Self-* Programming
Run-Time Parallel Control Search for Reflection Box*

Olga Brukman and Shlomi Dolev

Department of Computer Science,
Ben-Gurion University of the Negev,
Beer-Sheva, 84105, Israel
{brukman, dolev}@cs.bgu.ac.il

Abstract. Real life situations may require an automatic fast update of the control of a plant, whether the plant is an airplane that needs to overcome an emergency situation due to drastic environment change, or a process that needs to continue executing an application in spite of a change in an operating system behavior. Settings for run-time control synthesis are defined, assuming the environment maybe totally dynamic, but is *reentrant* and *history oblivious* for long enough periods. A *reentrant* environment allows several copies of a plant to interact with the environment independently; a *history oblivious* environment ensures a repetition (in the probabilistic case with the same probability) of an interaction starting with a plant in a certain state and replaying its output to the environment.
Total dynamic changes of the environment do not allow a definition of *weakly realizable* specifications, as weakly realizable specifications depend on the environment behavior. On line experiments of the environment assists in the implementation of *unrealizable* specifications; Automatically checking whether the *unrealizable* specifications define *weakly realizable* specifications, given the behavior restriction of the current environment. A successful search for a control implicitly identifies the *weakly realizable* specifications, and explicitly the implementation that respect the specifications.
Different settings and capabilities of the plant are investigated. In particular, (*i*) *plant state reflection* that allows observation of the current state of the plant, (*ii*) *plant state set* that generalizes the reset capability, allowing setting the plant to each of its states, and (*iii*) (static or dynamic) *plant replication* that allows instantiation of plant replicas or use of preexisting plant replicas for parallelizing testing algorithms. The algorithms presented prove that the above capabilities enable a polynomial search for a new control upon a drastic change of the environment.
Keywords: replication, state set, reflection.

1 Introduction

In the early attempts to reach supersonic speeds, flight pilots experienced a strange phenomenon that made their control surfaces useless, and their aircraft uncontrollable. The airplanes were saved by either reducing the speed or changing the usual control procedure [18]. Flying an airplane in a volcano ash cloud may stop the operation of the airplane. The airplane can still be saved if the pilots direct it out of the ash cloud, let the engines cool and then restart them [8]. These two examples demonstrate the type of dramatic control changes that sometimes has to be made on-line, without prior experience, when the environment changes unexpectedly.

Today, when a programmer creates a program, she/he designs the program for a certain environment. When the program encounters unanticipated environmental behavior, the program performance may degrade drastically, it may continue to execute while producing a faulty (unexpected) output, or it may crash. Programmers and system administrators use their accumulated knowledge of the system and of the environment to investigate and solve problems by patching up the system each time a new problem is detected. In many

* Partially supported by the Lynne and William Frankel Center for Computer Sciences, by a Deutsche Telecom grant, the Israeli Ministry of Science, and the Rita Altura Trust Chair in Computer Sciences. The paper abstract appeared in The Sixth NASA Langley Formal Methods Workshop (LFM 2008), NASA Conference Proceedings, April 2008.

cases, the solution is post mortem and off-line. Ideally, systems would be autonomous, i.e., the systems would be able to cope with unexpected situations dynamically and independently, without human intervention.

In the example of the plane in the volcano ash cloud, imagine that the plane is able to release miniature replicas of itself into the air, where each replica tries a different control program. The replicas that succeed in leaving the ash cloud successfully report back to the plane. Next, the plane uses the obtained control to leave the ash cloud. This is an example of an autonomic system that is able to deal with unexpected changes in the environment.

Growing interest in disaster recovery [9], RAS (Reliability, Availability, Serviceability) paradigm, autonomic computing [11], self-healing systems, and evolving systems, reflects the desire and the need for systems that automatically adapt themselves to an unpredictably changing environment.

Our contribution. We suggest a program search engine that uses parallelization to produce a supervisory control dynamically and automatically. The program search engine produces a control on the fly, such that the control continuously respects a set of (possibly unrealizable) specifications in the presence of dynamic changes in an environment. We do not make any assumptions about changes the environment may undergo during the program life cycle.

A natural environment is very big, sophisticated and dynamic. The environment can only be modeled by a non-deterministic infinite automaton. On-line learning of the environment automaton is impossible. Thus, we make a distinction between a plant – a machinery our program interacts with and the environment (the rest of the universe). We assume that a plant can be modeled by a deterministic or probabilistic finite automaton. Even if we are convinced, as Albert Einstein was, that "God does not throw a dice", i.e., the environment is deterministic, still the environment state cannot be contained by finite means and the environment automaton cannot be learned; in such a case, one may model the environment as an automaton with probabilistic transition function. We use testing techniques to obtain a control for the plant dynamically in an efficient manner. Our settings differ from the common approach where the whole environment is considered monolithically [19].

The plant is considered to be either an *rs-box* (reflection and set box) or a *black box*. The *rs-box* plant has a *state reflection* capability or a *state set* capability, or both. The *state reflection* capability provides access to the plant state (e.g., using Java reflection) without revealing the plant automaton transition function. The *state set* capability, which is a generalization of the *reset* capability, allows setting the plant state to a given state. We are able to record only the plant inputs and outputs for the *black box* plant.

We suggest a control search engine that finds a supervisory control that respects the specifications dynamically and automatically by experimenting on plant replicas. In order to detect the deterioration of a current control due to a change in the environment, the control search engine constantly monitors the control execution by obtaining a reliable record of the control-plant-environment interaction from a dependable entity called an *Observer*. The search engine evaluates the quality of the interaction and initiates a search for a new control if the current control does not respect the specifications.

We present control search algorithms for various plant settings: (*i*) *plant state reflection* which allows a control search algorithm to learn a connected component of a current plant state in the plant automaton graph (*ii*) *plant state set* that generalizes the reset capability, allows setting the plant to each of its states and exploring all connected components of the plant automaton graph, (*iii*) *static plant replication* that implies use of preexisting replicas for parallelizing testing algorithms or *dynamic plant replication* capability that allows instantiation of new replicas in run time, and (*iv*) deterministic or probabilistic plant automaton.

We show that parallelization and the ability to observe and to manipulate the plant state allow us to improve the control search complexity. The use of parallelization makes the search time reasonable for on line systems, trading off (possibly exponential or polynomial) time with a (exponential or polynomial) number of plant replicas. The plant state capabilities allow reduction of the number of experiments on the plant replicas from exponential (for a *black box* plant) to polynomial (for *rs*-box plant) in the number of plant automaton states.

1.1 Related Work

Statically generated adaptable programs. Several papers explore techniques for creating systems that are able to adapt themselves to a changing environment by using information about the environment that was accumulated off-line, e.g., planning, multi-version programming. Such a program is able to adapt itself only to a predefined set of changes in the environment, namely, all environment changes have to be anticipated off-line, as the program is being created. This implies that the program still may exhibit an undesired behavior if it encounters unexpected changes in the environment.

