DAS Departamento de Automação e Sistemas
CTC Centro Tecnológico
UFSC Universidade Federal de Santa Catarina

Expressing Temporized Properties in Ladder Diagrams

Relatório submetido à Universidade Federal de Santa Catarina
como requisito para a aprovação da disciplina:
DAS 5511: Projeto de Fim de Curso

Mathieu Granzotto

Florianópolis, Fevereiro de 2016

Expressing Temporized Properties in Ladder Diagrams

Mathieu Granzotto

Esta monografia foi julgada no contexto da disciplina

DAS 5511: Projeto de Fim de Curso

e aprovada na sua forma final pelo

Curso de Engenharia de Controle e Automação

Prof. Jean-Marie Farines

Banca Examinadora:

Prof. Jean-Marie Farines
Orientador no Curso

Prof. Max Hering de Queiroz
Avaliador

André Vinícius Rocha Silva
Debatedor

Nicholas Roberto Drabowski
Debatedor

Acknowledgements

To my parents, Claimar and Nathalie, for all their backing and support in every moment.

To my adviser, Professor Jean Marie Farines, for this opportunity.

To my supervisor, Silvano Dal Zilio, for his tutoring and guidance.

To Julia Thum, for all the daily moments of joy, encouragement and tenderness.

To my friend Artur Cook, always ready to discuss the next topic, which provided high doses of novelty during slow lectures.

To my Professors, colleagues and staff from DAS, for making this phase of my life so profitable.

Resumo estendido

Esse trabalho foi realizado dentro do grupo Vertics do LAAS-CNRS, Laboratório de Análise e Arquiteturas de Sistemas, localizado em Toulouse (França). Vertics possui um foco no estudo de métodos formais, em particular model checking, com o objetivo de desenvolver técnicas e ferramentas para aumentar a expressividade e escalabilidade destas. Model checking é uma poderosa ferramenta para tratar de sistemas discretos, capaz de expressar propriedades complexas além de manter a problemática resultante da explosão combinatória de estados sob controle.

Nessa monografia, eu procuro expandir uma ferramenta de formalização de diagramas Ladders, uma linguagem de programação para PLCs, Programadores Lógicos controláveis, amplamente usados em vários setores industrias. A ferramenta, inicialmente concebida no trabalho [1] de Ana Maria Carpes, foi pensada como uma forma de ajudar técnicos e engenheiros treinados em Ladder, mas com pouco, ou nenhum, conhecimento em métodos formais. O objetivo dessa ferramenta é fornecer a estes um método para automatizar a verificação de diagramas Ladder e aumentar a confiabilidade em seus sistemas, em comparação ao simples uso de caso de testes e simulações.

Durante o meu estágio, eu racionalizei a cadeia de ferramentas de verificação Ladder e reimplementei partes da ferramenta. Em particular, eu reimplementei a parte da cadeia de ferramenta responsável por importar diagramas Ladder criados no formato TC6, um formato open-source usado pelo programa PLCOpen editor. PLCOpen foi desenvolvido pela organização Beremiz como plataforma de desenvolvimento unificada para programs de PLC. O resultado de meu trabalho é uma ferramenta mais fácil de usar e mais integrada. Deveras, eu tornei acessível diferentes transformações em uma única ferramenta que é mais fácil de instalar e de manter. A ferramenta, que realiza 3 passos de transformação, tem como linguagem alvo Fiacre. Fiacre, uma linguagem formal criada especialmente para servir de fase intermediaria para definição de modelos formalizados, permite descrever comportamentos sequenciais e paralelos, através de processos e componentes.

Em sequência, eu motivo não só a necessidade de aplicar verificação formal para sistemas, mas como ferramenta para verificar a solidez da ferramenta de verificação em si. Por exemplo, eu uso a noção de observadores para verificar a implementação de *timers*, um Bloco Complexo usado para implementar timeouts. Observadores nada mais são que componentes formais que analisam um modelo, sem que modifiquem a execução deste. O observador descrito nesse trabalho me permitiu descobrir um bug antes não detectado no comportamento temporal do modelo formal gerado para o bloco TON (Timer ON).

