HAL
open science

CoMET: Compressing Microcontroller Execution Traces to Assist System Understanding

Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, Lydie Du Bousquet

To cite this version:

Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, Lydie Du Bousquet. CoMET: Compressing Microcontroller Execution Traces to Assist System Understanding. 2013. hal-00857299

HAL Id: hal-00857299

https://hal.science/hal-00857299

Submitted on 10 Sep 2013

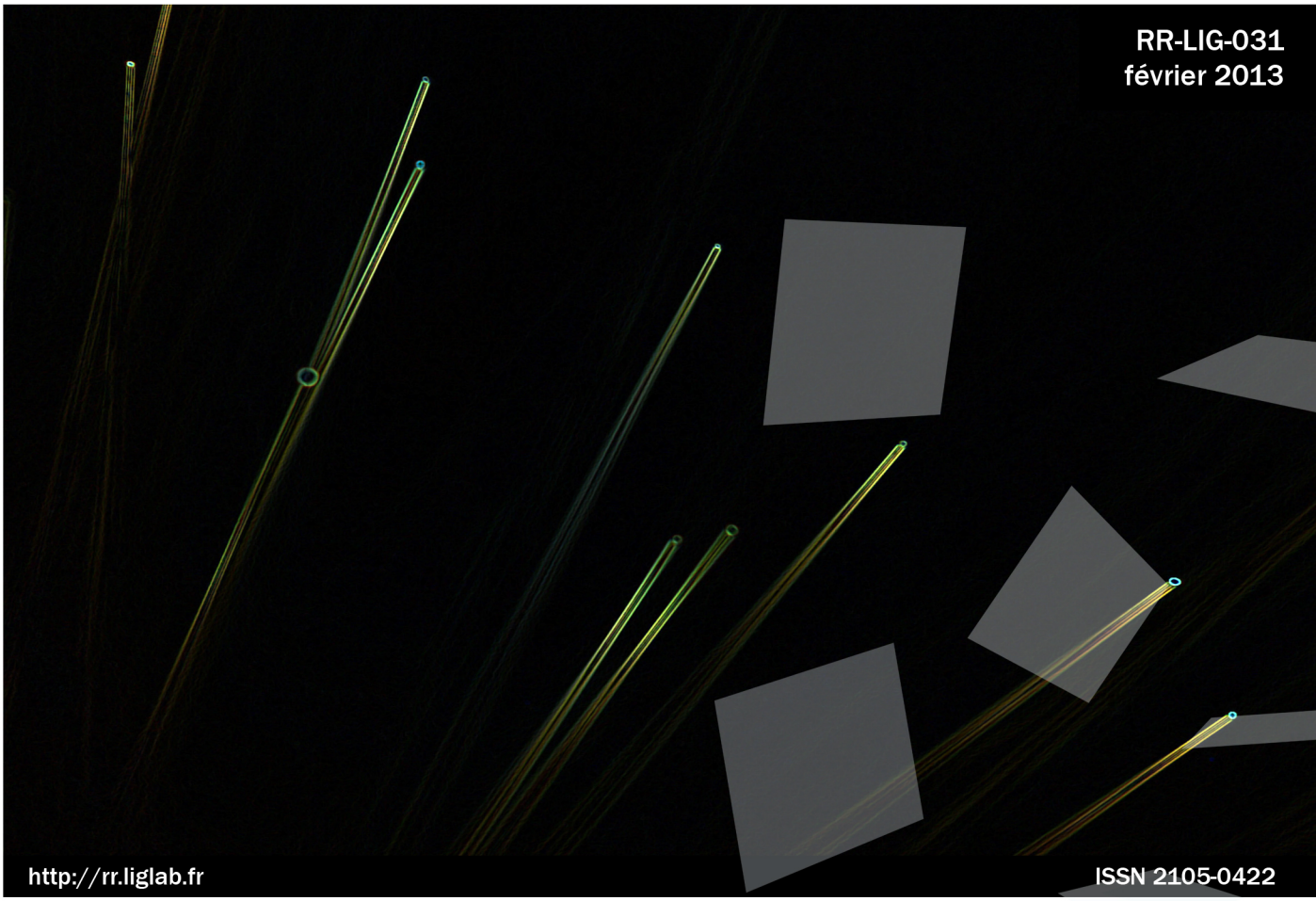
HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Les rapports de recherche du LIG

CoMET: Compressing Microcontroller Execution Traces to Assist System Understanding

Azzeddine AMIAR, PhD student, LIG, University of Grenoble (UJF), France
Mickaël DELAHAYE, Post Doc, LIG, University of Grenoble (UJF), France
Yliès FALCONE, Associate Professor, LIG, University of Grenoble (UJF), France
Lydie du BOUSQUET, Associate Professor, LIG, University of Grenoble (UJF), France



L I G

CoMET: Compressing Microcontroller Execution Traces to Assist System Understanding

Azzeddine Amiar, Mickaël Delahaye, Yliès Falcone, Lydie du Bousquet
Laboratoire d'Informatique de Grenoble (LIG), UMR 5217
UJF-Grenoble 1
BP 72, 38402 St Martin d'Hères, France
Email: FirstName.LastName@imag.fr

***Abstract*—Recent technology advances have made possible the retrieval of execution traces on microcontrollers. However, even after a short execution time of the embedded program, the collected execution trace contains a huge amount of data. This is due to the cyclic nature of embedded programs. The huge amount of data makes extremely difficult and time-consuming the understanding of the program behavior. Software engineers need a way to get a quick understanding of execution traces. In this paper, we present an approach based on an improvement of the Sequitur algorithm to compress large execution traces of microcontrollers. By leveraging both cycles and repetitions present in such execution traces, our approach offers a compact and accurate compression of execution traces. This compression may be used by software engineers to understand the behavior of the system, for instance, identifying cycles that appears most often in the trace or comparing different cycles. Our evaluations give two major results. On one hand our approach gives high compression rate on microcontroller execution traces. On the other hand software engineers mostly agree that generated outputs (compressions) may help reviewing and understanding execution traces.**

***Index Terms*—trace comprehension; cyclic program; execution trace; trace compression; Sequitur**

I. Introduction

Most of the automated systems contain microcontrollers, from car engines to domestic appliances. Although there are many development environments for embedded applications, there are few tools dedicated to their analysis. A manual analysis is a tedious and time consuming task [1]. In our work, we aim to make it easier for software engineers to understand a microcontroller behavior by exploring execution traces.

In general, software engineers have to either analyze electrical signals generated from components, or try to reproduce microcontroller behaviors using simulation [2], [3]. However, the manual process and the lack of information about external devices that are connected to a microcontroller, does not allow software engineers to reproduce the real behavior of the embedded application. In the past few years, a technique has been proposed to collect basic execution traces using a specific probe [4]. Recording microcontroller execution enables software engineers to collect information about the system behavior as an execution trace without input/output data [5], [6]. However, collected execution traces are often very large [7], [8] and incomprehensible. The huge amount of data is mainly due to the real-time and *cyclic* nature of microcontroller applications. Consequently, even for the most experienced engineer, a manual analysis of such traces takes significant effort and time. Since such applications are cyclic, a limited number of different instructions are executed numerous times.

