ADAPTATION OF CONTROL PROGRAMS TO ASYNCHRONOUS I/O UPDATES

Richard Šusta *

* *Department of Control Engineering,*
Faculty of Electrical Engineering, Prague
http://dce.felk.cvut.cz/susta/

Abstract: The paper discusses the problems with adaptation of industrial control programs from synchronous to asynchronous I/O update when peripherals are redeveloped with the aid of control area networks (CANs). CANs decrease necessary wires between controllers and technology, but they have lower data update rates in comparison with I/O buses. To accelerate and optimize I/O polling, I/O data can be updated asynchronously to evaluations of control algorithms.
If a control program was written with the assumption of synchronous I/O updates, its adaptation to asynchronous I/O updates requires its partial modifications to exclude possibility of dataraces. The simplest and robust way represents simulating of synchronous I/O updates by copying all input values into an array before each evaluation of the program, but such spare buffering generally increases responses to events. Therefore, the paper presents conditions for running the program (or its parts) without spare buffering of I/O data.

Keywords: PLC, scan dataraces, static verification, transfer sets, APLCTRANS

1. INTRODUCTION

Controlling or supervising a manufacture system usually needs cyclic transfers of great amounts of data between I/O hardware and computers. For that reason, industrial programs frequently operate directly in a cyclic manner, which requires that the programs have limited execution times and always terminate normally under any circumstances, otherwise a fatal error occurs. We will call such cyclic manner programs as *I/O handler programs*, in short *IOH-programs*, according to event handlers utilized in operating systems that have very similar properties.

The execution of an IOH-program can be scheduled by many ways, most frequently as:

- **continuous task** — its new execution begins after finishing a previous one;
- **periodic task** — the program is started at regular time intervals; or
- **event-driven task** — it just waits for events to occur.

Event-driven tasks usually deal with extraordinary situations or with fast I/O data. The implementations of numeric control algorithm depending on sampling period need periodic tasks and continuous tasks are suitable for the rest of operations.

IOH-programs rarely access I/O hardware directly. Such manipulation are usually too slow, require special processing and permissions, so they are also either not recommended or reserved only for special situations, for instance the sampling of analog data for a discrete control algorithm.

[1] This research is partially supported by the Rockwell Automation Services, Kolín, Czech Republic

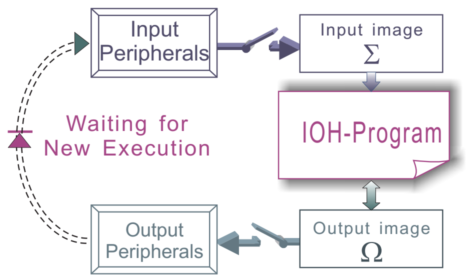

Fig. 1. IOH-Program in synchr. environment

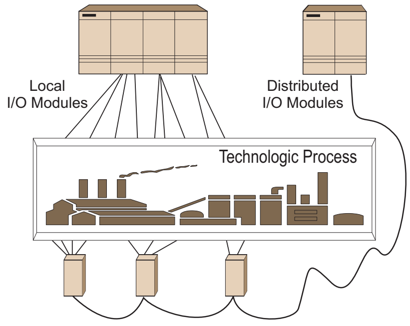

Fig. 2. Local and distributed I/O modules

Thus, IOH-programs ordinary run embedded in some proper hardware or software environment, as depicted in Figure 1. Such environments are also offered by robust PLCs (programmable logical controllers), which firmware synchronizes the evaluations of IOH-programs with I/O data by repeating three consequent steps:

Input scan: Hardware inputs are polled, or sampled respectively, and their values are stored into inner memory Σ, called *Input image*;

Program scan: user's IOH-program is executed once. It calculates new outputs and writes them into inner memory Ω, called *Output image*; and

Output scan: after termination of the program, the values in output image Ω are copied into corresponding peripherals.

These steps correspond to a regulator with cyclic sampling of I/O data. On classical computer, if an IOH-program is scheduled as a continuous task then it can be programmed by endless loop, for instance in Pascal:

```
repeat
  Read_Inputs(Σ);
  Execute_Program(Σ,Ω);
  Write_Outputs(Ω);
until false;
```

IOH-programs scheduled as periodic and event-driven tasks can be programmed as event handlers of timers or I/O peripherals.

However, the synchronous updating of I/O has many advantages, its major drawback is polling I/O data in one-stroke, which increases I/O scan times inadequately, especially, when I/O peripherals are connected by control area networks (CANs), as depicted in Figure 2.