Outra contribuição que faço é um método para definir propriedades de programas Ladders baseado em *Matrizes Causa e Efeito* (C&E Matrix), uma notação usada na industria para especificar diagramas Ladder.

Ao final, eu explico algumas limitações da ferramenta e dos frameworks usados, já que o modelo de desenvolvimento, MDE (Model Driven Environment), apesar de facilitar o desenvolvimento de cada passo da transformação isoladamente, não fornece muito suporte quando as diversas partes são conectadas, levando a algumas questões a serem respondidas para desenvolvimento futuro.

Indico também uma deficiência do ferramental de verificação formal de diagramas Ladder: falta uma ferramenta para traduzir os rastros de erros gerados, que não estão em um formato amigável para um usuário com pouco treino em ferramentas formais. Proponho então uma ferramenta de pós processamento do rastro de erros.

Eu concluo que a ampliação de especificações básicas que um programa Ladder, através de observadores, são não somente uma maneira de garantir a validade da ferramenta, mas um caminho para desenvolver uma linguagem de alto nível para a definição de especificações formais de programas Ladder.

Palavras-chave: model-checking, formal methods, PLC, Ladder Diagrams, MDE.

Abstract

This work was performed within the VERTICS group, at LAAS-CNRS. This research group is actively seeking improvements to current tooling in formal methods, in particular based on model checking. My objective during this internship was to expand a tool for the formal verification of Ladder Diagrams. This tool, originally created by Ana Maria Carpes [1], was developed with the intent to help technicians and engineers that are trained on Ladder programming but that have no, or limited knowledge, of formal methods. The goal is to provide them with a way to automatize the validation of Ladder diagrams and to increase the trust on their system compared to the simple use of test cases and simulations.

During my internship, I have streamlined the Ladder verification tool-chain and reimplemented parts of the tool. In particular, I have reimplemented the part of the tool-chain responsible for importing Ladder diagrams created using the TC6 exchange format. The end result is a tool that is easier to use and that is better integrated. Indeed, I have made accessible different transformations inside a single tool that is easy to install and to maintain. Another contribution of my work is a method for defining formal properties of Ladder programs based on *Cause and Effect Matrices* (C&E Matrix), a notation used in the industry for expressing the specification of Ladder diagrams.

In this report, I motivate the need to apply formal verification to systems but also as a tool to check the soundness of the verification tool itself. For instance, I use the notion of observers to check the implementation of *timers*, a complex block in Ladder diagrams that can be used to model timeouts. With this work, I was able to uncover an undetected bug on the temporal behaviour of the formal models generated with our tools.

Keywords: model-checking, formal methods, PLC, Ladder Diagrams, MDE.

List of Figures

Contents

1 Introduction

This report covers the use of formal verification techniques for checking the behavior of Ladder programs. Ladder diagram is a graphical notation used to represent logical sequential control of industrial equipments and processes. It is used in many industrial domains, such as the oil and gas industries, for example, or for programming railway equipments. More recently, Ladder logic has evolved into a programming language where graphical diagrams are used to program Programmable Logic Controllers (PLC). The Ladder notation is therefore used both to express the logical part of a manufacturing process and to develop software for programmable logic controllers. This notation is widely adopted and is still in use today. One of the main reason for its success is certainly the fact that it provides a notation very close to relay diagrams that can be easily understood by maintenance electricians and plant engineers. Ladder diagrams brought the flexibility of software while keeping the familiarity sought by technicians and engineers. On the other hand, the notation is quite far from what computer system engineers and programmers are used to work with. Therefore work needs to be done to adapt the technologies developed for software verification to the context of Ladder programs.