In this paper, we aim to help software engineers to look over an execution trace faster, and to make more informed decisions on which part(s) of the trace they need to analyze in detail. To facilitate the comprehension of execution traces, we propose a compression approach based on a grammar generation. We formalize and extend the Sequitur algorithm [9], which, given an input trace, produces a grammar by leveraging *regularities* frequently found in the input trace. The output grammar is an accurate but compact representation of the input trace. Our algorithms are implemented in a tool named J-Cyclitur, which is written in the Java programming language and is freely available at http://io32.forge.imag.fr. The J-Cyclitur enables us to compress several real traces coming from embedded applications (gathered using the ETM probe) and network traffic simulations.

The rest of the paper is organized as follows. The next section exposes the context of applications that are embedded on microcontrollers. Section III proposes a formalization of the Sequitur algorithm, this formalization is based on several informal descriptions found in the literature. In Section IV, we present the Cyclitur algorithm, which is an extension of the Sequitur algorithm in the context of cyclic embedded applications. Our algorithm is empirically validated in Section V. Section VI discusses related work, and, Section VII draws some conclusions and perspectives.

II. Microcontroller application context

A microcontroller is an integrated circuit that consists of all elements of a microprocessor-based structure [10], such as a microprocessor (CPU), data memory (RAM, EEPROM), program memory (ROM, OTPROM, UVPROM or EEPROM), and input/output. Thanks to the affordable prices of microcontrollers, the development of embedded applications is now within reach of non-specialists. For example, a 32-bit microcontroller, with 128Kb of SRAM and 1Mb FLASH, costs about 7 USD, and can meet numerous use cases.

In recent past, debugger engineers used oscilloscopes to analyze embedded applications by interpreting electric sig-

nals. Thanks to the arrival of new microcontrollers, the situation is changing. For instance, the microcontroller ARM Cortex-Mx [11] includes a part dedicated to trace collection, called ETM cell (Embedded Trace Macrocell) [4]. Using a probe [12], it is possible to obtain a trace in the form of a series of the so-called *instruction-sequence breaks* that reflects the execution. A sequence break occurs during the execution of a program when next instruction does not directly follow the current one, it is a *jump*.

A typical example of C program for microcontrollers is shown in Listing 1. If the condition at line 31 evaluates to false, the next executed instruction is not the instruction at line 32 but the instruction at line 35. The destination addresses of jumps are recorded by the ETM cell. The probe enables the user to extract the execution trace as a set of sequence breaks ordered by their occurrence in time. The execution trace obtained for the particular program shown in Listing 1 contains the following lines: 2, 7, 10, 13, 16, 19, 22, 25, 31, 35 and 39, where each line corresponds to a sequence break.

As can be seen in Listing 1, such program contains an infinite main loop (starting at line 2), to scan actively the inputs of the microcontroller and calculate the outputs. The huge amount of data contained in the trace (even after a short time execution) is mainly due to this infinite loop.

In this paper, we address the first two steps of the debug process, namely the comprehension and the analysis of execution traces. Indeed, we propose a method to compress the execution traces [13], [7], [14] of microcontroller embedded programs. We exploit the presence of the infinite main active loop to generate a compression. We call *loop header* the instruction that defines this main active loop. The compression is based on the Sequitur algorithm, which is discussed in the following section.

Listing 1. Example of C embedded application code

```
int main(void){                                                          1
  while(1){                                                              2
    static JOY_State_TypeDef JoyState = JOY_NONE;                        3
    static TS_STATE* TS_State;                                           4
    JoyState = IOE_JoyStickGetState();                                   5
    switch (JoyState) {                                                  6
      case JOY_NONE:                                                     7
        LCD_DisplayStringLine(Line5, "JOY: ---- ");                      8
        break;                                                           9
      case JOY_UP:                                                       10
        LCD_DisplayStringLine(Line5, "JOY: UP ");                        11
        break;                                                           12
      case JOY_DOWN:                                                     13
        LCD_DisplayStringLine(Line5, "JOY: DOWN ");                      14
        break;                                                           15
      case JOY_LEFT:                                                     16
        LCD_DisplayStringLine(Line5, "JOY: LEFT ");                      17
        break;                                                           18
      case JOY_RIGHT:                                                    19
        LCD_DisplayStringLine(Line5, "JOY: RIGHT ");                     20
        break;                                                           21
      case JOY_CENTER:                                                   22
        LCD_DisplayStringLine(Line5, "JOY: CENTER ");                    23
        break;                                                           24
      default:                                                           25
        LCD_DisplayStringLine(Line5, "JOY: ERROR ");                     26
        break;                                                           27
    }                                                                    28
    TS_State = IOE_TS_GetState();                                        29
    Delay(1);                                                            30
    if (STM_EVAL_PBGetState(Button_KEY) == 0) {                          31
      STM_EVAL_LEDToggle(LED1);                                          32
      LCD_DisplayStringLine(Line4, "Pol: KEY Pressed ");                 33
    }                                                                    34
    if (STM_EVAL_PBGetState(Button_TAMPER) == 0) {                       35
      STM_EVAL_LEDToggle(LED2);                                          36
      LCD_DisplayStringLine(Line4, "Pol: TAMPER Pressed ");              37
    }                                                                    38
    if (STM_EVAL_PBGetState(Button_WAKEUP) != 0) {                       39
      STM_EVAL_LEDToggle(LED3);                                          40
      LCD_DisplayStringLine(Line4, "Pol: WAKEUP Pressed ");              41
    }                                                                    42
  }                                                                      43
}                                                                        44
```

III. A FORMALIZATION OF SEQUITUR

In the literature, Sequitur [9] is described at an informal level. In this section we propose a formalization of Sequitur and the grammars that it generates.

Preliminaries: An alphabet Σ is a finite set of symbols. Given an alphabet Σ, a *string* (or word) is a sequence of symbols of Σ. The set of all finite strings on Σ is noted Σ^*. The length of a string $\sigma \in \Sigma^*$, noted $|\sigma|$, is the number of symbols in σ. The empty string over Σ, i.e., the string of length 0, is noted ϵ_Σ or ϵ when clear from context. By σ_i we indicate the i-th element of σ. A *digram* is defined as a string of length 2.

A. Sequitur Grammars

Below we present the grammar generated by Sequitur and their properties.

Definition 1 (Sequitur Grammar)**.** *A* Sequitur grammar *is a tuple* $\langle \Sigma, \Gamma, S, \Delta \rangle$ *where:*

- Σ *is a finite alphabet of* terminal *symbols,*
- Γ *is a disjoint finite alphabet of* nonterminal *symbols,*
- $S \in \Gamma$ *is a* start symbol*, i.e., a particular nonterminal,*
- *and* $\Delta \subseteq \Gamma \times (\Sigma \cup \Gamma)^*$ *is a finite set of* production rules *(or simply* rules*),*

such that the following properties are verified:

- *for every nonterminal* A*, there is a unique string* α *such that* $\langle A, \alpha \rangle$ *is in* Δ *and* $|\alpha|$ *is at least 2,*
- *there is a strict partial order over* Γ*, such that, for all production rule* $\langle A, \alpha \rangle \in \Delta$*, every nonterminal in* α *precedes* A*.*