Distributed I/O modules offer many indisputable advantages, of which we mention only significant reduction of the length of all necessary wires, but there are also some drawbacks. In the contrast to local I/O buses, which speed is practically determined only by their electrical parameters, the transfer rates of networks are limited by much more factors. Polling all I/O data in one stoke, as it is perfomed by synchronous updating, concentrates all data transfers into short intervals and easily causes overloading of networks.

An asynchronous updating of inputs and outputs allows much better optimization of network traffic. In this case, input Σ and output Ω images are usually converted into sets of *tags*.

Tags correspond to typed variables of classical programming languages, but with the addition of possible bounding their values to external sources, either local or remote, so their values can come from input modules (*input tags*) or from other computers, (*consumed tags*). [2]

If an input source is connected through local I/O bus, then its corresponding destination tag can be updated as fast as the communication devices can process the information, otherwise its update time is specified by an entered requested packet interval (RPI) which defines the required maximum amount of time between the updates, if the update is periodic, or a maximal delay between a change and sending new value, if its updating is bound to changes of a data source.

The opposite roles are played by *output* tags, which values are written into output modules, or *produced* tags, which are send to other connected computers.

One output or produced tag can be transmitted into more destinations, but any input or consumed tag is always updated at most from one source. Thus the value of any tag

(1) is always transmitted only unidirectionally, i.e. from its source to all possible destinations, if any; and
(2) can be also updated:
    - in shorter time than requested RPI and
    - during the execution of some instructions.

[2] Typical representatives of tag-based systems are PLCs of ControlLogix family manufactured by Allen Bradley, Rockwell Automation.

An IOH-program, which is scheduled as a continuous task with asynchronous I/O updating, can be programmed on a classical computer with the aid of two or more threads:

The main thread:	*Other threads:*
repeat	**repeat**
`IOH_Program(`Σ,Ω`);`	`IOManager(`Σ,Ω`);`
until `false;`	**until** `Terminated;`

where *Terminated* stands for the request to terminate the IOManager thread. Periodic and event-driven tasks can be again programmed as event handlers of timers or I/O peripherals.

The I/O update manager, which performs asynchronous updating, runs as a parallel program, which arises a possibility of dataraces. In parallel programming, dataraces are usually excluded by synchronization tools, as locks or mutex objects for protecting critical sections, but this method does not generally assure regular updates of all I/O data.

For that reason, utilizing synchronization tools need not be allowed necessarily in all tag-based environments. [3] But excluding mutual synchronization between I/O update manager and a IOH-program also means excluding many methods known from parallel programs. Therefore we must search another solution.

2. MODEL OF UPDATING ENVIRONMENT

In this paper, we aim to adaptation of a given IOH-program, which was written for an environment with synchronous I/O updating, to a new environment with asynchronous I/O updating.

First, we create model of the both environments to analyze their properties. We will distinguish between them by utilizing *TAG-IOH* for an environment with asynchronous I/O updating into tags and *S-IOH* for a synchronous environment depicted in Figure 3.

To isolate S-IOH and TAG-IOH differences from other difficult questions of parallel programming, we will assume:

(1) an IOH-program will be scheduled as a continuous task,
(2) its execution time is always limited by t_{ep} constant, and

[3] For example, the instruction set of tag based PLC family ControlLogix allows only few instructions for predefined interlocked operations, i.e., only during their execution, it is granted that their operands will not be updated. These instructions do not involve other operations, with the exception of adding possible random time delays to them, so they do not allow synchronizing between program and I/O update manager.

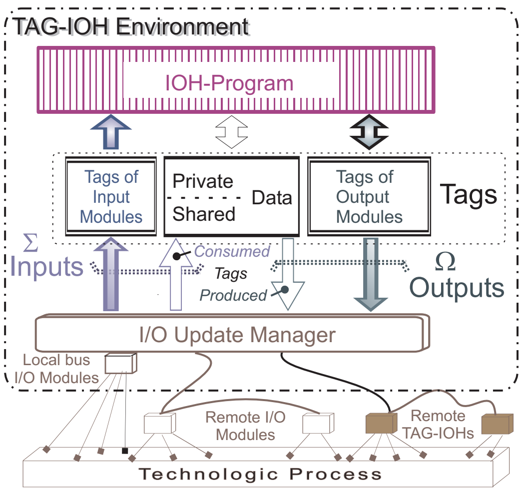

Fig. 3. Σ and Ω in TAG-IOH environment

(3) the members of Σ and Ω sets, which are accessed by the program, do not share mutually their memory locations, i.e., each member is mapped to its own storage.

