaction algorithms is accomplished primarily by worst-case analysis or by dominance relationships (Kondrak and van Beek, 1997). However, for NP-complete problems, a different approach is needed. In the past, intractable problems were avoided. Recently it has been recognized that for NP-complete problems solutions can be found, or the lack of a solution can be efficiently recognized, except for some input regions called *phase transition regions*.

The idea of *phase transitions* was first introduced in statistical physics. In Cheeseman et al. (1991), Cheeseman showed that for many NP-complete problems, one or more *order parameters* can be found such that *hard instances* (i.e., such that solutions are difficult to find) occur around particular critical values of these order parameters. These critical values are called *phase transition invariants*. Such values define a boundary that separates the space of problems into two regions or *phases*. While one region is under-constrained and easy to find a solution, the other region is over-constrained and unlikely to contain a solution. Really hard problems occur on the boundary between these two regions where the probability of finding a solution is low but not negligible.

Currently, a normal practice is to evaluate a proposed algorithm empirically on a set of randomly generated instances taken from the relatively hard phase transition region (Selman et al., 1992). As Cheeseman indicated, there is a need to produce phase transition diagrams for particular problem domains to help in identifying hard problems and predicting the existence of solution, such as shown in Purdom (1983). Phase transitions in constraint satisfaction problems have been studied by many researchers (cf. Prosser, 1996; Smith and Dyer, 1996; Gent et al., 1997; Selman and Kirkpatrick, 1996; Hogg and Williams, 1994; Hogg, 1996; Hogg et al., 1996; Xu and Li, 2000). At this point, the challenge is to identify order parameters and phase transition invariants for particular problem domains. In this paper we describe the results of our investigation to identify phase transition regions for the two selected domains of CSP.

8. Conclusions and future research

In conclusion, the main objective of this work was to show that the generic ideas from control theory can be used for controlling the performance of algorithms. To show this, two NP-hard constraint satisfaction problems have been selected to which a generic PID controller was used to manipulate a control variable (a parameter in the algorithm). Through a number of experiments, it was shown that adding a controller to a CSP solver had a clearly positive effect in terms of the solver's performance as measured by its capability of detecting phase transition regions, i.e., regions where solutions that satisfy the constraints don't exist and thus pursuing the search in these regions was not justified. Experiments with the two selected problems were performed in order to gain some understanding of the sensitivity of the performance of the controller to the selection of the control parameters. The experiments showed that the controller is highly sensitive to the selection of the initial value of the control variable, while it was much less sensitive to the tuning of the control parameters.

The two problems come from two different application domains (job scheduling vs. fixture design) and thus it can be expected that the approach proposed in this paper may be applicable to a wide range of domains where search for solutions that satisfy constraints is needed.

The approach was represented in generic terms, i.e., first the problem was conceptualized as a dynamical system, then control and performance variables were identified, a QoS function was defined, control goal (set point) was defined, and a control law (in this case PID) was defined. Although all these decisions required some significant analysis of the problem domain (CSP and branch and bound algorithms), the process used in this work is based on system theory and can be replicated for other applications, as well.

While this paper showed that an algorithm can be treated as a dynamical system and thus can be controlled using traditional controllers (like PID), more research is needed to develop a generic methodology for applying control theory to control algorithms. In particular the following questions need to be investigated. Do we always need to analyze the dynamics of a given algorithm before adding a self-control feedback loop? What is the procedure for identifying state, input and output parameters for algorithms? What is the procedure for selecting controlled variables and performance variables? How should the analysis of dynamics be integrated into the architecture based self-adaptation methodology?

Acknowledgments

We would like to thank the anonymous reviewers for their insightful suggestions on our paper. We sincerely believe that the incorporation of these suggestions made our paper much easier to read and better aligned with the directions and goals of the architecture based self-adapting computer systems.