A production rule $\langle A, \alpha \rangle \in \Delta$ is noted $A \rightarrow \alpha$, where the nonterminal A (resp. the string α) is called the *head* (resp. the *body*) of the rule. By extension, the *grammar body* is the set of the rules' bodies $\{\alpha \mid \exists A : \langle A, \alpha \rangle \in \Delta\}$. The *start rule* is the unique rule whose head is the start symbol S. Note that in Sequitur grammars, there is exactly one rule $\gamma \rightarrow \sigma$ per nonterminal $\gamma \in \Gamma$. In addition, a partial order over Γ ensures that the grammar is non-recursive (cycle-free). In the following, a Sequitur grammar is simply called a grammar and we consider a grammar $G = \langle \Sigma, \Gamma, S, \Delta \rangle$. When clear from context, we might refer to a rule by its head.

```
1  let S be a fresh nonterminal (S ∉ Σ ∪ Γ0)
2  G ← ⟨Σ, Γ0 ∪ {S}, S, Δ0 ∪ {⟨S, ϵ⟩}⟩
3  for i = 1 to |ω| do
4      append ωi to body of rule of S
5      while ¬ Uniqueness(G) ∨ ¬ Utility(G) do
6          if ¬ Uniqueness(G) then
7              let δ be repeated digram in G
8              if ∃⟨A, α⟩ ∈ Δ : α = δ then
9                  replace the other occurrence of δ in G
                   with A
10             else
11                 form new rule ⟨D, δ⟩ where D ∉ (Σ ∪ Γ)
12                 replace both occurrences of δ in G
                   with D
13                 Δ ← Δ ∪ {⟨D, δ⟩}
14                 Γ ← Γ ∪ {D}
15             end
16         else if ¬ Utility(G) then
17             let ⟨A, α⟩ ∈ Δ be a rule used once
18             replace the occurrence of A with α in G
19             Δ ← Δ ∖ {⟨A, α⟩}
20             Γ ← Γ ∖ {A}
21         end
22     end
23 end
24 return G
```

Fig. 1. Function Sequitur$(\Sigma, \omega, \Gamma_0, \Delta_0)$

B. Algorithm

Sequitur, proposed by Nevill-Manning and Witten [9], is a grammar-based compression algorithm used in a variety of fields, ranging from bio-informatics to natural language processing. To generate a compression, Sequitur takes a string as input, and finds repeated subsequences present in the string to build a compact grammar. It operates in linear time and in an online fashion. Each repetition gives rise to a rule in the grammar, and is replaced with a nonterminal symbol. The compression process is executed iteratively. To illustrate how Sequitur operates, let us consider the string: $abcabcabcabcabc$. Sequitur generates the following grammar:

$$S \rightarrow AAB$$
$$A \rightarrow BB$$
$$B \rightarrow abc$$

The original string contains 15 symbols and the grammar generated using Sequitur contains 11 symbols. The compression results in the explicit capture of the repetitions of the subsequence abc. We shall explain how this resulting grammar is obtained using the algorithm of Fig. 1. The referred lines are those of Fig. 1. Sequitur takes as input four parameters: a set of terminal symbols Σ, a string ω, the initial set of nonterminal symbols Γ_0, which can be the empty set or not[1], and Δ_0 the initial set of rules (can be empty or not). Sequitur starts with a start rule with ϵ as body, and iterates over the characters of the input string. Each character is handled in two phases. First, Sequitur adds the new character as a terminal symbol at the end of the body of the start rule (line 4). Second, if it is necessary (line 5), it applies one or more reduction steps in order to ensure two properties on the grammar: *digram uniqueness* (from line 6 to 15) and *rule utility* (from line 16 to 20).

Step	i	Grammar	Action	Line(s)	D. Uniq.	R. Utility
1	1	$S \rightarrow a$	append	4	true	true
2	2	$S \rightarrow ab$	append	4	true	true
3	3	$S \rightarrow abc$	append	4	true	true
4	4	$S \rightarrow abca$	append	4	true	true
5	5	$S \rightarrow abcab$	append	4	**false**	true
6	–	$S \rightarrow AcA$ $A \rightarrow ab$	add rule	11–14	true	true
7	6	$S \rightarrow AcAc$ $A \rightarrow ab$	append	4	**false**	true
8	–	$S \rightarrow BB$ $A \rightarrow ab$ $B \rightarrow Ac$	add rule	11–14	true	**false**
9	–	$S \rightarrow BB$ $B \rightarrow abc$	remove rule	17–20	true	true

TABLE I
GRAMMAR CONSTRUCTION FOR $abcabc$

1) Digram uniqueness: It states that a grammar should not contain two non-overlapping occurrences of the same digram in the grammar body.

Property 1 (Digram uniqueness)**.** *The digram uniqueness property holds for G, noted Uniqueness(G), if for all terminals $A, B \in \Gamma$, symbols $a, b, c, d \in \Sigma \cup \Gamma$, and strings $\alpha, \beta, \gamma, \delta \in (\Sigma \cup \Gamma)^*$, the two following statements hold:*

$$(A \neq B \wedge A \rightarrow \alpha\ ab\ \beta \wedge B \rightarrow \gamma\ cd\ \delta) \Rightarrow ab \neq cd \quad (1)$$
$$(A \rightarrow \alpha\ ab\ \beta\ cd\ \gamma) \Rightarrow ab \neq cd. \quad (2)$$

Statement (1) says that if G contains two rules $A \rightarrow \alpha\, ab\, \beta$ and $B \rightarrow \gamma\, cd\, \delta$ with $A \neq B$, then ab is not equal to cd. Statement (2) says that if G contains a rule $A \rightarrow \alpha\, ab\, \beta\, cd\, \gamma$, then ab is not equal to cd.

Sequitur ensures Property 1 as follows. If adding a symbol (line 4) produces a non-overlapping repetition of a digram in the grammar body, then the property does not hold anymore (line 6). Sequitur restores the property either by creating a new rule or by reusing an existing rule (lines 7–15). If the repeated digram corresponds already to the body of an existing rule, Sequitur replaces this digram with the corresponding nonterminal (line 9), which represents the concerned rule. Otherwise, it forms a new rule with the digram as body and a new nonterminal as head, and replaces both occurrences of the digram with this new nonterminal symbol (lines 11–14).

[1]In the original version of Sequitur, such a parameter does not exist. We present a parameterized form of Sequitur as it will be used by our extension in the next section.

Table I details step-by-step the application of Sequitur on the input string $abcabc$. For each step, this table indicates the number of read symbols (i), the current grammar, the undertaken action with the corresponding line(s) in the algorithm, and the status of the two properties (digram uniqueness and rule utility). On this example, after adding the symbol b to the body of rule $\langle S, abca\rangle$ at step 5, the digram ab occurs twice, and consequently the digram uniqueness property is violated. Sequitur creates a new rule $\langle A, ab\rangle$ with a new nonterminal A as head, and the digram ab as body, to replace both digram occurrences in the grammar with A, as in step 6.

2) Rule utility: It ensures that every rule (nonterminal) except the start rule is used more than once in the grammar body.

Property 2 (Rule utility)**.** *The* rule utility *property holds for G, noted $Utility(G)$ if:*

$$\forall A \in \Gamma\backslash\{S\} : \left(\sum_{\langle B,\beta\rangle \in \Delta} \left|\{i \in [1..|\beta|] \mid \beta_i = A\}\right| \right) \geq 2\,.$$

Sequitur ensures the rule utility property as follows. If a rule is referred to once (line 16), Sequitur eliminates it, and replaces the occurrence of its head (nonterminal) with its body (lines 17–20). Note that, this mechanism allows the formation of rules whose body consists of more than two symbols.

On the example of Table I, after adding the next input symbol c at step 7, Sequitur introduces the new rule $B \to Ac$ to ensure the digram uniqueness property in step 8. In the resulting grammar, the nonterminal A occurs once in the grammar body, precisely in the body of rule $B \to Ac$. In this case, in step 9, to restore the rule utility property, Sequitur eliminates rule $A \to ab$ and replaces the occurrence of A in the body of $B \to Ac$ with the digram ab.