The SA-CIRCA (Self-Adaptive Cooperative Intelligent Real-Time Control Architecture) [13] is an example of a program search engine that generates a program for a robot from full environment specifications and overall system goals. The program search engine dynamically generates patches of code for handling unexpected scenarios by executing a search algorithm with potentially unbounded time complexity. SA-CIRCA uses incremental improvement algorithms that trades off solution quality for shorter computation time.

A model-based approach for self-adaptive software [21] suggests that a programmer has to specify redundant methods for completing each method. If faults in a system indicate that a method does not achieve its goals, the management module deprecates the current method and chooses a substituting method from the list. The choice of the new method is based on reasoning about the consequences of the deprecated method's actions. Additional criteria for choosing the new method are safety, timeliness and accuracy criteria. The authors claim that as the self-adaptive software evolves, by deprecating the current methods and choosing better methods in the system context, the software becomes more robust and with better performance. This conclusion is strongly based on the fact that the programmer will supply efficient redundant methods that suit every state of the environment.

A system presented in [23] is able to alter its behavior by controlling its sub-components. The system acts as a controller which monitors its sub-components and reconfigures the composition of the sub-components choosing from a pool of predefined configurations in order to achieve the desired functionality and the system goals.

In our work we do not make any assumptions about the types of faults that may happen and we do not provide predefined alternative scenarios for anticipated faults. Our program search engine is able to create a completely new program upon (drastic) changes in a system environment.

Autonomic computing. The research in autonomic systems suggests accepting failures as a part of a system's existence. Instead of trying to anticipate all possible failures and problems, autonomic computing suggests enhancing systems with a mechanism for automatic recovery from failures. The decision on which recovery action to take place is made in run time, i.e., an autonomic system is a dynamically adaptable software. However, the recovery actions are predefined; therefore, an autonomic system is able to recover efficiently from only a restricted set of failures.

The common approach to autonomic computing [15] is an architecture consisting of a core component, a monitoring layer, an inference tool, and an affecting tool. The core component mostly executes correctly, but may fail occasionally. The monitoring layer constantly traces the component execution and streams its output as an input for the inference tool. The inference tool analyzes the component behavior and detects the faults and their source based on artificial intelligence techniques. Then, the inference tool suggests a recovery procedure based on some learning algorithm output or based on some predefined recovery scenarios. The recovery procedure is invoked through the affecting tool – a tool that is able to alternate a behavior of the core component.

The common autonomic computing architecture is unable to deal with drastic change in the environment: in this case the program search engine may repeatedly detect failures and initiate recovery actions, while not achieving any significant progress in terms of system goals.

Program synthesis from realizable specifications. This research area suggests compiling a program from requirements and environment specifications (environment variables and environment actions). The generated program respects the specifications with regard to the current environment, i.e., the program is absolutely correct with regard to the current environment. In [22] the specifications are a program in a high level specifications language, that are compiled into an executable code.

Partial but *realizable* specifications are compiled into a program that satisfies the specifications in [1, 19, 20, 17]. The compilation process is based on a two person game, where the players are the system and the environment. The compiler assumes that the specifications are a determined game, and one of the players has a winning strategy. The system wins the game if it is able to come up with a winning strategy for every state of the environment. The compilation process is exponential (or even double exponential) in the length of the specifications. Another major concern is that the environment is assumed to be known rather than a black box – the environment model is a part of the compilation process.

We note that an environment can change, leading to an unavailable model of the current environment. Even if the new specifications of the environment are available, a program has to be recompiled to accommodate the environment changes. In a sense our control synthesis algorithms parallelize black box or gray box verification algorithms (e.g., [16], [12]) and uses *reflection*, *set* and *replication* capabilities to make the search fast. Recent work proposes techniques for systems composed of deterministic black boxes and white boxes [12]; in such terms our settings may be viewed as a composition of a (deterministic or probabilistic) *reentrant* and *history-oblivious* black box with a (deterministic or probabilistic) rs-box plant. Most of the controls produced by our search engine are controls considered by a model checker. Namely, the produced control is either a chain automaton or an automaton in a form of a chain ending with a loop. A different finite state control automaton is presented for the probabilistic settings. To fit the run-time synthesis requirement we present a search engine that monitors the system and is able to trigger a search for a new control fast in the presence of a changing environment.

The rest of the paper is organized as follows. In Section 2 we present the system architecture and setting. In Section 3 we describe control search algorithms for a deterministic plant and a deterministic environment. In Section 4 we present control search algorithms for a probabilistic or deterministic plant and a probabilistic environment. Conclusions and optimizations appear in Section 5.

2 System Architecture and Settings

A system consists of three components: a *control*, a *plant*, and an *environment*. The system components and their interactions are presented in Figure 1 in the *System* box. The control receives input from a user, then calculates a plant input and sends it to the plant. The plant receives the input from the control and makes some internal calculation while interaction, i.e., sending and receiving some massages from the environment. Then, the plant calculates output message and sends it to the control. The control receives the plant output, makes some internal calculations and creates an output for the user.

Environment. An environment is an infinite non deterministic automaton. We assume that the environment acts as a deterministic or probabilistic automaton for every given period of time. This may be attributed to the approximate fixed location and time of interest of the plant-environment interaction. We also assume that the environment is *reentrant* and *history oblivious*. The *reentrant* property of the environment allows several copies of a plant to interact with the environment independently at the same time; the *history oblivious* property of an environment implies that plant replicas that are set to the same state will have identical interaction with the environment (in the probabilistic case – with the same probability). Thus, the interaction of an environment with a deterministic (probabilistic) plant replica set to a state s that receives a particular control sequence C is identical (implies identical probability for) to the interaction of the environment with another deterministic (respectively, probabilistic) plant that reached the state s in e other possible way, and that receives the same control sequence C.

Plant. A plant is modeled by a deterministic finite automaton, $A_p = \{\Sigma^p_{in}, \Sigma^p_{out}, S_p, start_p, \alpha_p, F_p\}$. The values of the plant input variables are symbols in the plant input alphabet Σ^p_{in}, such that $\Sigma^p_{in} = \Sigma^{pc}_{in} \times \Sigma^{pe}_{in}$, where Σ^{pc}_{in} is an alphabet of the plant input variables in the plant-control interaction and Σ^{pe}_{in} is an alphabet of the plant input variables in the plant-environment interaction. The plant output instances are symbols in the plant output alphabet $\Sigma^p_{out} = \Sigma^{pc}_{out} \times \Sigma^{pe}_{out}$, where Σ^{pc}_{out} is an alphabet of the plant output variables in the plant-control interaction and Σ^{pe}_{out} is an alphabet of the plant output variables in the plant-environment interaction. The plant is a finite automaton, which implies that Σ^{out}_{in} and Σ^{pe}_{out} are also finite. S_p is a collection of plant automaton states, $start_p \in S_p$ is an initial state of the plant automaton, α_p is a transition function