References

Åström, K.J., 1989. Adaptive Control. Addison-Wesley, Reading, MA.
Badr, N., Reilly, D., Taleb-Bendiab, A., 2002. A conflict resolution control architecture for self-adaptive software. In: ICSE 2002 Workshop on Architecting Dependable Systems, Orlando, Florida.
Beck, J., Prosser, P., Wallace, R., 2004. Variable ordering heuristics show promise. In: Principles and Practice of Constraint Programming – CP2004, 10th International Conference, LNCS, Springer, Toronto, pp. 711–715.
Blueprint, A.A., 2006. An architectural blueprint for autonomic computing. White paper, fourth edition. IBM Corporation, 2006.
Brandozzi, M., Perry, D.E., 2002. Architectural prescriptions for dependable systems. In: ICSE 2002 Workshop on Architecting Dependable Systems.
Brun, Y., Di, G., Serugendo, M., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Pezz, M., Shaw, M., 2009. Engineering self-adaptive systems through feedback loops. In: Software Engineering for Self-Adaptive Systems, pp. 48–70.
Cheeseman, P., Kanefsky, B., Taylor, W., 1991. Where the really hard problems are. In: Proc. IJCAI91, vol. 1, pp. 331–337.
Culberson, J., Schaeffer, J.,1996. Searching with pattern databases. In: Advances in Artificial Intelligence (Lecture Notes in Artificial Intelligence 1081). Springer-Verlag, pp. 402–416.
de Bruin, A., Rinnooy Kan, A., Trienekens, H., 1988. A simulation tool for the performance evaluation of parallel branch and bound algorithms. Mathematical Programming 42, 245–271.
Dechter, R., Pearl, J., 1985. Generalized best-first search strategies and the optimality of a*. Journal of the ACM 32 (3), 505–536.
Dechter, R., 1990. On the expressiveness of networks with hidden variables. In: Proceedings of the Eight National Conference on Artificial Intelligence (AAAI-90), Boston, MA, pp. 556–562.
Dechter, R., 1990. Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition. Artificial Intelligence 41 (3), 273–312.
Dechter, R., 1992. From local to global consistency. Artificial Intelligence 55, 87–107.
Doyle, J., Francis, B., Tannenbaum, A., 1990. Feedback Control Theory. Macmillan Publishing Co.
Fescioglu-Unver, N., Kokar, M.M., 2005. Effects of computation speed on the stability of a self-controlling process. In: Proceedings of the 12th IEEE International Conference and Workshops on the Engineering of Computer-Based Systems, pp. 398–405.
Fescioglu-Unver, N., Kokar, M., 2008. Self controlling tabu search algorithm for vehicle routing problem with time windows. In: 13th International Conference on Machine Design and Production, pp. 829–843.
Fescioglu-Unver, N., Kokar, M.M., 2008. Application of self controlling software approach to reactive tabu search. In: Proceedings of the Second IEEE International Conference on Self-Adaptive and Self-Organizing Systems.
Fescioglu-Unver, N., Kokar, M.M., 2011. Self controlling tabu search algorithm for the quadratic assignment problem. Computers & Industrial Engineering 60, 310–319.