Ladder is a logical sequencing language that has been standardized by the International Electrotechnical Commission (IEC) in its open international standard for programmable logic controllers (IEC 61131-3). We give an example of Ladder diagrams in Fig. 1. The name is based on the observation that diagrams look like ladders, with two main vertical rails (with the left rail being the control power hot wire, and the right rail the control power neutral) and a series of horizontal rungs between them. Each rung can be interpreted as a rule that the coil must abide. The Ladder notation provides several operators that can be used inside a rung: *contacts*, $\dashv\ \vdash$, which can be used to access the values of internal variables and the system inputs; *coils*, or actuators $-(\ \ )-$, that writes the result of evaluating a rung to a variable or to a system's output; *complex blocks*, like timers for example, that can be used to model timeouts; ... Finally, the contacts in a rung can be combined together with logical AND and OR operators.

Even with a notation as simple as this, it is very difficult to understand the expected behaviour of a Ladder program when its complexity grows. Indeed, multiple problems can occur. For instance, different rungs may make competing assignment to the same variable, then overshadowing previous modifications. We can also mention problems related to circular definitions of output variables, that may produce an unstable output even when the inputs of the system are fixed. A problem called *race condition* detection in Ladder terminology. Moreover, each new input variable may increase exponentially the number of possible states of the system; which means that traditional techniques based on manual

testing and simulations are impractical.

In this context, it can be useful to use more powerful techniques, such as formal verification methods. With the help of formal methods, in particular model-checking, one could guarantee a set of properties at once for all possible states of a system. During my internship at LAAS, I have worked on an automatic translation from Ladder programs into formal models that can be used by the model-checker Tina [2]. More precisely, I have worked on a tool-chain that generates Fiacre specifications from a given Ladder diagram. This work is a continuation of a previous PFC [1], performed by Ana Maria Carpes in 2012, where she developed a first version of the translation.

My goal was to extend the current tools in order to add the possibility to define temporal properties on the resulting models. In the context of Ladder logic, properties are often expressed using *Cause and Effect Matrices* (C&E Matrix). Therefore it is interesting to be able to translate these matrices into a formalism that can be accepted by the same model-checker that is used to check the Ladder specification. A Cause and Effect Matrix is a 2D table that relates a cause, the inputs (rows) of the matrix, to a list of expected effects of the PLC, the outputs (columns). We give a simple example of C&E matrix in Fig. 9. In this report (see Sect. 5) we describe how to perform this translation for the tools and models used in our tool-chain. More generally, during my internship, I have streamlined the Ladder verification tool-chain and reimplemented parts of the tool. In particular, I have reimplemented the part of the tool-chain responsible for importing Ladder diagrams created using the TC6 exchange format. The end result is a tool that is easier to use and that is better integrated. Indeed, I have made accessible different transformations inside a single tool that is easy to install and to maintain.

The present report is a synthesis of the knowledge acquired and the tooling developed during my stay at LAAS.

1.1 Context of my Internship

This project was realized within the VERTICS group at LAAS-CNRS, in Toulouse (France). The research topics of the VERTICS team are related with critical systems that have strong requirements in terms of temporal constraints. The design of such systems consists in defining and realizing software components characterized by strong temporal and cooperative properties. Formal description techniques, relying on mathematical bases, offer automatic verification at the specification level.

Concerning my course curriculum, my project is related to multiple classes in my engineering course of control and automation from UFSC, principally: *Modelagem e Controle de Sistemas e Eventos Discretos* (DAS5203), for the introduction in formal methods, and *Sistemas de Automação Discreta* (DAS 5307), for the introduction in PLC

systems and Ladder Diagrams.

1.2 Structure of the report

The report is structured as follows. In Chapter 2, I give an overview of the tools and theories that have been used in my work. The following chapters list the contributions I have made during my internship:

- Chapter 3: A streamlined and integrated tool for translating Ladder Diagrams in TC6 format to Fiacre specifications
- Chapter 4: An observer to verify the temporal behaviour of *timers* in Ladder Diagrams.
- Chapter 4.2: A correction for a previously undiscovered bug in the temporal behaviour of the model for *timers*
- Chapter 5: A method for defining formal properties from C&E Matrix

2 Technical Background

In this chapter we expose prior information needed for this project, with an introduction for the topics and tool used or closely tied. Figure 3 provides a quick overview of where which piece of technology is used in this project.