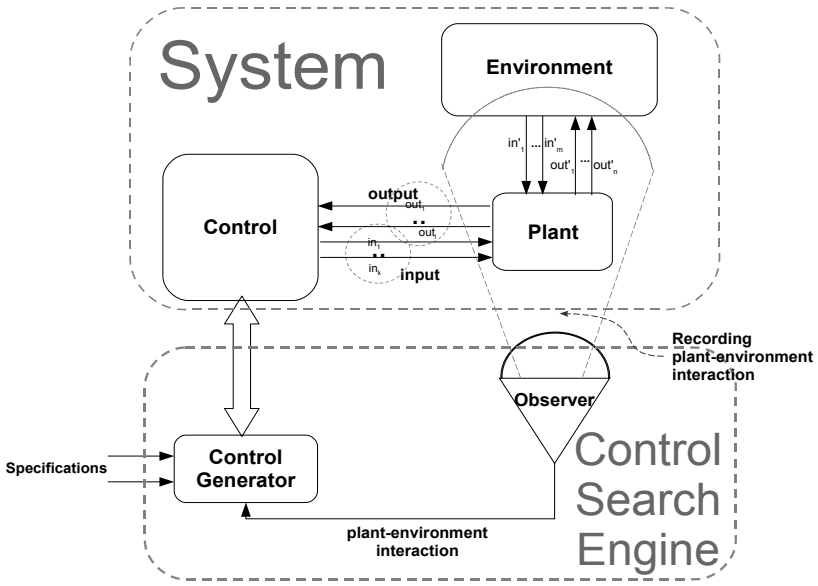

Fig. 1. The system and the control search engine architecture.

where $\alpha_p : S_p \times \Sigma^p_{in} \to S_p \times \Sigma^p_{out}$, and $F_p \in S_p$ is a list of the plant accepting states. α_p is subject to dynamic changes as the environment changes. For example, a spring of a machine on the Moon may react differently than on Earth.

The plant is either a *black box* or an *rs-box* (reflection and set box). A plant is a *black box* if one does not have an access to the plant state, only to the plant input and output. A plant is an *rs-box* if the plant has either the *plant state reflection* capability or the *plant state set* capability, or both. The *plant state reflection* capability implies that the plant state is fully exposed without revealing the plant automaton transition function. During the search procedure we can see the plant transitions from a state to a next state. The *plant state set* capability allows to set a plant to a given state, which implies that the control search engine is able to explore all connected components in the plant automaton graph.

Control. A control is modeled by a deterministic finite IO automaton $A_c = \{\Sigma^c_{in}, \Sigma^c_{out}, S_c, start_c, \alpha_c, F_c\}$. The values of the control automaton input variables are symbols in the control input alphabet Σ^c_{in}, such that $\Sigma^c_{in} = \Sigma^{pc}_{out}$, where Σ^{pc}_{out} is an alphabet of plant output variables in the plant-control interaction. The values of the control automaton output variables are symbols in the control output alphabet Σ^c_{out}, such that $\Sigma^c_{out} = \Sigma^{pc}_{in}$, where Σ^{pc}_{in} is an alphabet of the plant input variables in the plant-control interaction. S_c is a collection of control automaton states, $start_c \in S_c$ is an initial state of the control automaton, α_c is a transition function $\alpha_c : S_c \times \Sigma^c_{in} \to S_c \times \Sigma^c_{out}$, and $F_c \in S_c$ is a list of the control automaton accepting states.

A control automaton produces an output, which is, in turn, a plant input in the plant-control interaction. The control output implies the plant producing an IO sequence in plant-environment interaction, such that IO respects the specifications.

System. A system is an automaton A_{sys}, where $A_{sys} = A_c \times A_p \times A_{env}$. A system state s_{sys} is a combined state of the control s_c, the state of the plant s_p, and the state of the environment s_{env}. The system atomic step (a transition in the system automaton) $a_{sys} = \langle [s_c, s_p, s_{env}], [s'_c, s'_p, s'_{env}], io \rangle$ leads to transition from state $[s_c, s_p, s_{env}]$ to state $[s'_c, s'_p, s'_{env}]$. The transition consists of internal calculations of the system components (the control, the plant, and the environment), of a single interaction of the control with a user, of a single interaction of the control with the plant, and of a single interaction of the plant with the environment, i.e., $io \in (\Sigma^c_{in} \times \Sigma^{pc}_{in} \times \Sigma^{pe}_{in}) \times (\Sigma^c_{out} \times \Sigma^{pc}_{out} \times \Sigma^{pe}_{out})$. The system execution $E = c_1, a_1, c_2, a_2, \ldots$ consists of system

configurations c_i, where $c_i \in (S_c \times S_p \times S_{env})$ (the values of the system state variables), and of atomic steps a_i, so that c_{i+1} is reached from c_i by the execution of a_i.

Specifications. The desired system execution is defined by specifications. A specifications can be in the form of a *goal*, where the *goal* is a desired outcome (typically, in the form of unrealizable specifications), or as a *behavior*, where the *behavior* is a desired finite IO system sequence. For the sake of simplicity, the desired execution of a system is described by a set of behaviors.

The specifications are unrealizable [1], i.e., the specifications do not define a control. No meaningful *realizable* specifications exist for unpredictable dynamic environments. Unrealizable specifications are potentially more abstract and shorter than realizable specifications, and, therefore, less prone to human mistakes. Thus, unrealizable specifications allow better human-specifications interface than the human-program interface.

The control search engine searches for a control that makes a system to respect unrealizable specifications by on line experimentation on the plant replicas. During the on line experimentation, we implicitly check whether the unrealizable specifications define *weakly realizable* specifications, given the behavior restriction on the current environment. A successful search for a control implicitly identifies the *weakly realizable* specifications, and explicitly identifies the implementation that respects the specifications.

We consider two types of system behavior sequences, a *one shot behavior* and a *periodic behavior*. A *one shot behavior* is an IO sequence of length k produced by the system automaton, $bhv = \{io_1, \ldots, io_k\}$. We denote $bhv[i]$ to be the i-th entry in the behavior sequence, $bhv[i] = io_i$, and bhv_j to the suffix of size j of the behavior sequence, $bhv_j = \{io_{k-(j+1)}, \ldots, io_k\}$. A system *conforms to* the behavior bhv during an execution E if $E = ..., c_j, a_j, \ldots, c_{j+i}, a_{j+k}, ...$, such that $a_j = \langle *, *, io_1 \rangle, \ldots, a_{j+k} = \langle *, *, io_k \rangle$.

A *periodic behavior* sequence is defined by a *one shot behavior* of length P, $bhv = \{io_1, \ldots, io_P\}$, where P is the period length. A system *conforms to* the period behavior $bhv_p = \langle bhv, P \rangle$ during an execution E if there are infinitely many consequent subsequences of size P in E, so that the system conforms to the behavior bhv in every subsequence. A desired execution of an ongoing system is defined by a *periodic behavior*, while a *one shot behavior* defines an execution of a code section, e.g., a terminating function execution.

A control automaton that fits a one shot behavior is the chain automaton. A control automaton that fits a periodic behavior is the chain automaton for a one shot behavior with a loop.