Note that Sequitur terminates on all finite input strings. Each input symbol is added one at a time and each addition may trigger a finite sequence of reduction steps. Indeed, the sequence of steps is finite because each reduction step, for either property, ensures that the following quantity strictly decreases (w.r.t. the lexicographical order on $\mathbb{N}^2$):

$$\left\langle \sum_{\langle A,\alpha\rangle \in \Delta} \left|\{i \in [1..|\alpha|] \mid \alpha_i \in \Sigma\}\right|, \quad \left|\Delta\right| \right\rangle,$$

that is, either the number of terminal occurrences in the grammar (i.e., the sum of the number of terminal occurrences in each rule body) decreases, or it stays the same but the number of rules decreases. Sequitur compresses a string in linear time [9]. Indeed, given an input string of length n, if each addition may lead to up to $O(\sqrt{n})$ reduction steps, the overall algorithm execution does not take more than $O(n)$ reduction steps.

IV. Microcontroller Trace Compression

We describe in this section the improved Sequitur algorithm.

A. Squeezing repetitions

1) Enhanced string (r-string) and grammar (r-grammar): Given an alphabet Σ, an *r-string* ψ is a sequence of symbols with a number of consecutive repetitions. In other words, it is a sequence of pairs $\langle$*symbol*, *strictly positive integer*$\rangle$. The set of *r-strings* over Σ is $\Sigma_r^* = (\Sigma \times \mathbb{N} \setminus \{0\})^*$. In an r-string ψ, $\psi_{i,1}$ stands for the symbol of the i-th element of ψ and $\psi_{i,2}$ stands for its number of repetitions. $|\psi|$ denotes the number of elements in the r-string ψ. To lighten notations, repetition numbers are placed in superscript after symbols and they are omitted when equal to one. For instance, ab^5d^{10} is a shorthand for the sequence $\langle a, 1\rangle\langle b, 5\rangle\langle d, 10\rangle$. The *expansion* of the r-string $\psi \in \Sigma_r^*$, noted $\widehat{\psi}$, is a string in Σ^*, and is defined as follows:

$$\widehat{\psi} = \underbrace{\psi_{1,1} \cdots \psi_{1,1}}_{\text{repeated } \psi_{1,2} \text{ times}} \underbrace{\psi_{2,1} \cdots \psi_{2,1}}_{\psi_{2,2} \text{ times}} \cdots \underbrace{\psi_{|\psi|,1} \cdots \psi_{|\psi|,1}}_{\psi_{|\psi|,2} \text{ times}}\,.$$

Note that $|\psi|$ indicates the number of pairs $\langle$*symbol*, *number*$\rangle$, and not the number of symbols in the expansion ($|\widehat{\psi}|$).

Definition 2 (r-grammar)**.** *An r-grammar G' is a 4-tuple $\langle \Sigma, \Gamma, S, \Delta'\rangle$ where:*

- *Σ is a finite alphabet of* terminal *symbols,*
- *Γ is a disjoint finite alphabet of* nonterminal *symbols,*
- *$S \in \Gamma$ is a* start symbol*, i.e., a particular nonterminal,*
- *and $\Delta' \subseteq \Gamma \times (\Sigma \cup \Gamma)_r^*$ is a set of* r-production rules*,*

such that the following properties are verified:

- *for every nonterminal A, there is a unique r-string α such that $\langle A, \alpha\rangle$ is in Δ',*
- *there is a strict partial order over Γ, such that, for all production rule $\langle A, \alpha\rangle \in \Delta'$, every nonterminal in α precedes A.*

An r-production rule $\langle A, \alpha\rangle \in \Delta'$ is noted $A \to \alpha$, where the nonterminal A (resp. the r-string α) is called the *head* (resp. the *body*) of the rule. The *r-grammar body* is $\{\alpha \mid \exists A : \langle A, \alpha\rangle \in \Delta'\}$, i.e., the set of the rules' bodies. In the following, we consider an r-grammar $G' = \langle \Sigma, \Gamma, S, \Delta'\rangle$.

2) Enhanced compression ReSequitur: Given in Fig. 2, ReSequitur is inspired from Sequitur [9]. Indeed, it ensures that adapted versions of digram uniqueness and rule utility properties hold on its output r-grammar:

Property 3 (Digram uniqueness)**.** *The digram uniqueness property holds for G', noted $RUniqueness(G')$, if for all terminals $A, B \in \Gamma$, symbols $a, b, c, d \in \Sigma \cup \Gamma$, strictly positive integers $n, m, p, q \in \mathbb{N}\backslash\{0\}$, and r-strings $\alpha, \beta, \gamma, \delta \in (\Sigma \cup \Gamma)_r^*$, the two following statements hold:*

$$(A \neq B \wedge A \to \alpha\, a^n b^m\, \beta \wedge B \to \gamma\, c^p d^q\, \delta) \implies a^n b^m \neq c^p d^q \tag{3}$$

$$(A \to \alpha\, a^n b^m\, \beta\, c^p d^q\, \gamma) \implies a^n b^m \neq c^p d^q\,. \tag{4}$$

Statement (3) states that if Δ' contains two rules $A \to \alpha\, a^n b^m\, \beta$ and $B \to \gamma\, c^p d^q\, \delta$ with $A \neq B$, then $a^n b^m$ is not

equal to $c^p d^q$ (that is, $a \neq c \vee b \neq d \vee n \neq p \vee m \neq q$). Statement (4) states that if Δ' contains a rule $A \rightarrow \alpha\, a^n b^m\, \beta\, c^p d^q\, \gamma$, then $a^n b^q$ is not equal to $c^p d^q$.

Property 4 (Rule utility)**.** *The rule utility property holds for G', noted $RUtility(G')$, if:*

$$\forall A \in \Gamma \setminus \{S\} : \left(\sum_{\langle B,\beta\rangle \in \Delta'} \sum_{i \in [1..|\beta|]} \begin{cases} \beta_{i,2} & \text{if } \beta_{i,1} = A \\ 0 & \text{if } \beta_{i,1} \neq A \end{cases} \right) \geq 2 .$$

Property 4 states that for each nonterminal $A \in \Gamma$, if A is not the start symbol S, it must be used at least twice in the grammar body (taking account consecutive repetitions).

In addition, ReSequitur ensures another property stating that any digram in an r-grammar body is composed of different symbols.

Property 5 (No consecutive repetition)**.** *G' has* no consecutive repetitions*, which is noted $RConsecutive(G')$, if:*

$$\forall a, b \in \Sigma \cup \Gamma, \forall n, m \in \mathbb{N} \setminus \{0\}, \forall \alpha, \beta \in (\Sigma \cup \Gamma)^*_r, \forall C \in \Gamma : \langle C, \alpha\, a^n b^m \beta \rangle \in \Delta' \Rightarrow a \neq b .$$

To ensure this property at each iteration of the outer loop, ReSequitur merges every digram of the form $a^n a^m$ into a single repeated symbol a^{n+m} (Fig. 2, lines 6–8).

B. Detecting and exploiting cycles with Cyclitur

Recall that the objective of our work is to compress cyclic traces extracted from microcontrollers. The main characteristic of the programs at the origin of these traces, is the active main loop, which is often expressed using a `while(1)` statement.

The first step in our approach is *cycle detection*. A cycle is a subsequence of the execution trace that represents one execution of the main active loop of the embedded program. Cycle detection relies on the localization of a special event that represents the loop header, noted lh.