2.1 PLC - Programmable Logic Controller

2.1.1 Ladder Diagrams

Ladder Diagram is a graphical programming language that is to be executed by a PLC. Inspired by relay logic, it saw widespread adoption by the industry since technicians could work on a digital paradigm, with greater flexibility and scalability, while maintaining previous expertise.

It is formed by multiple 'rungs' (or lines), with each line having a composition of relays (represented by ⊣ ⊢) and ending on a coil (represented by –( )–). The relays reads the variable associated with it, and relays can be mixed in a serial or parallel fashion, representing an AND or OR operations. The coil writes to the associated variable.

The execution of ladder program is similar to a REPL (Read Eval Print Loop): all the inputs are read at the start of a cycle, and then each rung is executed sequentially, left to right, top to bottom. The modification of a variable by the rung is seen immediately by the next rung, but the result is only outputted at the end of the cycle.

To increase flexibility, multiple vendors saw the need of adding complex blocks with special purpose, for example timers and counters.

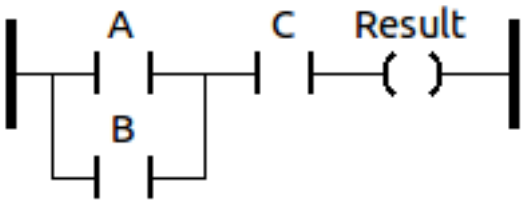

Figure 1 – A Ladder Diagram

2.1.2 IEC 61131-3

To increase compatibility and support for PLC between vendors, the IEC 61131 industry standard was created. The third part, is the one responsible to describe the programming languages to be used on a PLC.

2.1.3 PLCOpen & TC6

One shortcoming of the standard is that while it defines what a ladder diagram is and how to execute, it doesn't describe a file format, how a ladder diagram should be stored. As an attempt to fix this issue and provide a common development framework across vendors, the XML TC6 open format was conceived by PLCOpen Organization[1], which include all the necessary information to be used by a PLC editor. Another organization, Beremiz[2], built an open source editor for PLC programs, called PLCOpen editor. I use PLCOpen to create the ladder diagrams for this work, and the TC6 XML file is the starting point of the transformation tool-chain.

2.2 Formal Methods

Brute force made elegant, Formal Methods is a set of techniques to explore all possible states of a system, all the while keeping a compact and query-able structure of such states. For a proper introduction to model-checking, I refer the reader to [3].

2.2.1 Fiacre Language

Fiacre [4] is an expressive formal language for model checking, where one is able to build systems out of stand alone processes that communicate through ports or shared variables. It was developed as an intermediate language for compiling high-level system descriptions to a fundamental formal format, Time Transition Systems, TTS.

Fiacre is based on two constructs. A process: which represent sequential behaviour with states, transactions guarded by variables or communication events, assignments and timed events. A component: which represent the parallel behaviour between components or process, able to define ports and variables to be shared.

Fiacre also supports declarations of properties and assert directives.For example, LTL (Linear Temporal Logic) formulas can be specified directly in Fiacre.

Code 2.1 is an example Fiacre Process, representing a Ladder processing cycle of a PLC. It features the reading stage (line 8–9), the Ladder rung execution (starting at line 14), and the writing stage (line 10–11). A final stage (line 12–13) is present to indicate that

[1] http://www.plcopen.org/
[2] http://beremiz.org/

the next cycle happens at another time-step; Code 2.2 is a Fiacre Component, featuring a composition of the PLC with a plant.

```
process Scan
    [portInputs: in arrayIn, portOutputs: out arrayOut]
    is
    states initial, writing, final, rung_1
    var varsIn: arrayIn
    init
       to initial
    from initial
       portInputs? varsIn; to rung_1
    from writing
       portOutputs! [s1];  to final
    from final
       wait[1,1];          to initial
    from rung_1            to writing
```

Code 2.1 – Example Fiacre Process

```
component wTON
    is
    port portInputs : in out  arrayIn in [0,0],
         portOutputs: in out arrayOut in [0,0]
    par * in
        PLC   [portInputs,portOutputs]
    ||  Plant [portInputs,portOutputs]
    end
wTON
```

Code 2.2 – Example Fiacre Component

The point of entry, a 'main', of Fiacre specifications is a top-level component with no ports.