Control search engine. The control search engine architecture is presented in Figure 1 in the *Control Search Engine* box. The control search engine receives as an input specifications. The control search engine outputs a control that makes a plant to respect the specifications, i.e., the control makes the plant to produce an IO sequence that respects the behavior, when executing in the current environment.

The control search engine has two main components, a *Control Generator* and an *Observer*. Next we present in detail the control search engine components and their interactions.

• **Control generator.** The control generator calculates the control automaton either by using an exhaustive search and on-line experimentation on replicated plants [14] or by using experiments for constructing a plant automaton and searching it to find the control automaton. The control automaton is continuously adapted by the control search engine to accommodate changes in the environment.

The control generator algorithm differ according to the available plant capabilities, *plant state reflection* and *plant state set*. We can use dynamic plant replication to reduce the number of replicated plants in case a *plant state reflection* capability is available: Algorithms I to IV are designed for a deterministic plant and a deterministic environment. Algorithms III and IV can be adapted to have dynamic plant replication, which implies a better run-time complexity. Algorithms V and VI are designed for probabilistic environment, where the plant can be either deterministic or probabilistic. Figure 2 presents the list of control search algorithms for different settings, the total number of steps in all experiments, and the length of the longest experiment. N_{max} is the upper bound on the number of plant states $N = |A_p|$.

• **Observer.** One functionality of the *Observer* is to provide access to values of the plant-environment interaction variables, which cannot be accessed directly. Another functionality of the *Observer* is to provide a sanity check for the plant output in the plant-control interaction: The control generator must receive a reliable plant state in order to be able to evaluate the quality of a control. Input and output exchange between the control and the plant is subject to dynamic unexpected changes and, therefore, is unreliable. One possible example of such a change is an incorrect connection of the output wires of the plant to the

control module. For example, the wire of the plant output variable out_i is plugged in the place of the control input variable in_j and the wire of the plant output variable out_j is plugged in the place of the control input variable in_i. Thus, monitoring the control-plant interaction using only the input and output variables does not allow reliable detection of the real status of the plant, which may lead to a totally erroneous conclusion about the way the control is functioning. The observer inputs are used to solve such problems.

3 Control Search Algorithms for Deterministic Plant and Deterministic Environment

In this section we present algorithms for a deterministic environment and a deterministic plant (algorithms *I-IV* in Figure 2). Each algorithm has two threads, *Control Search Thread* and *Monitoring Thread*. The *Control Shearch Thread* finds a control that respects the provided specifications if such a control exists. The *Monitoring Thread* monitors the quality of the currently executing control. If the current control is not respecting the specifications, the monitoring thread restarts the control generator thread.

Algorithm I: Static plant replication, no state state, no state reflection. The algorithm code is presented in Fig. 3. The algorithm input is a periodic behavior tuple $bhv_p = \langle bhv, P \rangle$, the upper bound on the number of states in the plant automaton N_{max}, and the plant input alphabet in the plant-control interaction is Σ_{in}^{pc}.

The desired periodic behavior may begin from every possible plant state. Therefore, we examine controls that reach all states in the connected component of the plant current state s_{p-curr}, and search for a control that obtains the periodic behavior starting from every reached state. The control sequences with prefixes of lengths 0 to N_{max} are used to reach every state in the connected component of the current plant state s_{p-curr} (line 1). We search for a control sequence suffix that obtains a periodic behavior from a plant state reachable from s_{p-curr} by trying all possible control sequences of length P concatenated $N_{max} + 1$ times (line 1).

There are *IN* possible input sequences that need to be examined (see lines 1-2 for the exact value of *IN*). The generator creates IN plant replicas (line 3), each in the current state s_{p-curr} of the plant. The generator executes each input on a distinct plant replica in parallel and records the subsequent plant-environment interaction IO using the *Observer* (lines 5-6). If the recorded IO sequence has an integer number of consequent appearances of bhv (line 7), then the algorithm returns the corresponding control $input_i$ (line 8).

If there does not exist a control that begins from the current plant state s_{p-curr} and that respects the behavior in the given environment, the algorithm terminates and returns *no control* (line 9). The algorithm is unable to find a control sequence that starts from a plant state that is not reachable from s_{p-curr} due to lack of the *plant state set* capability.

The monitoring thread code appears in lines 10-12. The monitoring thread receives from the *Observer* a reliable record of the plant-environment interaction. For any given IO suffix of length P, there must be a suffix of bhv of length i, $0 \leq i \leq P$, that is followed by a prefix of bhv_p of length $P - i$. The monitoring thread keeps a record of the last P inputs and outputs in the plant-environment interaction and checks that

Algorithm	**Reflection**	**Set**	**Total Number of Steps in All Experiments**	**Longest Experiment**
I	¬Available	¬Available	$O((PN_{max})\lvert\Sigma_{in}^{pc}\rvert^{N_{max}P})$	$O(PN_{max})$
II	Available	¬Available	$O(N\lvert\Sigma_{in}^{pc}\rvert^{P})$	$O(P)$
III	¬Available	Available	$O(N\lvert\Sigma_{in}^{pc}\rvert)$	$O(N)$
IV	Available	Available	$O(N\lvert\Sigma_{in}^{pc}\rvert)$	$O(1)$
V(Probabilistic)	Available	Available	$O(N^2 \cdot \lvert\Sigma_{in}^{pc}\rvert \cdot \lvert\Sigma_{in}^{pe}\rvert \cdot \lvert\Sigma_{out}^{pe}\rvert \cdot SP)$	$O(1)$
VI(Probabilistic)	Available	¬Available	$O(N^2 \cdot \lvert\Sigma_{in}^{pc}\rvert \cdot \lvert\Sigma_{in}^{pe}\rvert \cdot \lvert\Sigma_{out}^{pe}\rvert \cdot SP)$	$O(1)$

Fig. 2. Properties of the control search algorithms.

Algorithm I (Static Replication from Current State, No Reflection)
input: $bhv_p = \langle bhv, P\rangle$, N_{max}, Σ_{in}^{pc}
output: control C
Control Search Thread
1 $input^* = \{(\Sigma_{in}^{pc})^i((\Sigma_{in}^{pc})^P)^{(N_{max}+1)} : 0 \le i \le N_{max}\}$
2 $IN = |input^*|$
3 replicate IN plants from s_{p-curr}: $p_1, p_2, \ldots, p_{IN}$ //Plant replication
4 **in parallel**
5 $\forall input_i \in input^*$
6 $execute(p_i, input_i)$ and $IO_i = Observer.recordFor(|input_i|)$
7 if $IO_i == IO_{start} bhv^x$// $x \in \mathbb{N}$
8 return $C = input_i$
9 return *no control*
Monitoring Thread
10 while (true)
11 if $Observer.recordFor(P) \not\Rightarrow bhv$
12 if $\nexists ControlGenarator\ ControlGenerator.start()$

Fig. 3. Algorithm I: static replication from current state, no state reflection.

every such suffix contains a prefix and a suffix of bhv as described above. The monitoring code is identical in all algorithms for deterministic settings.