Definition 3 (Set of Cycles)**.** *For a string ω, and a loop header lh, the set of cycles in ω using the loop header lh is defined as:*

$$\begin{aligned} C(\omega, lh) = \{ \langle i, j \rangle \in \mathbb{N}^2 \mid\ & 1 \leq i \leq j \leq |\omega| \\ & \wedge\, \omega_i = lh \\ & \wedge\, (j = |\omega| \vee \omega_{j+1} = lh) \\ & \wedge\, \forall k \in [i+1..j] : \omega_k \neq lh \} . \end{aligned}$$

$C(\omega, lh)$ is a set of pairs of indexes, where each pair $\langle i, j \rangle$ represents a subsequence (cycle) $\omega_{i..j}$. For each cycle $\langle i, j \rangle$, the subsequence $\omega_{i..j}$ starts with the loop header lh (except for the first one if the trace does not start with the loop header), contains at most one occurrence of the loop header lh, and ends either with the symbol before the next occurrence of the loop header lh or at the end of the sequence ω.

Detecting cycles using the loop header event essentially consists in dividing the trace into blocks, where each block represents a specific cycle. ReSequitur is applied on each of these block, to detect repetitions inside cycles (Fig. 3, lines 2–7), while sharing the same set of rules. Applying ReSequitur algorithm on the compression produced by the previous step allow us to detect similar sequences of cycles in the trace (Fig. 3, line 8).

```
1  let S be a fresh nonterminal representing a rule
   (S ∉ Σ ∪ Γ0)
2  G ← ⟨Σ, Γ0 ∪ {S}, S, Δ0 ∪ {⟨S, ϵ⟩}⟩
3  for i ← 1 to |ω| do
4      append (ω_i)^1 to body of rule S
5      while ¬RUniqueness(G) ∨ ¬RUtility(G) ∨
       ¬RConsecutive(G) do
6          if ¬RConsecutive(G) then
7              let a, n, m be s.t. a^n a^m is a r-digram in G
8              replace every occurrence of a^n a^m in G
               with a^(n+m)
9          else if ¬RUniqueness(G) then
10             let δ be a repeated r-digram in G
11             if ∃⟨A, α⟩ ∈ Δ : α = δ then
12                 replace the other occurrence of δ in G
                   with A
13             else
14                 form new rule ⟨D, δ⟩ where D ∉ (Σ ∪ Γ)
15                 replace both occurrences of δ in G
                   with D
16                 Δ ← Δ ∪ {⟨D, δ⟩}
17                 Γ ← Γ ∪ {D}
18             end
19         else if ¬RUtility(G) then
20             let ⟨A, α⟩ ∈ Δ be a rule used once
21             replace the occurrence of A with α in G
22             Δ ← Δ ∖ {⟨A, α⟩}
23             Γ ← Γ ∖ {A}
24         end
25     end
26 end
27 return G
```

Fig. 2. Function ReSequitur$(\Sigma, \omega, \Gamma_0, \Delta_0)$

```
1  ω' ← ϵ
2  foreach ⟨i, j⟩ ∈ C(ω, lh) in increasing order do
3      ⟨Σ', Γ', S', Δ'⟩ ← ReSequitur(Σ, ω_(i..j), Γ0, Δ0)
4      Γ0 ← Γ'
5      Δ0 ← Δ'
6      ω' ← ω' · S'
7  end
8  ⟨Σ'', Γ'', S'', Δ''⟩ ← ReSequitur(Σ, ω', Γ0, Δ0)
9  return ⟨Σ'', Γ'', S'', Δ''⟩
```

Fig. 3. Function Cyclitur$(\Sigma, \omega, \Gamma_0 = \varnothing, \Delta_0 = \varnothing, lh)$

V. Implementation and Evaluation

The first step in the evaluation of our trace-compression approach is an experimental evaluation, which consists in comparing grammars obtained by applying Sequitur and Cyclitur on various execution traces. The experimental evaluation was made possible thanks to J-Cyclitur. The second step is a qualitative evaluation, which evaluates the quality of the produced compressions, and measures their capability to help software engineers decide which parts they want to analyze in detail.

A. J-Cyclitur

J-Cyclitur is a tool written in Java in 4,000 LOC that implements both Sequitur and Cyclitur algorithms. It takes as input an execution trace file. It extracts automatically a string of symbols. For instance, given a microcontroller execution trace, a symbol is a program counter. Then J-Cyclitur compresses the string using Sequitur and Cyclitur algorithms. Finally, it outputs a grammar, either as text or as a Java object for programmatic use.

B. Metrics of experimental evaluation

In the following, we use a given string (trace) ω, and the output grammar (resp. r-grammar) generated with Sequitur (resp. Cyclitur), noted $G = \langle \Sigma, \Gamma, S, \Delta \rangle$.

1) Grammar size: The *size* of the grammar G is the sum of the number of symbol occurrences in its body (both terminals and nonterminals) and the number of its rules.

2) Compression ratio: Noted $Comp(G)$, this metric is used to compare the degree of compression of grammars generated using Sequitur and Cyclitur.

$$Comp(G) = \frac{Size(G)}{|\omega|}$$

3) Compressed cycle size: For a generated compression, this third metric indicates how many symbols are analyzed in the grammar to detect a particular cycle. This metric can also be seen as the number of paths that the debugger engineer shall follow in the grammar to find an entire cycle.

Note that the cycle detection step in Cyclitur ensures that each cycle corresponds to a single symbol, then the *compressed cycle size* is always one with Cyclitur.

C. Programs and traces

The traces used to evaluate our approach come from five embedded programs, which are provided by STMicroelectronics and EASii IC. For confidentiality reasons, programs are not described. In the following we denote by Pi the program number i.

Once at a time, each program is downloaded on a STM32F107 EVAL-C microcontroller board and executed. The trace produced by the execution is recovered using a Keil Ulinkpro probe [12], and saved in *CSV* format. In the trace file, for each instruction, we have its *index*, which is an ID, the *time* when it was executed, its corresponding *assembly instruction* and the *program counter (PC)*. For our compression approach we are only interested in the PCs. For each program, five execution traces are produced.

D. Results

Table II contains the results of the experimental evaluation, where each line represents a trace of a program. The two columns (*# Symbols*) and (*# Cycles*) represent respectively the number of symbol occurrences and the number of cycles in a trace. For a grammar G generated by Sequitur, Table II contains its size ($Size(G)$), its compression ratio ($Comp(G)$), and the average of the compressed cycle size over cycles ($AvgAC(G)$). For a grammar G' generated by Cyclitur, Table II contains its size ($Size(G')$), and its compression ratio ($Comp(G')$). For Cyclitur, Table II does not contain $AvgAC(G')$, which is the average of the compressed cycle size in the grammar G', because, by construction, the compressed cycle size of any cycle in a Cyclitur-generated grammar is always equal to one.

Fig. 4 graphically summarizes these results. For both algorithms (Sequitur in gray and Cyclitur in white), and for each program, Fig. 4 displays the arithmetic average of the compression ratios over the five collected traces of the program.

Note that the use of the other average measures gives different values, but the same result: the difference between the generated compressions using Sequitur and Cyclitur. In Fig. 4, the arithmetic average is used to have a better illustration. The compression ratio takes value between 0 and 1, where 0 represents the best compression, and 1 the worst.