Sets Σ and Ω are specified for S-IOH environment by Figure 1. For TAG-IOH environment, we create Σ as the set of all input and consumed tags, as depicted in Figure 3.

Similarly, Ω will contain all output tags and all produced tags. Finally, all tags or variables accessed by the program are included into V set of variables.

The third assumption in the list above assures that Σ, V, and Ω are three disjoint sets, which we utilize in the following definition.

Definition 1. (IOH-program). Let be Σ, Ω, and V three disjoint sets of variables. We denote by $\mathcal{P}$ any program that satisfies the following:

- it always terminates and its execution time is less than a given constant T,
- it utilizes Σ as its inputs, Ω as its outputs, V as its internal variables, and
- it accesses nothing outside *storage* $S = V \cup \Sigma \cup \Omega$.

We denote by $\mathbb{P}\langle\Sigma, V, \Omega, T\rangle$ the set of all such IOH-programs, i.e., $\mathcal{P} \in \mathbb{P}\langle\Sigma, V, \Omega, T\rangle$.

Our the model of S-IOH environment consists of one thread with endless loop, as shown on Figure 4. Maximum execute times of the input updates, the program runs, and output updates are given by constants t_{ei}, t_{ep}, and t_{eo}.

Notice that the program has also write access into Σ input image, when it is embedded in S-IOH environment. It is sometimes suitable for quick

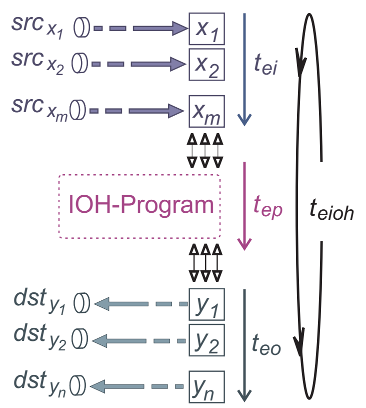

Fig. 4. Model of S-IOH environment

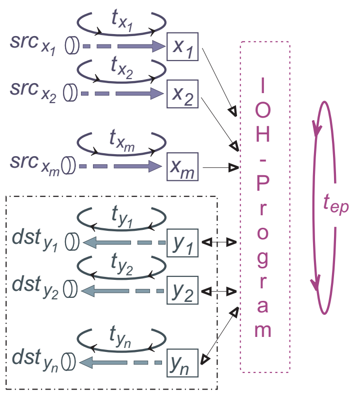

Fig. 5. Model of TAG-IOH environment

readdressing of inputs without many changes in the program, if required.

TAG-IOH model requires specifying RPI times in addition to storage S. In the following paragraphs, we denote a common index set by I, i.e.,

$$I \stackrel{df}{=} \{1, 2, \ldots |I|\} \tag{1}$$

Let all $x_i \in \Sigma$, $i \in I$, $|\Sigma| = |I|$, be periodically updated from some src_{x_i} external sources in randoms intervals, whose lengths t of time have unknown probability distributions, so $\mathcal{P}$ program may make only one assumption that t satisfies $t_{x_i} \geq t > 0$ where t_{x_i} stands for some given RPI time of x_i tag.

Similarly, we assume that the values of all $y_j \in \Omega$, $j \in I$, $|I| = |\Omega|$ are periodically copied into dst_{y_j} external destinations in some random intervals t satisfying $t_{y_j} \geq t > 0$, where t_{y_i} are given RPI times.

In this case, $\mathcal{P}$ program is executed on a TAG-IOH by the way that corresponds to $m+n+1$ threads (see Figure 5) where $m = |\Sigma|$ and $n = |\Omega|$. The thread of user's IOH program performs endless loop with maximum length of program scan time equals to t_{ep}.

Remark 2. We utilized so $m + n$ thread only for simplification of the model. In reality, too many active threads decrease performance of operating systems. Therefore, the practical solution of I/O update manager could, for example, consists of one thread program that dispatches events, as incoming network packets, to sub-handlers. But properties of this solution will be similar to our thread model.

To optimize I/O polling, TAG-IOH environment can allow disabling automatic output updates, which is emphasized by the dashed rectangle in Figure 2. In such case, the program must request the updates for each group of output tags after finishing their evaluations.