The sum of control sequence lengths examined over all replicas is used to measure the algorithm complexity. Algorithm I experiments complexity is $O((PN_{max})|\Sigma_{in}^{pc}|^{N_{max}P})$ [1].

Theorem 1. *The algorithm in Fig. 3 finds a control sequence C such that when C is applied to the plant in the current state s_{p-curr}, the plant produces an IO sequence that respects the periodic behavior $bhv_p = \langle bhv, P\rangle$ if such a control exists.*

Proof. Assume towards a contradiction that there exists a control $C = \{in_1, in_2, \ldots, in_j, (in_{j+1}, \ldots, in_{j+1+p})^*\}$, $j \le N_{max}$, that obtains the periodic behavior bhv_P. However, the search algorithm in Fig. 3 does not find C, i.e., returns *no control* or a control that does not imply the periodic behavior.

The existence of control C implies the existence of a plant state s_{start}, such that s_{start} is reached from s_{p-cur} by executing control $C_1 = \{in_1, in_2, \ldots, in_j\}$, and the execution of $C_2 = \{in_{j+1}, \ldots, in_{j+1+p}\}$ beginning from s_{start} exhibits the desired behavior bhv_p.

Every state in the connected component of s_{p-cur} in the graph describing the plant automaton, in particular s_{start}, can be reached by executing a control of length N_{max} or less. In line 1 the algorithm generates all controls of size less than or equal to $N_{max} + P$; in particular the algorithm produces a control $C_{try} = C_1 \cdot (C_2)^{N_{max}+1}$. The control C_{try} is experimented on a replica. By the fact that the environment is reentrant and is history oblivious the experiment succeeds. Thus, the algorithm finds a control that brings the plant to the state s_{start} and produces an IO sequence in the plant-environment interaction with a subsequent integer number of behavior sequences.

Next we prove that a control returned by the algorithm implies an infinite loop. Any sequence chosen by the algorithm has a suffix with $N_{max} + 1$ repetitions of a control sequence of length P. At least two such repetitions start in the same plant state, s_r. Therefore, there is a subsequence of an integer number of control C_2 repetitions that starts in s_r and ends in s_r while producing the IO sequence that has the subsequent integer number of behavior sequences. s_r is reached by the control that is synthesized by the algorithm and the periodic behavior is obtained by that control, therefore the contradiction.

[1] We note that for non-parallel search settings it may be possible to obtain a better complexity, when N_{max} is significantly greater than N, using the results in [24, 6, 2] as suggested in [12].

Algorithm II: Static plant replication, state set, no state reflection. The *plant state set* allows the control search engine to explore all connected components in the plant automaton graph and always find a control if it exists. Even though we do not have the *plant state reflection* capability, i.e., we do not have access to a plant state, the *plant state set* allows us to set the plant to all possible distinct states. Note that knowing the set of the plant states does not imply knowledge of the plant automaton: the plant automaton transition function can change due to the changes in the current environment.

The *Control Generator Thread* is designed as follows. We create all possible control (input) sequences of length P, $(\Sigma_{in}^{pc})^P$. We try each control sequence from every plant state. If the control sequence produces IO sequence that equals to the desired bhv the algorithm returns a tuple consisting of a beginning plant state and a control sequence $\langle s_{begin}, C\rangle$. The control is an infinite loop in which we set the plant to state s_{begin} by *set* instruction and then execute control sequence C.

The *Monitoring Thread* is the same as in Figure 3. The complexity of the experiments of Algorithm II is $O(PN)$.

Algorithm III: Static plant replication, no state set, state reflection. In these settings the plant state is fully exposed. During the search procedure we can see the plant transition from one state to the next, and, therefore, gain a full description of a connected component of the current plant state s_{p-curr} in the plant automaton.

Due to *static plant replication* we are unable to exploit the reflection attribute in order to improve the algorithm complexity: we release all plant replicas simultaneously and are only able to observe the state transitions. Thus, the algorithm in these settings is identical to Algorithm I.

Still, we are able to exploit the *plant state reflection* capability in case N_{max} is unknown and we may have several stages for generating replicas. We consider a multi-phase search algorithm: N_{max} is set to initial value N_0 and we execute Algorithm I with the given N_{max} value. If the control is found the multi-phase algorithm terminates. Otherwise, we double the value of N_{max} and execute algorithm I again. In each phase all the visited plant automaton states and edges i.e., recording which inputs are applied to the plant state, are recorded. If in the next phase we do not encounter new states then we may conclude that we have explored the whole connected component of s_{p-curr} in the plant automaton. If the control was not found so far we terminate and return *no control*, which implies that there is no control that begins in s_{p-curr}.

• **Algorithm III: Dynamic plant replication, no state set, state reflection.** The algorithm is presented in Figure 4. The *plant state reflection* and *dynamic plant replication* capabilities allows us to learn the connected component of s_{p-curr} in the most efficient manner by experimentation on the plant replicas: we record the visited plant states and prune search paths that reach already visited states (lines 1-14). Note that the algorithm explores all outgoing edges of each state s_i reachable from s_{p-curr} exactly once.

The variable *states* is the list of states that were reached, but have not been fully explored (line 1), i.e., the algorithm have not checked all outgoing edges of a state $s \in states$. The variable *visitedStates* is the list of states for which their outgoing edges were fully explored by the algorithm (line 2). The variable *automatonComponent* is a list of transition in the plant automaton graph observed by the algorithm (line 3). The algorithm chooses a state $s_i \in states$ (line 5-6) and creates plant replicas set in state s_i (line 6). The algorithm explores all outgoing edges from s_i by applying every possible input on a different plant replica in parallel (lines 7-9). If a state s'_j reached from s_i by applying input in_j, was not not explored yet we add s'_j to the list of states to be explored and add the recorded transitions to the automaton component graph (lines 11-13). Finally, when no new states are encountered, we conclude that we have explored the connected component of s_{p-curr} in the plant automaton graph and construct the observed connected component graph A'_p (line 14).

Next, we search the connected component for a control that produces a periodic behavior in the same manner as in Algorithm I (lines 15-21). The algorithm returns *no control* if there is no control that starts in s_{p-curr}.

In the worst case, the algorithm explores all edges in the graph describing the plant automaton during the experimentation stage, namely, $N|\Sigma_{in}^{pc}|$ edges. Therefore, the algorithm experimentation complexity is $O(N|\Sigma_{in}^{pc}|)$.