Fikes, R., 1970. REF-ARF: a system for solving problems stated as procedures. Artificial Intelligence 1, 27–120.
Freuder, E.C., 1978. Synthesizing constraint expressions. Communications of ACM 21 (11), 958–966.
Freuder, E.C., 1988. Backtrack-free and backtrack-bounded search. In: Kanal, L., Kumar, V. (Eds.), Search in Artificial Intelligence. Springer-Verlag, London, UK, pp. 343–369, http://portal.acm.org/citation.cfm?id=60727.60737.
Freuder, E.C., 1982. A sufficient condition for backtrack-free search. Journal of the ACM 29 (1), 24–32.
Garlan, D., Schmerl, B., Chang, J., 2001. Using gauges for architecture-based monitoring and adaptation. In: The Working Conference on Complex and Dynamic Systems Architecture, Brisbane, Australia.
Gaschnig, J., 1979. Performance measurement and analysis of certain search algorithms. Ph.D. thesis. Dept. of Computer Science, Carnegie Mellone University.
Gent, I.P., MacIntyre, E., Prosser, P., Shaw, P., Walsh, T.,1997. The constrainedness of arc consistency. In: Proceedings of CP-97. Springer, pp. 327–340.
Gent, I.P., Prosser, P., Walsh, T., 1999. The constrainedness of search. In: Proceedings of AAAI-96, pp. 246–252.
Ginsberg, M.L., McAllester, D.A., 1994. Gsat and dynamic backtracking. Journal of Artificial Intelligence Research 1, 25–46.
Goldman, R.P., Musliner, D.J., Krebsbach, K.D., Boddy, M.S.,1997. Dynamic Abstraction Planning. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97). AAAI Press /MIT Press, Providence, Rhode Island, pp. 680–686, http://citeseer.nj.nec.com/goldman97dynamic.html.
Gueret, C., Jussien, N., Prins, C., 2000. Using intelligent backtracking to improve branch-and-bound methods: an application to open-shop problems. European Journal of Operational Research 127 (2), 344–354.
Haralick, R., Elliott, G., 1980. Increasing tree search efficiency for constraint satisfaction problems. Artificial Intelligence 14 (3), 263–313.
Hellerstein, J.L., Diao, Y., Parekh, S., Tilbury, D., 2004. Feedback Control of Computing Systems. John Wiley and Sons.
Hogg, T., Williams, C.P., 1994. The hardest constraint problems: a double phase transition. Artificial Intelligence 69, 359–377.
Hogg, T., Huberman, B.A., Williams, C., 1996. Frontiers in problem solving: phase transitions and complexity, introduction. Special issue of Artificial Intelligence 81 (1–2), 1–15.
Hogg, T., 1996. Quantum computing and phase transitions in combinatorial search. Journal of Artificial Research 4, 91–128.
Karsai, G., Sztipanovitz, J., 1999. A model-based approach to self-adaptive software. IEEE Intelligent Systems and Their Applications 14 (3), 46–53.
Klein, M., Kazman, R., Attribute-Based Architectural Styles (CMU/SEI-99-TR-022). Software Engineering Institute, Carnegie Mellon University, 1999. http://www.sei.cmu.edu/library/abstracts/reports/99tr022.cfm.
Kokar, M., Baclawski, K., Eracar, Y., 1999. Control theory-based foundations of self-controlling software. IEEE Intelligent Systems and Their Applications 14 (3), 37–45.
Kokar, M., Passino, K., Baclawski, K., Smith, J.,2001. Mapping an application to a control architecture: specification of the problem. In: IWSAS, Vol. 1936 of Lecture Notes in Computer Science. Springer, pp. 75–89.
Kondrak, G., van Beek, P., 1997. A theoretical valuation of selected algorithms. Artificial Intelligence 89, 365–387.
Korf, R., 1985. Depth-first iterative-deepening: an optimal admissible tree search. Artificial Intelligence 27, 97–109.
Korf, R., 1997. Finding optimal solutions to Rubik's cube using pattern databases. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97), Providence, RI, pp. 700–705.
Korf, R.E., 2010. Artificial intelligence search algorithms. In: Atallah, M.J., Blanton, M. (Eds.), Algorithms and Theory of Computation Handbook. Chapman & Hall/CRC, pp. 22–22, http://portal.acm.org/citation.cfm?id=1882723.1882745.
Kumar, V., 1992. Algorithms for constraint satisfaction problems: a survey. AI Magazine 13 (11), 32–44.
Laddaga, R., 1999. Creating robust software through self-adaptation. IEEE Intelligent Systems and Their Applications 14 (3), 26–29.
Mackworth, A.K., 1977. Consistency in networks of relations. Artificial Intelligence 8 (1), 99–118.
Mehl, M., Muller, M., Popov, T., Scheidhauer, K., 1995. DFKI Oz User's Manual. German Research Center for Artificial Intelligence.
Mitten, L., 1970. Branch-an d-bound methods: general formulation and properties. Operations Research 18 (1), 24–34.
Montanari, U., Rossi, F., 1996. Constraint solving and programming: what's next? ACM Computing Surveys 28 (4es), Article no. 70, http://dx.doi.org/10.1145/242224.242314.
Montanari, U., 1974. Networks of constraints: fundamental properties and application to picture processing. Information Sciences 7, 95–132.