In contrast, all Σ input tags are always updated independently to the execution of IOH program and, therefore, Σ *are read only data in TAG-IOH* because the program cannot reliably store any temporary values in them.

We may consider all possible write accesses into Σ tags as program errors, which we employ in the adaptation.

3. I/O LATENCIES

There are two important differences between S-IOH and TAG-IOH environments caused by distinct I/O updates - I/O latencies and dataraces. In this section, we consider the latencies.

Definition 3. Let $x \in \Sigma$ be an input. Suppose the existence of two functions that return last times before a given t time, when

- $\text{tch}(x, t)$ - data source of x has changed its value, and
- $\text{tup}(x, t)$ - stored value of x was updated.

Input latency of x in time t is defined by

$$\text{til}(x, t) \stackrel{df}{=} t - \text{tch}(x, \text{tup}(x, t)) \tag{2}$$

Definition 4. Let $x, y \in \Sigma$ be any two inputs. their mutual input latency in a given time t by $|\text{til}(x, t) - \text{til}(y, t)|$.

In other words, an input latency specifies a time delay at time t between actual value of an input and its value read from Σ in time t, as depicted in Figure 6. Mutual input latencies specify time delay between samples stored in Σ.

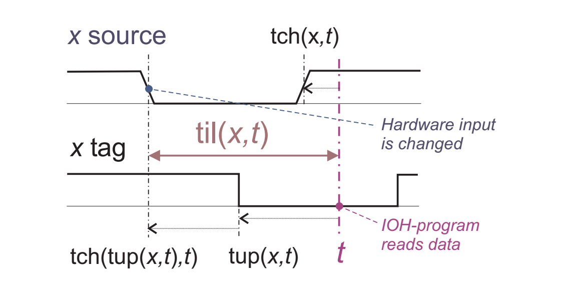

Fig. 6. Input latency

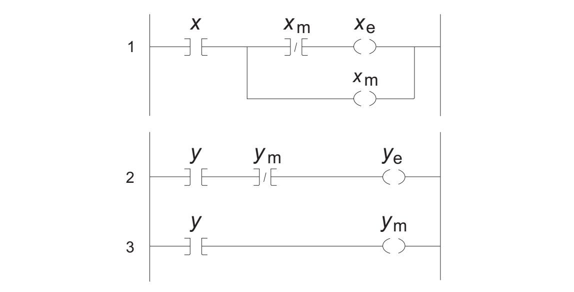

Fig. 7. Edge detection

	Input latencies	
	One Input	*Mutual*
S-IOH	$\leq t_{\mathrm{ei}}+t_{\mathrm{ep}}$	$\leq t_{\mathrm{ei}}$
TAG-IOH$_{(-B)}$	$<RPI$	$\leq RPI+t_{\mathrm{ep}}$
TAG-IOH$_{(+B)}$	$\leq RPI+t_{\mathrm{ep}}$	$<RPI$

(-B) normal, (+B) with I/O bufering

Table 1. Input latencies of models

Input latencies are summarized in Table 1. For S-IOH model, maximal input latency equals to $t_{\mathrm{ei}}+t_{\mathrm{ep}}$, i.e., the duration of input sampling plus the execution time of IOH-program. Its mutual input latency depends only on t_{ei} because all inputs are read during one input scan.

When discussing TAG-IOM model we need to distinguish if IOH-program emulates synchronous environment, which is a natural solution for preventing problems with asynchronous sampling — the input values are read into an array before running IOH-program and the array values are utilized instead of actual input tags. Similarly, outputs are written in another array and copied to output tags at the end of the execution. We will call this approach as *I/O buffering* .

Input latencies of TAG-IOH model without an I/O buffering are determined only by actual RPI times of x_i inputs in the question (t_{x_i} in the model). Mutual I/O latencies are given as the maximum of RPI times of inputs involved, to which we must add the execution time of an IOH-program, at most t_{ep}, if the first variable is read at beginning and the second before the end of the program. The similar conclusion can also be derived for Ω outputs and their latencies.

If an I/O buffering is employed in TAG-IOH model then input latencies are increased by the execution time because values stored in a buffer are not updated. On the other hand, mutual input latencies are frozen to the moment of buffering and do not depend on the execution time.

However, mutual latencies are non-zero, neither for S-IOH nor for TAG-IOH. They are not a characteristic behavior of a TAG-IOH — it may only emphasize them.