Algorithm III
(Dynamic Plant Replication, No State Set, State Reflection)
input: $\langle bhv, P\rangle$, Σ_{in}^{pc}
output: control C
Control Search Thread
//Plant Automaton Discovery
$states = \{s_{p-curr}\}$//states to explore
$vistedStates = \{\}$//explored states
$automatonComponent = \{\}$
while ($states \neq \emptyset$)
  $\forall s_i \in states$
    $states = states \backslash \{s_i\}$
    replicate $|\Sigma_{in}^{pc}|$ plants in state s_i: $p_1, \ldots, p_{|\Sigma_{in}^{pc}|}$
    in parallel $\forall p_j$// Experimentation
      $\forall in_j \in \Sigma_{in}^{pc}$
        $execute(p_j, in_j)$ //plant state after execution is s'_j
          if $s'_j \notin visitedStates$
            $states = states \cup \{s'_j\}$
            $automatonComponent = automatonComponent \cup \langle s_i, in_j, s'_j\rangle$
$A'_p = constructAutomata(automatonComponent)$ //connected component automaton
//Search For Control
$n = |A'_p|$
$input^* = \{(\Sigma_{in}^{pc})^i((\Sigma_{in}^{pc})^P)^{(n+1)} : 0 \leq i \leq n\}$
$\forall input_i \in input^*$
  $execute(p_i, input_i)$ and $IO_i = Observer.recordFor(|input_i|)$
  if $IO_i == IO_{start} bhv^x$//$x \in \mathbb{N}$
    return $C = input_i$
return *no control*
Monitoring Thread
while (true)
  if $Observer.recordFor(P) \nRightarrow bhv$
    if $\nexists ControlGenarator\ ControlGenerator.start()$

Fig. 4. Algorithm III: dynamic replication from current state and state reflection.

Algorithm IV: Static plant replication, state set, state reflection. In these settings we are able to set a plant to every possible state. We learn the plant automaton transition function by applying to every plant state all possible inputs and observing the reached state. We use the plant states and the plant automaton transition function to construct the plant automaton.

We search the graph off-line for a periodic control. Starting from every state s_i we check all possible input (control) sequences of of size P. If a sequence C drives the plant to produce IO sequence, such that $bhv = IO$, and the last transition of C brings us back to state s_i, then the algorithm returns C. The algorithm complexity is $O(N|\Sigma_{in}^{pc}|)$, which is the total number of steps in all experimentations.

4 Control Search Algorithms for Deterministic/Probabilistic Plant and Probabilistic Environment

Nature (the environment) may exhibit probabilistic behavior, responding to actions differently but with the same probability distribution. The fact that the plant is not aware of the entire state (or output) of the environment can be modeled by probabilistic plant-environment IO interaction, where environment reactions to the plant inputs are probabilistic. In this section we turn to assume that the environment is probabilistic and a plant is either probabilistic or deterministic. The probabilistic plant has a probabilistic transition

function: for every two states s and s' there is a probability pr, $0 \leq pr \leq 1$, of transition from the state s to the state s' upon input σ while implying io in the plant-environment interaction. The transition function probabilities are not known and may change if the environment changes. The suggested algorithms for probabilistic settings are applicable in both the plant-environment combinations, in which the environment is probabilistic and the plant is either probabilistic or deterministic.

A control search algorithms for probabilistic environment settings has to be executed all the time (unlike for deterministic environment settings), due to the fact that the environment reactions are probabilistic, and, moreover, the plant transition function is probabilistic too (in case the plant is probabilistic). The control search algorithm decides on the next step based on the current state of the plant and the last IO exchange in the plant-environment interaction during run time.

Preprocessing. We assume that during preprocessing we operate on plant replicas with *plant state reflection* and *plant set* capabilities. Thus, we are able to create all plant states and learn the plant automaton transition function by observation. During the preprocessing stage we learn the (probabilistic) plant automaton transition function by experimentations. We calculate additional data structures based on the plant automaton.

We assume knowing the smallest probability value pr_{min} in the plant automaton transaction function. Alternatively, we may assume that pr_{min} is the smallest positive float value that can be defined by numeric capabilities of a computer that executes the preprocessing. *Safety paramter (SP)* is the number of experiments required in order to find edges with the minimal probability pr_{min} with high probability. Thus, $SP \cong 1/pr_{min}$.

•*Probabilistic Plant Automaton Graph (PPAG) table.* We compute a table where each entry is of the form $PPAG[s_i, s_j, \sigma, io] = pr_{i,j}$, where s_i and s_j are plant states, $pr_{i,j}$ is the probability to reach s_j from s_i with input $\sigma \in \Sigma_{in}^{pc}$ while producing $io \in \Sigma_{in}^{pe} \times \Sigma_{out}^{pe}$ in the plant-environment interaction.

We compute $PPAG[s_i, s_j, \sigma, io]$ by experimentation on plant replicas as follows. For each two states s_i and s_j and for each $\sigma \in \Sigma_{in}^{pc}$, we create $SP \cdot |\Sigma_{in}^{pe}| \cdot |\Sigma_{out}^{pe}|$ plant replicas set to the state s_i and give each plant replica the input σ. We compute the percentage of times of the total number of experiments the plant transits to the state s_j while producing io in the plant-environment interaction. The experiments complexity of calculating all entries in the $PPAG$ table is $O(N^2 \cdot |\Sigma_{in}^{pc}| \cdot |\Sigma_{in}^{pe}| \cdot |\Sigma_{out}^{pe}| \cdot SP)$.

• *"Removing" unsafe states from PPAG table.* During the experiments we record the plant automaton transitions $[s_i, s_j, \sigma, io]$, in which a plant replica suffers a damage when moving from state s_i to state s_j upon input σ. We consider transitions and states in which a plant suffers a damage to be *unsafe*. Thus, we forbid application of the input σ on the plant replica set to the state s_i. The plant will not make unsafe transitions to unsafe states. Note, that the added constraint implies that the plant in the state s_i is not able to reach some other states besides *unsafe* states.

•*Behavior Suffix Probability (BSP) table.* An entry in the *Behavior Suffix Probability (BSP)* table is $BSP[s, j] = [pr, \sigma]$, which means that the maximal probability to obtain the behavior sequence suffix of length j, bhv_j, when the plant is in state s_i is pr, and this probability is achieved by applying an input σ. We denote by $BSP[s, j].pr$ the value of the first variable in the $BSP[s, j]$ entry and by $BSP[s, j].\sigma$ the value of the second variable in the $BSP[s, j]$ entry.

We calculate BSP table iteratively. First, we calculate all entries for $j = 1$. Then, based on the calculated entries, we calculate all entries for $j = 2$, and so on, till we calculate all entries for $j = k$. In order to calculate an entry $BSP[s, 1]$ we check all entries in $PPAG$ table of the form $PPAG[s, *, *, io_k]$. Suppose $pr_{max} = max\{PPAG[s, s_i, \sigma_j, io_k] : s_i \in S_p, \sigma_j \in \Sigma_{in}^{pc}\}$ and σ_{max} is the input that produces io_k with the highest probability pr_{max} among all other possible inputs. This implies that $BSP[s, 1] = [pr_{max}, \sigma_{max}]$.