In Fig. 4, we compare for each program, the average of compression computed using the grammars generated by Sequitur and the grammars generated by Cyclitur. For example, for program P1, we observe that our approach produces better compression than Sequitur. The sizes of grammars generated by Sequitur for the execution traces of P1 vary from 1,181 up to 2,631; they contain 214-387 terminals, 728-1,689 nonterminals, and 239-1,689 rules. The sizes of original traces vary between 1,048,571 and 1,048,576, with 31,562-53,821 cycles. Therefore, the ratios of compression vary between 0.0011 and 0.0025. Cyclitur produces grammars whose sizes vary from 1,139 to 2,044 with 213-385 terminals, 687-1,237 nonterminals, and 239-423 rules. The compression ratios vary between 0.0010 and 0.0019. Also, when Sequitur-generated grammars require 5-13 symbols ($AvgAC(G)$) to represent a cycle, Cyclitur ensures each cycle can be represented as a unique symbol. In Fig. 4, we note that for all the programs used in the experimental evaluation, the use of Cyclitur generates a better compression than Sequitur. The ratios of compression are better from 15% to 30%.

E. Cyclitur and network traces

We evaluate our trace-compression approach on four additional traces obtained from network simulations. The considered network is a Multi-Channel Multi-Interface Wireless Mesh Network (WMN) with routers based on the IEEE 802.11 technology [15]. The loop header used to detect cycles and

TABLE II
EVALUATION RESULTS

Program	Trace			Sequitur			Cyclitur	
		# Symbols	# Cycles	$Size(G)$	$Comp(G)$	$Avg(AC)$	$Size(G')$	$Comp(G')$
P1	Trace 1	1048575	53821	2631	0.002509120	12.004738	2044	0.001949312
	Trace 2	1048576	51574	1884	0.001796722	9.207381	1455	0.001387596
	Trace 3	1048576	50482	1814	0.001729965	5.076247	1798	0.001714706
	Trace 4	1048571	53819	1880	0.001792916	13.413709	1510	0.001440055
	Trace 5	1048574	31562	1181	0.001126292	7.039004	1139	0.001086237
P2	Trace 1	1048575	27542	22012	0.020992299	4.867071	17945	0.017113702
	Trace 2	1048575	10621	20317	0.019375820	4.411111	17728	0.016906754
	Trace 3	1048575	26043	19662	0.018751162	3.932647	15190	0.014486327
	Trace 4	1048576	1038	20674	0.019716263	7.436837	17002	0.016214371
	Trace 5	1048574	32515	20116	0.019184149	6.086793	15888	0.015152006
P3	Trace 1	1048572	49208	1918	0.001829154	8.897677	1331	0.001269345
	Trace 2	1048571	49207	1830	0.001745232	4.697557	1397	0.001332289
	Trace 3	1048573	49205	1813	0.001729016	11.443277	1463	0.001395230
	Trace 4	1048576	49207	1961	0.001870155	4.955595	1409	0.001343727
	Trace 5	1048576	49209	1842	0.001756668	9.495285	1741	0.001660347
P4	Trace 1	1048567	47029	1846	0.001760498	6.724356	1498	0.001428616
	Trace 2	1048571	53854	2250	0.002145777	8.729226	1630	0.001554497
	Trace 3	1048574	47031	2032	0.001937870	9.841527	1458	0.001390460
	Trace 4	1048575	53860	1808	0.001724245	5.849607	1494	0.001424791
	Trace 5	1048575	53866	1947	0.001856806	8.279625	1716	0.001636507
P5	Trace 1	1048573	64527	309	0.000294686	12.249899	139	0.000132561
	Trace 2	1048576	64527	304	0.000289917	2.810030	142	0.000135422
	Trace 3	1048570	52782	1641	0.001564989	9.285046	1334	0.001272209
	Trace 4	1048571	64526	317	0.000302316	2.594638	144	0.000137330
	Trace 5	1048576	64528	298	0.000284195	5.250004	146	0.000139236

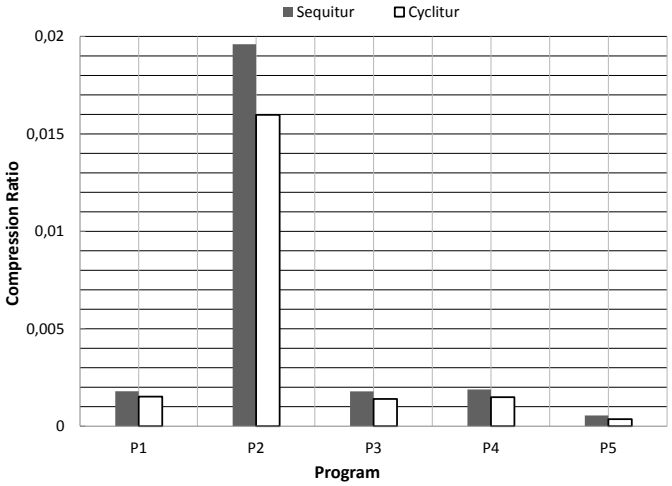


Fig. 4. Compared average compression ratios for each program

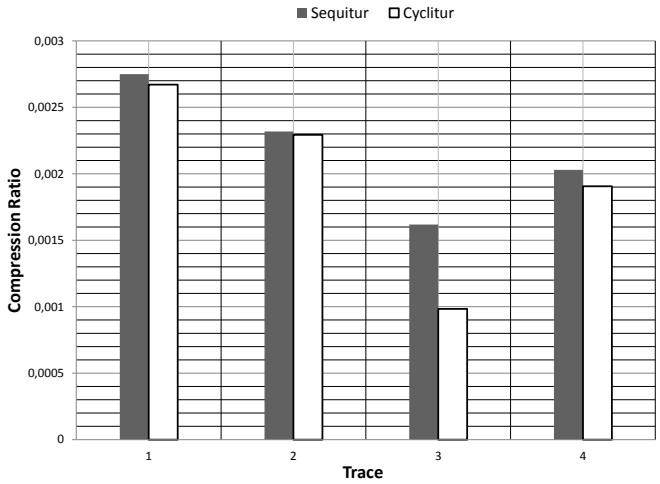


Fig. 5. Compared compression ratios for each WMN trace

to divide generated traces into blocks is a specific event that refers to the emission a request from client to server.

The first trace consists of 6,011,850 events spread over 9,574 cycles. The ratio of compression using Sequitur is 0.0027% for a generated grammar whose size is 16,531, and which contains 744 terminals, 12,375 nonterminals and 3,412 rules. The size of the grammar generated using Cyclitur is 16,057; it contains 373 terminals, 13,884 nonterminals and 4182 rules. The compression ratio for Cyclitur grammar is 0.0026%.

The second trace consists of 8,040,942 events spread over 9,574 cycles. Sequitur generates a grammar whose size is 18,639, and which contains 716 terminals, 14,042 nonterminals and 3,881 rules. The size of the grammar generated using Cyclitur is 18,439; it contains 373 terminals, 13,884 nonterminals and 4,182 rules. The use of Sequitur and Cyclitur on the second trace gives respectively 0.00023% and 0.0022% as compression ratio. Cyclitur generates a grammar that contains more rules than the grammar generated by Sequitur, although the Cyclitur compression is easier to understand and to analyze, because it is more compact and it facilitates cycle detection.

The third WMN trace contains 10,312,955 events spread over 33,834 cycles. Sequitur generates a grammar that contains 605 terminals, 12,584 nonterminals, and 3,501 rules. The compression ratio for the Sequitur grammar is 0.0016%. Cyclitur generates a grammar that contains 272 terminals, 7,783 nonterminals, and 2,090 rules. The compression ratio

for the Cyclitur grammar is 0.0009%.