Musliner, D.J., 2000. Imposing realtime constraints on self-adaptive controller synthesis. In: Robertson, P., Shrobe, H., Laddaga, R. (Eds.), Self-Adaptive Software, First International Workshop, IWSAS 2000, Vol. 1936 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, Berlin, Heidelberg, Oxford, UK, pp. 143–160.
Oreizy, P., Gorlick, M.M., Taylor, R.N., Heimbigner, D., Johnson, G., Medvidovic, N., Quilici, A., Rosenblum, D.S., Wolf, E.L., 1999. An architecture-based approach to selfadaptive software. IEEE Intelligent Systems 14 (3), 54–62.
Padulo, L., Arbib, M., 1974. System Theory: A Unified State-space Approach to Continuous and Discrete Systems. W.B. Saunders Company, Philadelphia, PA.
Pezze, M., Muller, H., Shaw, M., 2008. Visibility of control in adaptive systems. In: Second International Workshop on Ultra-Large-Scale Software-Intensive Systems (ULSSIS 2008), at the International Conference on Software Engineering (ICSE 2008), pp. 23–26.
Prosser, P., 1996. An empirical study of phase transitions in binary constraint satisfaction problems. Artificial Intelligence 81, 81–109.
Purdom, P., 1983. Search rearrangement backtracking and polynomial average time. Artificial Intelligence 21 (1–2), 11–134.
Reece, S., 2000. Self-adaptive multi-sensor systems. In: Robertson, P., Shrobe, H., Laddaga, R. (Eds.), Self-Adaptive Software, First International Workshop, IWSAS 2000, Vol. 1936 of Lecture Notes in Computer Science (LNCS). Springer-Verlag, Berlin, Heidelberg, Oxford, UK, pp. 224–241.
Robertson, P., Laddaga, R., 2004. The grava self-adaptive architecture: history; design; applications; and challenges. In: Proceedings of the 24th International Conference on Distributed Computing Systems Workshops, vol. 7, pp. 298–303.
Schulte, C., Smolka, G., Finite Domain Constraint Programmin in Oz: A Tutorial. Available on line at: http://www.mozart-oz.org/documentation/fdt/.
Schulte, C., 2002. Programming constraint services. In: Vol. 2302 of Lecture Notes in Artificial Intelligence. Springer-Verlag, Berlin, Germany, http://link.springer.de/link/service/series/0558/tocs/t2302.htm.
Selman, B., Kirkpatrick, S., 1996. Critical behavior in the computational cost of satisfiability testing. Artificial Testing 81 (1–2), 273–295.
Selman, B., Levesque, H., Mitchell, D., 1992. A new method for solving hard satisfiability problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence, pp. 440–446.
Shamma, J.S., 1996. Linearization and gain-scheduling. In: The Control Handbook. CRC Press.
Shaw, M., 1995. Beyond objects: a software design paradigm based on process control. ACM Software Engineering Notes 20, 20–21.
Smith, B., Dyer, M., 1996. Locating the phase transition in binary constraint satisfaction problems. Artificial Intelligence 81, 155–181.
Sztipanovits, J., Karsai, G., 1997. Model-Integrated Computing, IEEE Computer, 110–112.
Van Roy, P., Haridi, S., 2004. Concepts, Techniques, and Models of Computer Programming. MIT Press.
Waltz, D., 1975. Understanding line drawings of scenes with shadows. In: Winston, P.H. (Ed.), The Psychology of Computer Vision. McGraw Hill, Cambridge, MA, pp. 19–91.
wen Cheng, S., Garlan, D., Schmerl, B., Sousa, J.P., Spitznagel, B., Steenkiste, P., Hu, N., 2002. Software architecture-based adaptation for pervasive systems. In: Proceedings of the International Conference on Architecture of Computing Systems: Trends in Network and Pervasive Computing, pp. 67–82.
Xu, K., Li, W., 2000. Exact phase transitions in random constraint satisfaction problems. Journal of Artificial Intelligence Research 12, 93–103.
Xun, Y., Kokar, M.M., Baclawski, K., 2004. Control based sensor management for a multiple radar monitoring scenario. Information Fusion: An International Journal on Multi-Sensor, Multi-Source Information Fusion 5, 49–63, 1.

Yönet A. Eracar joined Teradyne in 1996 as a software engineer working on Teradyne's digital, analog and high speed subsystem test instruments. He has held various technical lead and project management positions. Currently, Yönet is managing the software department in Teradyne's Defense and Aerospace Business Unit. Yönet received BS degrees in electrical engineering and physics from Bosphorus University, an MS degree in computer systems engineering, and a doctorate in industrial engineering from Northeastern University.

Mieczyslaw M. Kokar is Professor of Electrical and Computer Engineering at Northeastern University in Boston, USA. He holds an MS (1969) and a PhD (1973) degree in computer and system engineering, both from the Wroclaw University of Technology. His technical research interests include information fusion, ontology-based information processing, cognitive radio and self-controlling software. Professor Kokar teaches various graduate courses in software engineering, formal methods and artificial intelligence.