Assume we have calculated all entries in BSP table for $j \leq m$. In order to calculate entries $BSP[s, m+1]$ for every plant state s we calculate $pr(s, bhv_{m+1}, \sigma)$ – a probability to obtain the control sequence suffix bhv_{m+1} starting from the plant set to the state s by applying the input $\sigma \in \Sigma_{in}^{pc}$. We compute $pr(s, bhv_{m+1}, \sigma)$ by exploring all neighbors of the state s that can be reached by the input σ: $pr(s, bhv_{m+1}, \sigma) = \sum_{s'} PPAG[s, s', \sigma, bhv[k - (m + 1)]] \cdot BSP[s', j]$. Finally, we choose an input σ_{max} that implies bhv_{m+1} starting from s with the highest probability pr_{max} ($pr_{max} = max\{pr(s, bhv_{m+1} : \sigma \in \Sigma_{in}^{pc}\}$) among all other inputs in Σ_{in}^{pc}. Thus, $BSP[s, m + 1] = [pr_{max}, \sigma_{max}]$. The BSP table is computed based on

the *PPAG* table only, thus the computation is off-line, i.e., the computation does not require experimentations.

•*Reachability Probability (RP) table.* An entry in the *Reachability Probability (RP)* table is $RP[s, s'] = [pr, \sigma]$, where pr is the maximal probability pr for reaching a plant state $s \in S_p$ from a plant state s and σ in the first input that should be to the plant to reach the state s' with the probability pr. The maximal probability is an approximation since we compute the probability of the best path rather than the probability of all the paths.

We denote by $RP[s, s'].pr$ the value of the first variable in the $RP[s, s']$ entry and use $RP[s, s'].\sigma$ to denote the value of the second variable in the $RP[s, s']$ entry.

We compute the table entries for every two state $s_i, s_j \in S_p$ by invoking a modified version of *Dijkstra* algorithm [7] on the plant automaton graph as represented by the *PPAG* table. In the modified *Dijkstra* algorithm we use *max* function instead of *min* function and multiplication instead of summation. We invoke the modified *Dijkstra* algorithm with s_i as initial state and find a path with maximal probability to reach every other state s_j from s_i. The *RP* table is computed based on the *PPAG* table only, thus the computation is off-line.

•*Detecting and "removing" deadlock states in the PPAG table.* We call a state s a *deadlock state* if there is no control sequence that starts in s and exhibits a desired behavior. A deadlock state s entry in the *BSP* table is of the form $BSP[s, k] = [0, *]$.

A control search algorithm with no *plant state set* capability will be in a livelock after reaching a deadlock state. Thus, for such algorithms we "remove" deadlock states from *PPAG* by avoiding a control symbol that drives the system to one of these states, in the same manner as we "removed" *unsafe* states. We find all *deadlock* states reachable from s_{p-curr} by using *BFS* or *DFS* algorithms [7] and denote them by S_{reach}.

We mark an input $RP[s, s'].\sigma$ as *inapplicable from the state* s if $s' \in S_{reach}$. Then, we recompute the *BSP* table and repeat the procedure for removing deadlock states. We repeat the procedure till no more deadlock states are found. The computation of the deadlock states is done using *RP* and *BSP* tables only, thus the computation is off-line.

Algorithm V: Static plant replication, state set, state reflection. The control search algorithm is presented in Fig. 5. The algorithm executes the loop in lines 2-9 infinitely often. The algorithm finds a state s_{start} that has the highest probability that the desired behavior will be obtained when starting from the plant set to s_{start}. We find the state s_{start} using the *BSP* table and sets the plant to the state s_{start} (lines 2-3). The variable *index* is the length of the obtained control sequence so far (line 4). We execute the input as retrieved from the *BSP* table in order to achieve the current desired behavior sequence suffix with the highest probability (line 6) until the whole control is found (line 5). If the transition to the next state s_{next} produces *IO* corresponding to the current entry in the behavior sequence suffix, and the probability to obtain the rest of the behavior from the state s_{next} is bigger than the probability to obtain the behavior from s_{curr} then we increase the length of the obtained control sequence (line 8). Otherwise, if we reached a state from which we have a lower probability to obtain the rest of the behavior, we initialize the control sequence length and set the current state to the best stating state in the same manner as in line 2 (line 9).

The algorithm experimentation complexity equals to the experimentation complexity of computing the *PPAG* table and is $O(N^2 \cdot |\Sigma_{in}^{pc}| \cdot |\Sigma_{in}^{pe}| \cdot |\Sigma_{out}^{pe}| \cdot SP)$.

Algorithm VI: Static plant replication, no state set, state reflection. The algorithm is presented in Fig. 6. First, we present function *ReachBetterStart* (lines 9-11). The function makes transitions till it finds a state s reachable from s_{p-curr}, such that when starting in s a better probability for completing the desired behavior than when starting from s_{p-curr} is obtained.

In the *Control Search Thread* we execute the loop in lines 2-8 move to the best reachable starting state using *ReachBetterStart* function (line 3) and then we start searching for control in the same manner as in Algorithm *V* (lines 4-8). In case the algorithm deviates from the best possible path to obtain the control, we initialize the control sequence size and search for the best state reachable from the current plant state to start the control search anew (line 9). The algorithm experimentation complexity (computing the *PPAG* table) is $O(N^2 \cdot |\Sigma_{in}^{pc}| \cdot |\Sigma_{in}^{pe}| \cdot |\Sigma_{out}^{pe}| \cdot SP)$.

Algorithm V
(Probabilistic, Static Plant Replication, State Set, State Reflection)
input: $bhv_p = \langle bhv, P\rangle(|bhv| = k)$, Σ_{in}^{pc}
On-Line Control Search Thread
1 while $(true)$
2 $s_{start} = \{s_j : BSP[s_j, k].pr = max\{BSP[s_i, k].pr, 1 \leq i \leq N\}\}$
3 $setPlant(s_{start})$ $//s_{p-curr} = s_{start}$
4 $index = 0$
5 while $(index \leq k)$
6 $[s_{next}, io] = execute(s_{curr}, BSP[s_{curr}, k - index].\sigma)$
7 if $io = bhv[index]$ and $BSP[s_{next}, k - index - 1] > BSP[s_{curr}, k - index]$
8 $index++$
9 else $index = 0$, $s_{curr} = \{s_j : BSP[s_j, k].pr = max\{BSP[s_i, k].pr, 1 \leq i \leq N\}\}$

Fig. 5. Algorithm V: probabilistic environment, static replication, state state and state reflection.

Algorithm VI
(Probabilistic, Static Plant Replication, No State Set, State Reflection)
input: $bhv_p = \langle bhv, P\rangle(|bhv| = k)$, Σ_{in}^{pc}
On-Line Control Search Thread
1 while $(true)$
2 $ReachBetterStart(s_{p-curr})$
3 $index = 0$
4 while $(index \leq k)$
5 $[s_{next}, io_{index}] = execute(s_{curr}, BSP[s_{curr}, k - index].s)$
6 if $io_{index} = bhv[index]$ and $BSP[s_{next}, k - index - 1] > BSP[s_{next}, k - index]$
7 $index++$
8 else $index = 0$, $ReachBetterStart(s_{p-curr})$
ReachBetterStart
input: s_{curr}, $bhv_p = \langle bhv, P\rangle(|bhv| = k)$
9 while$(\exists s : RP[s_{curr}, s] \cdot BSP[s, k] > BSP[s_{curr}, k])$
10 $\sigma_{max} = \{RP[s_{curr}, s].\sigma : max(RP[s_{curr}, s].pr), s \in |A_p|\}$
11 $execute(s_{curr}, \sigma_{max})$

Fig. 6. Algorithm VI: probabilistic environment, static replication from current state and state reflection.