The last WMN trace contains 13,883,977 events spread over 2,797 cycles. The ratio of compression using Sequitur is 0.0020% for a generated grammar whose size is 28,181 and which contains 661 terminals, 21,575 nonterminals and 5,945 rules. The size of the grammar generated using Cyclitur is 26,468; it contains 349 terminals, 20,108 nonterminals and 6,011 rules. The compression ratio for Cyclitur grammar is 0.0019%.

The previous results show that Cyclitur can be used likewise to compress trace collected from networks. In Fig. 5, we note that for all network traces used in the experimental evaluation, the use of Cyclitur generates a better compression than Sequitur. Note that the compression ratio takes value between 0 and 1, where 0 represents the best compression, and 1 the worst.

F. Quality evaluation

We performed a study to evaluate the quality of the produced compressions, and to measure their capability to help software engineers decide which parts they want to analyze in detail.

1) Subjects and the object of study: To evaluate trace compressions, we made an online evaluation using:

- An execution trace that contains 15,391 lines[2],
- The Sequitur-generated compression[3] (123 lines),
- The Cyclitur-generated compression[4] (113 lines).

The execution trace was obtained by running on a STM32 evaluation board, a program that simply consists in counting the number of iterations of the main loop. If the iteration number is divisible by two, the program prints a specific text on the LCD screen; otherwise it turns on some LEDs for two seconds.

Without providing program source code, we asked 20 software engineers to evaluate the capability of the compressions to assist in the understanding of system behavior. Assisting in the understanding of system behavior consists in helping decide which cycles in the trace should be analyzed in detail. Among the software engineers that participated to this quality evaluation, 20% analyze execution traces never or rarely, 55% analyze execution traces from occasionally to often, and 25% analyze execution traces from frequently to always.

2) The evaluation of compressions: To measure the subjective quality of the trace compression techniques, we designed an online opinion poll, which involves the active participation of human judges. Participants rate the compressions generated using Sequitur and Cyclitur based on the help provided to decide what parts of trace they need to analyze in detail.

In the first phase, the software engineers were presented with the generated trace and asked to answer the question "*What do you think about the trace?*". For the question, the responders had to choose one or more choices among the following:

- "Huge amount of data",
- "Incomprehensible data",
- "Comprehensible data".

The size of the provided trace represents only around 2% of the size of one of our benchmark trace, however, 60% of the responders agreed that the trace file contains a huge amount of data. Concerning the trace comprehension, while 30% found the trace comprehensible, 55% found it incomprehensible.

In the second phase, the software engineers were presented with the compression generated using Sequitur and asked to answer the question "*Does the compression help you decide which parts you want to analyze in detail?*". The software engineers answered by "*yes*" or "*no*". The results show that 65% answered by no, while 35% found that the Sequitur compression helps to decide which parts to analyze in detail.

The next phase consists in evaluating Cyclitur compression. The developers were presented with the compression generated using Cyclitur and asked to answer two questions. First question was "*Do you notice a specific cycle in the compression?*". The developers answered "*yes*" or "*no*". By analyzing the results we observe that 60% of the software engineers found that the Cyclitur compression helps to identify which of cycles should be analyzed in detail. According to the suggestions of the software developers that answered the question by "*no*", the main reason behind their negative answer choice was the absence of a visualization.

The second question was "*With respect to the Sequitur compression, do you think that Cyclitur compression is more helpful?*". The software engineers had five options to choose from to answer the previous question, where each option being assigned a score: *not helpful* (assigned a score of 1), *slightly helpful* (score of 2), *moderately helpful* (assigned a score of 3), *very helpful* (assigned a score of 4), *extremely helpful* (assigned a score of 5).

We observe that 100% of software engineers found that the Cyclitur compression is more helpful in analyzing an execution trace than the Sequitur compression. In more details, 20% had found that the Cyclitur compression is extremely helpful comparing to the Sequitur compression, and 25% found it very helpful.

In order to understand how developers performed the evaluation, we asked them to answer one follow-up question, which is: "*Using Cyclitur compression, what did help you decide which cycle(s) should be analyzed in detail*". The cycle detection and the position of cycles in the Cyclitur compression were used up to 57% to decide which cycles the software engineers need to analyze. The number of repetitions of cycles in the compression was used 89% to decide which cycles to analyze.

VI. Related Work

Compressing traces of microcontrollers with the objective of analyzing them remains a challenge. In other areas, particularly in object oriented context, there are numerous studies concerning reduction and compression of execution traces.

[2]Available online at http://io32.forge.imag.fr/formulaire/Trace.txt
[3]Available online at http://io32.forge.imag.fr/formulaire/Sequitur.txt
[4]Available online at http://io32.forge.imag.fr/formulaire/Cyclitur.txt

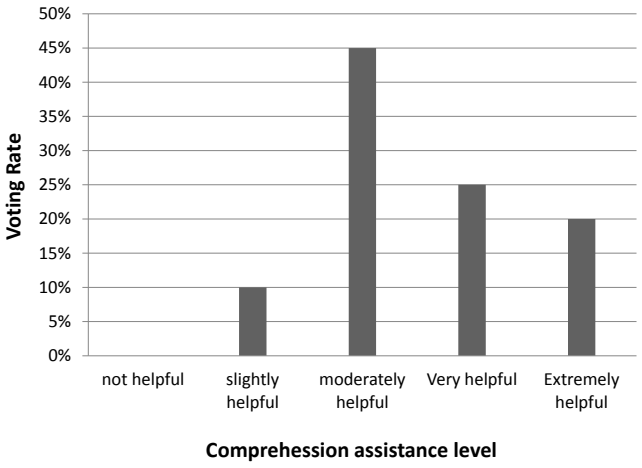


Fig. 6. The Understanding assistance level of Cyclitur compression

Hamou-Lhadj and Lethbridge [7] use an acyclic oriented graph representing method calls to compress traces. A rule-based approach using method calls is proposed in [16], [17], which aims to build a compression by using the construction of trees. It is possible to summarize large execution traces using finite automata like in [18]. In addition to their trace compression, in [14] Hamou-Lhadj and Lethbridge, propose the deletion of implementation and useless details to ease the analysis of execution traces. These object oriented approaches are not suitable for our purpose for multiple reasons. First, they discard the order of events, which is paramount to understand a program. Second, they use input/output data. In our context, such information is rarely available, and raises important storage problems. Third, these approaches reason about method calls. In optimized microcontroller code, function calls alone are often inadequate to understand the program, since the core logic of a program is sometimes coded in a single function.

General data compression methods have been used for program traces, for example, Gzip [19], or VPC4 [8]. However, such compression usually generates a result that is not understandable by the engineer. The use of these techniques may imply loss of reference points, which are cycles in our approach. We think that cycles revealed in the process of compression may assist automatic trace analysis, for example using cycle matching [20].

An approach proposed by James R. Larus in [13] captures the program execution using trace recording, and represents a dynamic executed control flow of a program. To compress traces, the author uses Sequitur to find inherent regularities (e.g., repeated code). The DAG representation of grammar produced by Sequitur is called Whole-Program Path (WPP). This approach is interesting; however, its main motivation is the detection of hot subpaths, which are short sequences of acyclic paths that are costly. Our work consists in compressing traces by detecting and exploiting cycles. It allows us to reach better compression ratios than Sequitur. Also, our approach keeps cycles as reference points while the WWP approach does not take cycles into account. That is why, the WWP approach is not relevant in our work context.