2.2.2 Vertics Tools

- Frac - The Fiacre compiler, generating TTS
- Tina - Construct the reachability graph from the compiled TTS
- Selt - LTL Model checker.
- Play - Stepper simulator. Highly used for debugging purposes.

The work-flow (Figure 2) of a final Fiacre specification is to compile it with Frac, and use Tina and Selt to automatically verify properties that are specified in Fiacre. Play can be used on the reachability graph generated by Tina, thus by having a Fiacre translation of your system, not only you enable model checking, but you have a simulator for free. An example for the necessary steps is in section 3.2.

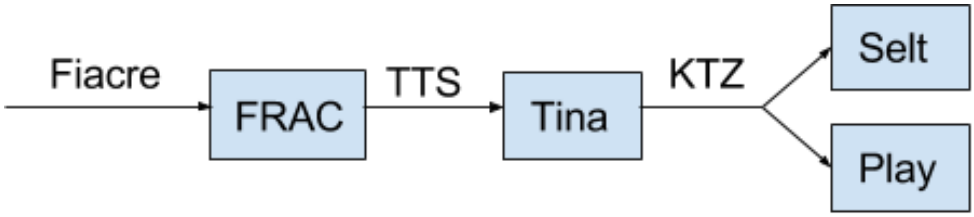

Figure 2 – Typical work-flow of a Fiacre specification

2.3 Eclipse Modeling Framework

Based on java and eclipse environment Eclipse Modeling Framework[3], EMF, supplies an SDK for Model Driven Engineering (MDE) development. MDE is a software development paradigm built upon the strengths of Oriented Object paradigm.

The core power of MDE is to provide automated assistance to software design with well defined model of a problem domain. With the assistance of those tools, a programmer can work with a higher level of abstractions, based on models, and leaving implementation detail for the tool. By explicitly defining your problem domain in a model level abstraction, it becomes less error-prone to maintain and modify code that interfaces other systems.

For the tool explored in this project we use three use cases for MDE development, each one provided by a different tool. Class Generation, provided by EMF basic classes, Model Translation, from ATL, and Model Backtracking, from Xtext.

By being closely integrated in Eclipse, the development of models happens inside a proven, widely used, production environment that any java developer is familiar too.

2.3.1 Ecore

Ecore is the heart of the EMF. Is the standard meta model accepted for the tools provided by EMF, providing constructs very similar to UML's class diagrams.

An Ecore model is a set of EPackages (Objects without methods), their Eattributes and relationships.

[3] https://eclipse.org/modeling/emf/

By having an Ecore model of your data, multitude of tools can be constructed upon. The EMF can automatically generate a GenModel of an Ecore model, which can then generate functional java classes to be used in development, as well as parsers for XML files that conforms to your model.

Ecore is used in this project as a meta model for the models of TC6, Ladder and Fiacre. Data that is conform to such a model is an instance of the model.

The transformation step in section 3.1.2 uses Ecore for generating java classes for both TC6 and ladder data, as well as parsing and unparsing.

2.3.2 ATL

ATL[4], Atlas Transformation Language, is a toolkit specialized in model-to-model transformation. It is a specialized language to describe a source-to-target mapping. A program is a set of rules that defines how the source model are composed and mapped to the target model. ATL dispose of many additional facility, such as defining helper functions and global query of the source model, to enable the construction of complex mappings. Rules can be programmed in a declarative or imperative manner.