Monitoring and triggering control search. A change in the environment is, in fact, a change in the probabilities of the transitions. Alas, we may be able to find such a change only after visiting the same state $SP \cdot |\Sigma_{in}^{pe}| \cdot |\Sigma_{out}^{pe}|$ times as we did during the preprocessing stage. We compare the probabilities computed in the preprocessing stage and the probabilities observed during run time. In case there is a significant change in the probabilities the monitoring thread initiates recalculation of the preprocessing stages.

5 Conclusions and Optimizations

A framework for run-time synthesis is presented, where on-line parallel experiments are used to rapidly find a control. The framework also separates the plant from the environment and demonstrate the importance of the plant state *reflection* and plant state *set* attributes. Several optimizations and directions for future research are possible:
Extracting abstractions. One possible optimization of the sequence search procedure is to use subgoals. A subgoal is a mile stone a control has to reach in order to reach its goals. Then the final control is a combination of controls for subgoals. Some subgoals can be provided to the control generator in order to

accelerate the search. Assume that there is a sequence that the control search engine identified as a sequence that leads to achieving one of the subgoals, e.g., sending a message over a network. The control search engine stores the sequence as a procedure abstraction and can use it when recomputing a control.

Partial reflection and set. Assume that there is *partial plant state reflection*, i.e., not all states are transparent, but the number of non-transparent states between any two transparent states after any possible dynamic change of the plant in the environment settings is *dist*. In order to assume *partial plant state reflection*, one must be able to prove that the distance between two transparent states is constant for every possible control execution. The algorithm for static replication of a plant from a current state with *plant state reflection* will work in these settings as well. In this case, the search time complexity increases by the factor of $|\Sigma_{in}^{pe}|^{dist}$, which is a constant.

A *partial state set* that ensures that after any dynamic change there exists at least one state with the *state set* attribute in each connected component of the plant automaton graph is also helpful.

Acknowledgments. We thank Moshe Vardi and Doron Peled for reading and discussing the paper in early stages of the work.

References

1. M. Abadi, L. Lamport, P. Wolper. "Realizable and Unrealizable Specifications of Reactive Systems". *Proceedings of the 16th International Colloquium on Automata, Languages and Programming (ICALP'89)*, pp. 1-17, Stresa, Italy, July 1989.
2. D. Angluin. "Learning Regular Sets from Queries and Counterexamples". *Information and Computation*, vol. 75(2), pp. 87-106, 1987.
3. A. Arora, M. Demirbas and S. Kulkarni. "Graybox Stabilization", *International Conference on Dependable Systems and networks* (DSN), pp. 389-400, 2001.
4. O. Brukman, S. Dolev. "Recovery Oriented Programming". *Proc. of 8th International Symposium on Stabilization, Safety, and Security of Distributed Systems (SSS'06)*, pp. 152-168, Dallas, Texas, USA, November 2006.
5. O. Brukman, S. Dolev, E.K. Kolodner. "Self-Stabilizing Autonomic Recoverer for Eventual Byzantine Software". *Proc. of the IEEE International Conference on Software-Science, Technology & Engineering (SWSTE'03)*, pp. 20-29, Herzelya, Israel, November 2003.
6. T. S. Chow. "Testing Software Design Modeled By Finite-State Machines". *IEEE Transactions on Software Engineering*, vol. 4, pp. 178-187, 1978.
7. T. H. Cormen, C. L. Leiserson, R. L. Rivest. *Introduction to Algorithms*. The MIT press, 1996.
8. "Danger to Aircraft from Volcanic Eruption Clouds". https://volcanoes.usgs.gov/Hazards/Effects/Ash+Aircraft.html, 2000.
9. *Disaster Recovery Journal*. http://www.drj.com/, 2008.
10. S. Dolev. *Self-stabilization*. The MIT press, March 2000.
11. IBM. Autonomic computing. http://www.research.ibm.com/autonomic, 2001.
12. E. Elkind, B. Genest, D. Peled, H. Qu. "Grey-Box Checking". *Formal Techniques for Networked and Distributed Systems (FORTE'06)*, pp. 420-435, Paris, France, 2006.
13. D. J. Musliner. "Imposing Real-Time Constraints on Self-Adaptive Controller Synthesis". *Proc. of the International Workshop on Self-Adaptive Software*, pp. 143-160, Oxford, England, April 2000.
14. J. von Neumann. "The Theory of Self reproducing Automata". University of Illinois Press, Urbana, Illinois, USA, 1966.
15. D. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. "Recovery Oriented Computing(ROC): Motivation, Definition, Techniques and Case Studies". UC Berkeley Computer Science Technical Report UCB/CSD-02-1175, Berkeley, CA, March 2002.
16. D. Peled, M. Y. Vardi, M. Yannakakis. "Black Box Checking". *Journal of Automata, Languages and Combinatorics*, vol. 7(2), pp. 225-246, 2001.
17. N. Piterman, A. Pnueli, Y. Sa'ar. "Synthesis of Reactive(1) Designs". *Proc. of the 7th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'06)*, pp. 364-380 , Charleston, SC, USA, January 2006.
18. W. H. Phillips. "Journey in Aeronautical Research: A Career at NASA Langley Research Center". http://history.nasa.gov/monograph12/monograph12.htm, Chapter 11, 1998.

19. A. Pnueli, R. Rosner. "On the Synthesis of a Reactive Module". *Proceedings of the 16th ACM Symposium on Principles of Programming Languages (POPL'89)*, pp. 179-190, Austin, Texas, USA, January 1989.
20. A. Pnueli, A. Zaks, L. D. Zuck. "Monitoring Interfaces for Faults". *Electronic Notes in Theoretical Computer Science*, vol. 144(4), pp. 73-89, 2006.
21. P. Robertson, B. Williams. "Automatic Recovery from Software Failure". *Communications of the ACM*, vol. 49(3), pp. 41-47, 2006.
22. T. Rothamel, Y. A. Liu, C. L. Heitmeyer, E. I. Leonard. "Generating Optimized Code from SCR Specifications". *Proc. of the ACM SIGPLAN/SIGBED conference on Language, compilers and tool support for embedded systems (LCTES06)*, pp. 135-144, Ottawa, Ontario, Canada ,June 2006.
23. G. Tziallas, B. Theodoulidisb. "A Controller Synthesis Algorithm for Building Self-Adaptive Software". *Information and Software Technology*, vol. 46(11), pp. 719-727, 2004.
24. M. P. Vasilevskii. "Failure Diagnosis of Automata". *Kibernetika*, vol. 4, pp. 299-347, 1973.