Similarly to the approach of James R. Larus, we rely on Sequitur, a grammar-based compression algorithm. Such algorithms are the objects of active research in information theory (cf. [21] for a survey). In particular, several extensions of Sequitur aimed at obtaining better compressions by producing smaller grammars. For instance, Yang and Kiefer proposed a generalization of Sequitur to n-grams (rather than digram) [22]. This change ultimately leads to smaller grammars. However using this algorithm comes at a price: the algorithm does not share the same time and space complexity as Sequitur, and as a result, is not usable on huge program traces. On the contrary, Cyclitur keeps the same complexity as the original Sequitur algorithm.

VII. Conclusion and Perspectives

In the microcontroller context, the new microprocessors ARM Cortex-Mx [11], enable recording the component behavior and save it as an execution trace file. Analyzing execution trace may help in the understanding and debugging tasks. However, even for a short execution time, the generated trace contains a huge amount of data, making the system comprehension task difficult and tedious. It is important to provide to the engineer in charge of debugging an overview of the execution trace.

In this article we propose an approach that aims to provide a compression of trace. The compression process is based on a grammar-based compression using our improvement of Sequitur [9], named *Cyclitur*. Our approach starts by dividing a trace into cycles, where each cycle is an execution of the active main loop. The second step consists in discovering and compressing similarities inside cycles. The detection and compression of repetitions and regularities in terms of cycles is the third and last step of our approach.

Our approach is evaluated, first quantitatively to compare its compression rate to the existing Sequitur algorithm; second qualitatively to determine if such compression could be useful to software engineers. The quantitative evaluation shows that our approach generates an equivalent or better compression than Sequitur on benchmark execution traces. On microcontroller execution traces, Cyclitur compression ratios were better than Sequitur compression ratios from 15% to 25%. The quality survey points out three important facts. First, software engineers usually find that Cyclitur-generated trace compressions are easier to analyze than their Sequitur-generated counterparts. Second, Cyclitur may help in identifying and locating important details in an execution trace. Third, our survey highlights the need of a graphical visualization for software engineers to analyze such a trace compression.

As a future work we intend to facilitate the understanding of the compression through visualization. We also intend to help in diagnostic by adapting dynamic validation and data mining techniques [23], for compressed traces, to help in fault localization.

ACKNOWLEDGMENT

This work has been funded by the French-government Single Inter-Ministry Fund (FUI) through the IO32 project (Instrumentation and Tools for 32-bit Microcontrollers). The authors would like to thank STMicroelectronics, AIM and ESAii IC for their help.

REFERENCES

[1] A. Rohani and H. Zarandi, "An analysis of fault effects and propagations in AVR microcontroller ATmega103(L)," in *International Conference on Availability, Reliability and Security (ARES)*, Mar. 2009, pp. 166–172.

[2] G. Shamnur and R. Berigei, "XStatic: A simulation based ESD verification and debug environment," in *9th International Symposium on Quality Electronic Design (ISQED)*, Mar. 2008, pp. 441–444.

[3] D. Chatterjee, C. McCarter, and V. Bertacco, "Simulation-based signal selection for state restoration in silicon debug," in *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, Nov. 2011, pp. 595–601.

[4] ARM. Embedded trace macrocells (ETM). [Online]. Available: http://www.arm.com/products/system-ip/debug-trace/trace-macrocells-etm/

[5] H. F. Ko and N. Nicolici, "Automated trace signals identification and state restoration for improving observability in post-silicon validation," in *Design, Automation and Test in Europe (DATE)*, Mar. 2008, pp. 1298–1303.

[6] K. Basu and P. Mishra, "Efficient trace signal selection for post silicon validation and debug," in *24th International Conference on VLSI Design*, Jan. 2011, pp. 352–357.

[7] A. Hamou-Lhadj and T. C. Lethbridge, "Compression techniques to simplify the analysis of large execution traces," in *10th International Workshop on Program Comprehension (IWPC)*. IEEE Computer Society, 2002, pp. 159–.

[8] M. Burtscher, I. Ganusov, S. Jackson, J. Ke, P. Ratanaworabhan, and N. Sam, "The VPC trace-compression algorithms," *IEEE Transactions on Computers*, vol. 54, no. 11, pp. 1329–1344, Nov. 2005.

[9] C. G. Nevill-Manning and I. H. Witten, "Identifying hierarchical structure in sequences: A linear-time algorithm," *Journal of Artificial Intellgence Research (JAIR)*, vol. 7, pp. 67–82, 1997.

[10] J. S. Parab, V. G. Shelake, R. K. Kamat, and G. Naik, "Exploring C for microcontrollers: A hands on approach," in *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. Springer, 2007, pp. 1–18.

[11] ARM. Cortex-M series. [Online]. Available: http://www.arm.com/products/processors/cortex-m/

[12] K. by ARM. ULINKpro. [Online]. Available: http://www.keil.com/ulinkpro/

[13] J. R. Larus, "Whole program paths," in *ACM SIGPLAN conference on Programming Language Design and Implementation (PLDI)*. ACM, 1999, pp. 259–269.

[14] A. Hamou-Lhadj and T. C. Lethbridge, "Summarizing the content of large traces to facilitate the understanding of the behaviour of a software system," in *14th IEEE International Conference on Program Comprehension (ICPC)*. IEEE Computer Society, 2006, pp. 181–190.

[15] C. De Oliveira, F. Theoleyre, and A. Duda, "Connectivity in multi-channel multi-interface wireless mesh networks," in *7th International Wireless Communications and Mobile Computing Conference (IWCMC)*, Jul. 2011, pp. 35–40.

[16] K. Taniguchi, T. Ishio, T. Kamiya, S. Kusumoto, and K. Inoue, "Extracting sequence diagram from execution trace of Java program," in *8th International Workshop on Principles of Software Evolution (IWPSE)*. IEEE Computer Society, 2005, pp. 148–154.

[17] A. Hamou-Lhadj, "Effective exploration and visualization of large execution traces," in *4th IEEE International Workshop on Visualizing Software for Understanding and Analysis (VISSOFT)*, Jun. 2007, pp. 152–153.

[18] M. Heizmann, J. Hoenicke, and A. Podelski, "Refinement of trace abstraction," in *16th International Symposium on Static Analysis (SAS)*. Springer-Verlag, 2009, pp. 69–85.

[19] J.-L. Gailly and M. Adler. Gzip. [Online]. Available: http://www.gzip.org/

[20] J. Aoe, *Computer Algorithms: String Pattern Matching Strategie*. IEEE Computer Society Press, 1994.

[21] M. Charikar, E. Lehman, D. Liu, R. Panigrahy, M. Prabhakaran, A. Sahai, and A. Shelat, "The smallest grammar problem," *IEEE Transactions on Information Theory*, vol. 51, no. 7, pp. 2554–2576, Jul. 2005.

[22] E.-H. Yang and J. C. Kieffer, "Efficient universal lossless data compression algorithms based on a greedy sequential grammar transform—Part one: Without context models," *IEEE Transactions on Information Theory*, vol. 46, no. 3, pp. 755–777, May 2000.

[23] U. Fayyad, G. Piatetsky-shapiro, and P. Smyth, "From data mining to knowledge discovery in databases," *AI Magazine*, vol. 17, pp. 37–54, 1996.

L I G