4. DATARACES IN TAG-IOH

Dataraces, known from parallel programs, are discussed in many papers, for instance in Choi et al. (2002), and usually defined as two memory accesses which satisfy four *datarace conditions*:

(1) the two accesses are to the same memory locations and at least one of the accesses is a write operation;
(2) the two accesses are executed by different threads;
(3) the two accesses are not guarded by a common synchronization object (lock); and
(4) there is no execution ordering enforced between two accesses, for example by thread `start` or `join` operations.

All inputs Σ and outputs Ω of TAG-IOH satisfy datarace conditions. We pick up only such accesses to them, which could result in an erroneous behavior on TAG-IOH, but not on S-IOH, which means inputs Σ. They only are changed by updating processes of TAG-IOH model, and at any moment, unlike S-IOH. Thus, some codes will not work correctly with TAG-IOHs.

Possible dataraces in outputs Ω export problems to the external units, about which we made no assumption. Therefore, we will not study Ω outputs here.

We demonstrate this fact on two examples of well known rising edge detection in inputs x and y, see Figure 7. The result x_e (rung 1) of the edge detection will be 1 for one program scan, in which x input has just changed to 1, otherwise $x_e=0$. The result y_e behaves similarly.

The ladder diagram can be converted into the following codes:

$$\begin{array}{lll} x_s := x & x_e := x_s \wedge \neg x_m; & x_m := x_s; \\ & y_e := y \wedge \neg y_m; & y_m := y; \end{array}$$

The first line copies x variable into temporary x_s, stored into an evaluation stack, then x_e rising edge is evaluated with the aid of x_s. The second line describes mathematically identical operation, but performed without temporary storage for y input.

To analyze the program, we define its all traces.

Definition 5. Let $\mathcal{P} \in \mathbb{P}\langle\Sigma, V, \Omega, T\rangle$ be a IOH-program. We define *set of $\mathcal{P}$ traces* as a subset $\mathrm{trace}(\mathcal{P}) \subset \mathbb{P}\langle\Sigma, V, \Omega, T\rangle$ that includes all possible programs, which code consists of instructions executed during one execution of $\mathcal{P}$ program. *Set of traces* $\mathrm{trace}(\mathcal{P})$ *is deterministic* if it holds for any $\mathcal{P}_{\mathrm{tr}} \in \mathrm{trace}(\mathcal{P})$ that $\{\mathcal{P}_{\mathrm{tr}}\} = \mathrm{trace}(\mathcal{P}_{\mathrm{tr}})$.

Any deterministic set of traces contains only IOH-programs with single unchangeable streams of instructions, but it does not still assure its usability, because some special IOH-programs can have sets of traces with huge cardinalities. On the other hand, these programs will be probably excluded from analyses for their complexity in any case.

To express possible change of the values of input tags x and y, we mark the accesses to them by superscripts to express the information about instant of time, in which the value of an input tag was sampled.

Definition 6. Given $x \in \Sigma$ input tag, $\mathcal{P} \in \mathbb{P}\langle\Sigma, V, \Omega, T\rangle$ program and that have deterministic set of traces. If x value is read in some discrete time t in a $\mathcal{P}_{\mathrm{tr}} \in \mathrm{trace}(\mathcal{P})$, then we denote such read *instant of* x tag by $x^{\langle t\rangle}$.

The set of traces of the rising edge detection contains one IOH-program, the original program itself. Utilizing integer times, we mark the access to the value of x tag by $x^{\langle 1\rangle}$ and two accesses to the value of y tag on the rung 2 and 3 by $y^{\langle 2\rangle}$ and $y^{\langle 3\rangle}$. We obtain the program:

$$\begin{array}{lll} x_s := x^{\langle 1\rangle} & x_e := x_s \wedge \neg x_m; & x_m := x_s; \\ & y_e := y^{\langle 2\rangle} \wedge \neg y_m; & y_m := y^{\langle 3\rangle}; \end{array}$$

The tag x_e still depends only on single time instant of x_s, but y_e depends on $y^{\langle 2\rangle}$ and y_m where is stored $y^{\langle 3\rangle}$. If $y^{\langle 2\rangle} = 0$ and $y^{\langle 3\rangle} = 1$ in some program execution accidentally due to updating the value of y tag, then the rising edge of y will not be detected.