The core transformation step (section 3.1.2) uses ATL to map Ladder models to Fiacre models.

2.3.3 Xtext

Xtext[5] is a framework for Language development. By defining a grammar, a full compiler stack is provided by Xtext, including parser, linker, type-checker, compiler, coupled with integration to an editor, like eclipse. Xtext is also compatible with EMF. That is: from a grammar definition, Xtext is able to provide an Ecore model and an unparser.

That means that any instance of this Ecore model can be translated to text, automatically. This is used for transformation step in section 3.1.3.

[4] http://www.eclipse.org/atl/
[5] https://eclipse.org/Xtext/

3 Transformation Toolchain description

In this chapter we present the translation tool, with an explanation of the translations steps and its usage with other Formal Methods tools.

3.1 TC6 → Fiacre

By starting by the format provided by PLCOpen, TC6, the developed tool is able to delivery a standard Fiacre text file, compatible with FRAC. A stand alone tool was created to integrate the intermediary steps, so an eventual user has no need to understand or install the multiples frameworks involved. In Figure 3 an overview for the translation pipeline is given.

3.1.1 TC6 → LD

The first step of the transformation is to provide a compact and descriptive version of a ladder program, to facilitate the following steps. TC6, the format used by PLCOpen, is too complex to be translated directly into Fiacre: not only it has extraneous data, like project details, a ladder program is represented by a graph of weakly connected graphical elements, starting from the rightmost element (the right power rail).

While this format is flexible, it is also too permissible a valid graph is not a valid ladder program, the core aspects of ladder, logical OR and AND, are hidden in the graph structure. Thus, before translating to a Fiacre specification directly, it is best to have data available in a more beneficial abstraction, that by itself would ensure that the ladder program is valid and data can be readily accessed. This is the core concept of the MDE methodology.

Here, I implemented this transformation with aid of EMF's code generators and

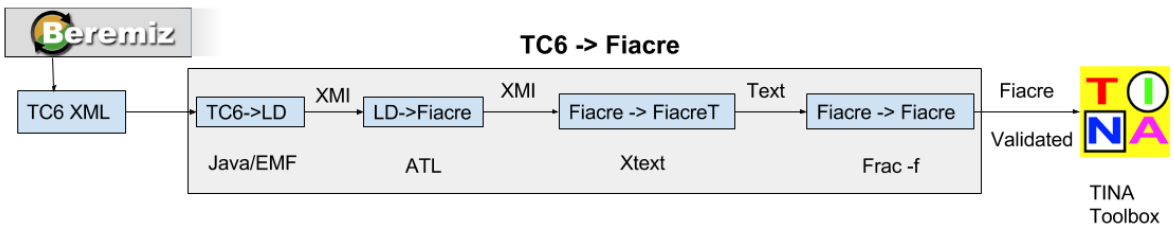

Figure 3 – The tool pipeline

some mapping logic. With EMF, I generated java classes for both the source model, TC6, and the target model, Ladder. The generated classes provides TC6 parsing (reading from a XML) and unparsing to Ladder (saving in XMI). XMI, XML Metadata Interchange is an XML with structural data that conforms to an Ecore model, in this case the Ladder Ecore. The EMF provides assistance for the translation by guaranteeing that any generated file from an instance model is a valid representation of its Ecore model.

For the mapping, I created an algorithm to traverse the TC6 tree data and reconstructed a valid Ecore Ladder representation of the data. The traversal happens from the right power rail to the left power rail of the Ladder Diagram, because of how connections are described in the TC6 format. During the traversal multiple metada from the Ladder Diagram are saved to be later used by EMF to generate the XMI file. The algorithm is able to capture block properties, like negation, set and reset. The EMF XMI generator, together with the Ladder Ecore model, is able to reject certain malformed programs, like in sequence coils or lacking target variables.

3.1.2 LD → Fiacre XMI

With a comprehensible format for a ladder diagram, the next step is to reconstruct the data with Fiacre constructs. This is done by ATL, a 'model-to-model' toolkit. This step was developed in the previous work [1] from Ana Maria Carpes.