Lemma 7. Let $\mathcal{P} \in \mathbb{P}\langle\Sigma, V, \Omega, T\rangle$ be arbitrary IOH-program. Given $x \in \Sigma$ input. If it holds for all $\mathcal{P}_{\mathrm{tr}} \in \mathrm{trace}(\mathcal{P})$ that $\mathcal{P}_{\mathrm{tr}}$ code contains only one instant of x, that $\mathcal{P}$ has no dataraces in x.

If x variable is read only once then no datarace can exist. Lemma gives simple method, but their application is limited to trivial cases. For example, it does not exclude the presence of dataraces in $\mathcal{P} \in \mathbb{P}\langle\{x\}, \{i_1, i_2\}, \emptyset, T\rangle$ program, which is evidently datarace free: $i_1 := x^{\langle 1\rangle};\quad i_2 := x^{\langle 2\rangle};$

Proposition 8. Let $\mathcal{P} \in \mathbb{P}\langle\Sigma, V, \Omega, T\rangle$ be arbitrary IOH-program. Given $x \in \Sigma$ input. If it holds for all $\mathcal{P}_{\mathrm{tr}} \in \mathrm{trace}(\mathcal{P})$ that all values of all $V \cup \Sigma$ variables were derived only from one instant of x, that $\mathcal{P}$ has no dataraces in x.

The structure of the proof is straightforward. If each value variable $y \in V \cup \Sigma$ depends only on one instant, a possible change of x value will have no influence.

The proposition does not contains 'iff' clause because it can announce false alarms, for example, in $y = 1 \wedge x \wedge x$, but it concerns mainly to cases when one or more instants of variable are redundant and have no influence on a result, which are rare situations.

Proposition 8 offers possibility applying dataflow analysis methods that were developed for optimizing compilers. Many publications studies this problem, for instance Sathyanathan (2001) or Zheng (2000).

In the following subsection, we present a method applicable to some subset of IOH-programs. Even if it does not process sufficiently special operation, as pointer of arrays, it has simple implementation. We use it as an overview of the problem.

4.1 Testing dataraces

We will analyze the dependencies of variables on input Σ. First we define sets for storing this information.

Definition 9. (ψ-pair). Let S be arbitrary non-empty set of variables. We define *ψ-pair on S* as $\psi\langle x, X\rangle$, where $x \in S$ and $X \subseteq S$, and two operators:

$$\mathrm{dom}(\psi\langle x, X\rangle) \stackrel{df}{=} X$$
$$\mathrm{co}(\psi\langle x, X\rangle) \stackrel{df}{=} x$$

where $\mathrm{dom}(\psi\langle x, X\rangle)$ and $\mathrm{co}(\psi\langle x, X\rangle)$ stand for *domain* and *codomain* of $\psi\langle x, X\rangle$. We denote a set of all ψ-pairs on given S by $(S)^{\psi *}$.

Definition 10. Any subset $\tilde{X} \subseteq (S)^{\psi *}$ satisfying that $\mathrm{co}(\psi\langle x, V\rangle) = \mathrm{co}(\psi\langle x, V\rangle)$ implies $i = j$ for all $\psi\langle x_i, V\rangle, \psi\langle x_j, V\rangle \in \tilde{X}$, is called a *$\Psi$-set on S* . We denote the set of Ψ-sets for S variables by $\Psi(S)$, i.e., $\tilde{X} \in \Psi(S)$.

In any Ψ-set $\tilde{X} \in \Psi(S)$, one ψ-pair exists at most for each variable $x_i \in S$ with the codomain equal to x_i. We also define codomains of Ψ-sets as the sets of codomains of all its ψ-pairs.

Definition 11. Given a Ψ-set $\tilde{X} \in \Psi(S)$. We define its codomain as:

$$\mathrm{co}(\tilde{X}) \stackrel{df}{=} \left\{x_i \mid \psi\,\langle x_i, V\rangle \in \tilde{X}\right\} \tag{3}$$

Definition 12. A composition $\tilde{Z} = \tilde{X} \circledcirc \tilde{Y}$ of two given $\tilde{X}, \tilde{Y} \in \Psi(S)$ is $\tilde{Z} \in \Psi(S)$, $|\tilde{Z}| = |\tilde{X}|$, containing ψ-pairs $\psi\,\langle z_i, Z_i\rangle \in \tilde{Z}$. These ψ-pairs are constructed by the following algorithm:

First step:

for all $\psi\,\langle x_i, X_i\rangle \in \tilde{X}$, $i \in I$, $|I| = |\tilde{X}| = |\tilde{Z}|$
do begin:
  $\psi\,\langle z_i, Z_i\rangle := \psi\,\langle x_i, X_i\rangle$
    for all $\psi\,\langle y_j, Y_j\rangle \in \tilde{Y}$ do begin:
      if $y_j \in X_i$ then $Z_i := (Z_i - \{y_j\}) \cup Y_j$
    end
end

Second step:

for all $\psi\,\langle y_i, Y_i\rangle \in \tilde{Y}$, $i \in I$, $|I| = |\tilde{Y}|$
do begin:
  if $y_i \notin \mathrm{co}(\tilde{Z})$ then $\tilde{Z} := \tilde{Z} \cup \{\psi\,\langle y_i, Y_i\rangle\}$

In words, the first step tests if the codomain of a ψ-pair from $\tilde{Y}$ is in domain of any ψ-pair from $\tilde{X}$. If it is satisfied, ψ-pair from $\tilde{Y}$ replaces by its domain the variable in the domain of ψ-pair in $\tilde{X}$.

The second step adds to the result all ψ-pair in $\tilde{Y}$, which codomains are not in the codomain of the result.

Proposition 13. The composition $\circledcirc$ is associative on $\Psi(S)$.

Proof: Ψ-sets are derived by simplifying transfer set theory (see Šusta (2003)). If we assume the existence of some abstract $\odot$ binary operation that satisfies idempotency ($x \odot x = x$) and commutativity ($x \odot y = y \odot x$) laws for any two arbitrary tags x, y, then we can create mapping of ψ-pairs into transfer sets. For instance, a ψ-pair $\langle x, \{x, y, z\}\rangle$ is mapped into $x := x \odot y \odot z$ assignment, which has direct conversion to $\{\hat{x}[\![x \odot y \odot z]\!]\}$ transfer set. The associativity was already proved for transfer sets.

The analogy between ψ-pair and expression allows utilizing them for testing variable dependency. We create *instantized input set* $\Sigma^{\langle t\rangle}$ of all instants of input tags in a given program trace $\mathcal{P}_{\mathrm{tr}} \in \mathrm{trace}(\mathcal{P})$.

We describe dependencies in the instructions of some trace by *psi*-pairs, which we consecutively compose by $\circledcirc$ operation to one Ψ-set D. Finally, we test dataraces by the following algorithm that composes mutually ψ-pairs to find out all dependencies.

Algorithm 1. Testing dataraces:

Input: Given a non-empty instantized storage $S^{\langle t\rangle} = \Sigma^{\langle t\rangle} \cup V \cup \Omega$ and Ψ set $\tilde{D} \in \Psi(S^{\langle t\rangle})$.
Initialization: Loop index $i = 0$.
Step 1: Utilizing $\langle x_i, X_i\rangle \in \tilde{D}$ we perform for all $j \in I, |I| = |D|, j \neq i$: "If $\langle x_j, X_j\rangle \in \tilde{D}$ satisfies that $x_j \in X_i$, then $X_i := (X_i - \{x_j\}) \cup X_j$."
Step 2: If $i < |D|$ then we increment i and repeat Step 1, otherwise we proceed to the final test.
Final test: If any $\mathrm{dom}(\langle x_i, X_i\rangle) \in \tilde{D}$ contains two different instants of one input, then x has possible datarace.

The algorithm requires the maximum amount of memory $|V \cup \Omega| * |S^{\langle t\rangle}|$, contains finite loop and always terminates at most after $|D| * (|D| - 1)$ steps. [4] Each loop cycle adds dependencies of one variable into all domains of such ψ-pairs, in which domains it is presents, therefore, if any tag depends on more instants of some input, then they must appear at least in one domain set of the result.

5. CONCLUSION — ADAPTATION OF IOH-PROGRAM

The adaptation of a program can be suitable mostly in two cases to short the expensive commissioning phase:

- Peripherals have been redeveloped with the aid of control area networks (CANs) and we want to adapt major parts of our old reliable program.
- New program was written by technologists accustomed to programming in synchronous I/O update environment, so it must adapted to asynchronous environment.

First, we utilize the results presented in Table 1 for TAG-IOH and decide if we need to minimize mutual latencies for some signals in subset $\Sigma \cup \Omega$. For such I/O groups, we add their I/O buffering into the program. It is ordinary required for all numeric control algorithms or other parts which functionality depend on a proper sampling.

Finally, we consider dataraces. The I/O buffering exclude them. If we have buffered all Σ inputs, then the program is surely datarace free. Otherwise, if I/O buffering concerns only some subset of Σ, we test all unbuffered inputs by either Proposition 8 or Lemma 7.

[4] The reduction of steps to half is possible, but it leads to less comprehensible form.

No.	Size [kB]	Inputs	Possible dataraces	
			Lemma 7	**Prop. 8**
1	3.2	5	5	5
2	3.5	4	4	3
3	6.7	4	4	3
4	13.4	94	63	54
5	28.2	132	102	88
6	143.8	419	25	2

Table 2. Tested PLC 5 Programs

We can try simple Algorithm 1 in Subsection 4.1. If it fails to give results for a $x \in \Sigma$, we apply I/O buffering to x, or we try any more exact and complex dataflow analysis.

5.1 Experimental Results

Algorithm 1 was tested on 6 program fragments extracted from different PLC programs that were written by several programmers for various industrial technologies. All fragments have only one possible trace (see Definition 5), which is common feature of many PLC programs.

Unfortunately, we have tested only PLC 5 programs because the import modules for another PLCs have not been finished yet, and we could analyze only fragments, since every PLC program contains some PLC dependent parts, for example special initializations of I/O modules.

The results are presented in Table 2. They are listed in size order measured in bytes occupied by programs in PLC internal memory. The table shows the number of their inputs, which must be tested, and possible input dataraces detected applying Lemma 7 and Proposition 8.

Lemma 7 does not require any additional equipment — we directly utilized the cross reference list in RSLogix 5 programming environment for PLC 5 processors. Testing dataraces according Proposition 8 was performed by an external program that processed PLC 5 programs exported into text files.

Proposition 8 gives more exact results. The biggest program (number 6) was nearly datarace-free due to copying many inputs into auxiliary variables to allow fast readdressing of I/O signals.

Much smaller programs 4 and 5 have many possible dataraces because they prioritized reducing auxiliary variables and preferred repeating inputs conditions to simplify troubleshooting of technologies.

Therefore, no statistically significant relation can be deduced. The number of inputs suspected for possible dataraces in a PLC program too depends on controlled technologies, employed programming styles, and additional requirements. On the other hand, the application of detailed tests according to Proposition 8 could save time in special cases.

6. RELATED WORKS

Dataraces are intensively studied and many papers deal with them, for example Choi et al. (2002); Ramanujam and Mathew (1994) cited in the previous sections. Unfortunately, all approaches that we have found assumed the knowledge of the source codes of all analyzed treads. No available publication studied a case similar to TAG-IOH, when some threads have random behavior and no synchronizations are available.

The optimization of compilers is the main domain of dataflow analysis used for tracking variable dependences. We already mentioned Sathyanathan (2001) and Zheng (2000), in which are long lists of publication dealing with this topic. The book Nielson et al. (1999) presents good overview of many methods.

Our simple method for testing variable dependences, presented in Subsection 4.1, utilizes the results of the transfer sets published in Šusta (2004) or Šusta (2003).

REFERENCES

J.-D. Choi, K. Lee, A. Loginov, R. O'Callahan, V. Sarkar, and M. Sridharan. Efficient and precise datarace detection for multithreaded object-oriented prog. In *ACM SIGPLAN 2002 Conference on Programming Language Design and Implementation (PLDI), Berlin, Germany*, June 2002.

Flemming Nielson, Hanne Riis Nielson, and Chris Hankin. *Principles of Program Analysis*. Springer-Verlag, 1999. ISBN 3–540–65410–0.

J. Ramanujam and A. Mathew. Analysis of event synchronization in parallel programs. In *Languages and Compilers for Parallel Computing*, pages 300–315, 1994.

Patrick W. Sathyanathan. *Interprocedural Dataflow Analysis - Alias Analysis*. PhD thesis, Stanford University, Computer Systems Laboratory, June 2001.

Richard Šusta. *Verification of PLC Programs*. PhD thesis, CTU-FEE Prague, May 2003. avail. at http://dce.felk.cvut.cz/susta/.

Richard Šusta. Low cost simulation of PLC programs. In *7th IFAC Symposium on Cost Oriented Automation COA 2004, Gatineau (Québec) Canada*, pages 219–224. Université du Québec en Outaouais, 2004.

B. Zheng. *Integrating Scalar Analyses and Optimizations in a Parallelizing and Optimizing*. PhD thesis, Dept. of Computer Science and Engineering, University of Minnesota, 2000.