The core concept of this transformation is in two parts: one is the generation of the specification of a PLC scanning a Ladder program, creating a process similar to Code 2.1, with the associated internal variables and rung-by-rung execution of logical expressions, and the generation of the exterior ambient that interacts with the PLC, similar to Code 2.2.

The transformation also creates timers as a separated process from the Scan. The timer communicates with the Scan process through two ports, one for the timer input and one for the timer output. All timer composed with Scan form the component PLC.

The executing environment, called Plant, is a simple dummy system that can write variables freely, with no interactions from the output of the PLC. The component generated uses some layering, called InputGlue and OutputGlue, so that the interaction of the PLC is done on centralized ports, thus helping code generation that limits the number of possible states.

For a more in depth explanation of this step, refer to [1] and [5].

I've made some improvements in this step, so the format from the previous transformation to be accepted, and by removing bugs created by edge cases, like the presence of Complex Blocks (like Timer) in certain programs.

3.1.3 Fiacre XMI → Fiacre

Fiacre XMI is still not the end step, since it's not a valid input for the FRAC compiler. For this last step, we use the work of F. Zalila [6], which developed an Xtext Grammar for Fiacre. With this grammar, Xtext is able to resolve an XMI file and backtrack a textual format for Fiacre.

Since the original work of F. Zalila target Fiacre *, a dialect of Fiacre, I lightly modified the Xtext grammar so that the generated text is a valid Fiacre file accepted by Frac.

3.1.4 Fiacre → Pretty Printed Fiacre

Although the last step should be a valid (minus bugs) Fiacre file, this can only be guaranteed if the generated Fiacre file can pass the parsing and type-checker phase of Frac. By running the generated Fiacre file against Frac with flag -f, the tool is able to give instant feedback of the success of the transformation.

An added bonus of this step is that the processed file is Pretty Printed Fiacre, with better readability compared to the Fiacre generated by Xtext.

I've included this step on the translation tool as it is much helpful for verifying the generated Fiacre file. The program is able to search Frac in expected locations in a portable manner, for both windows and linux. A configuration file can be created by the user in case Frac is in another Path.

3.2 An example

Finally, after the introduction of the tool-chain, we show a basic work-flow, starting by PLCOpen and ending in Play.

We start with a basic simple program in Ladder with a timer, made in PLCOpen:

#	Nome	Class	Tipo	Location
1	a1	Input	BOOL	
2	s1	Output	BOOL	

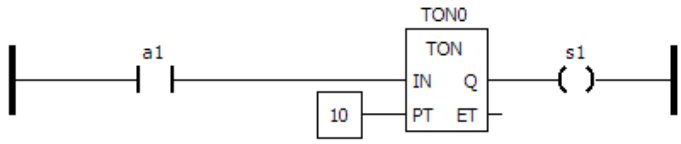

Figure 4 – A Ladder Diagram example

And run the translating tool, plus the following steps required by Tina toolkit:

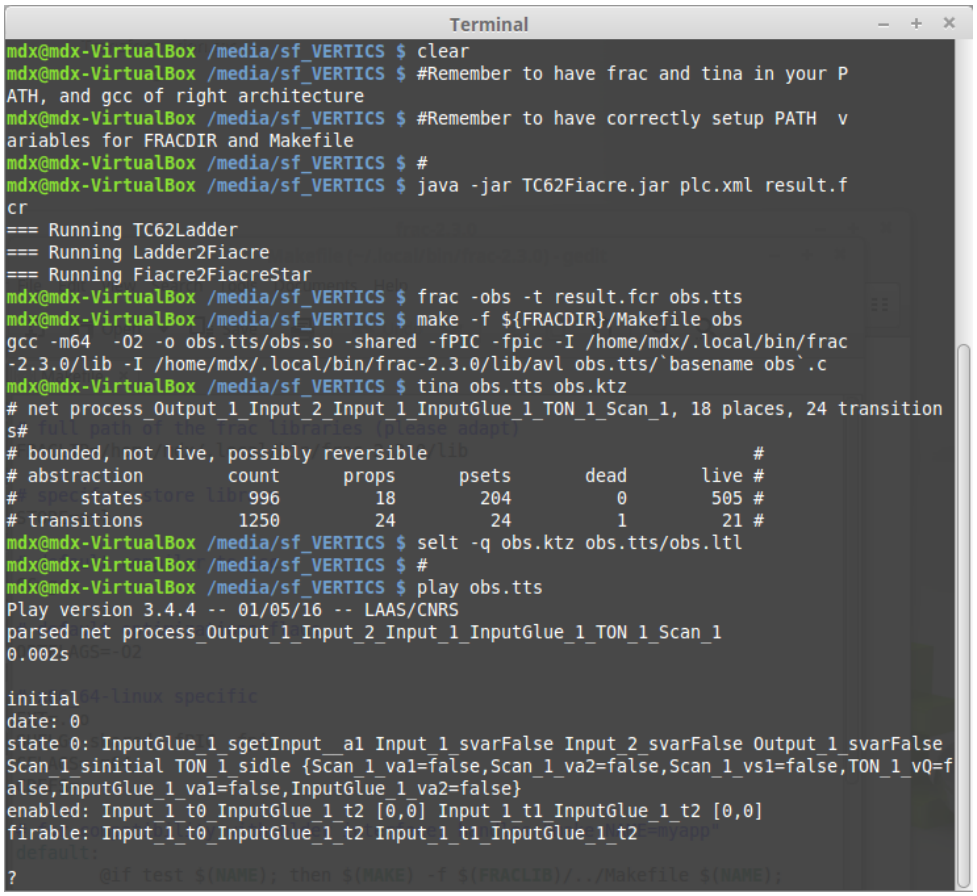

Figure 5 – The complete work-flow

All intermediate files generated may be accessed by the user, in case the tool requires debugging. In Code 3.1 we have the generated Scan process of this example. The reader may find a full Fiacre specification in appendix C.

```
process Scan
    [portInputs: in arrayIn, portOutputs: out arrayOut,
     portTON0_IN: out bool , portTON0_Q: in bool]
    is
    states initial, writing, final, rung_1, rung_11
    var varsIn: arrayIn,
        a1: bool := false,
        s1: bool := false,
        TON0_IN: bool := false,
        TON0_Q: bool := false
    init
       to initial
    from initial
       portInputs? varsIn;
       a1 := varsIn[0];
       to rung_1
    from writing
       portOutputs! [s1];
       to final
    from final
       wait [1,1];
       to initial
    from rung_1
       TON0_IN := a1;
       portTON0_IN! TON0_IN;
       to rung_11
    from rung_11
       portTON0_Q? TON0_Q;
       s1 := TON0_Q;
       to writing
```

Code 3.1 – Generated Scan

4 Checking Timed Temporal Properties on the Ladder TON Block

Providing a translation tool is not enough. It is imperative to be able to verify if the final formal representation of the ladder program is valid, i.e: it matches its run-time behaviour when running inside a PLC. A key point to be tested is the behaviour of the TON block, which has additional complexity for using communication ports on top timed transactions. In particular, my goal is to verify the temporal aspect of the generated formal model. This model was proposed by Ana Maria Carpes during her PFC [1].

To perform this test, an observer is well placed. An observer is nothing more than another component running in parallel of the observed system, while keeping track of the evolution of said system. By defining in the observer what is a valid, or invalid, sequence of states for the main system program, we can then assert an LTL (Linear Time Logic) property over the observer, for example "Is observer state DetectedFailure reachable?".

The attentive reader must have noticed: we are using model checking to verify our application of model checking. This is appealing, since it resembles unit testing: what is verified is a property of a ladder program, not the implementation of this particular translation tool. As the tool evolves, the semantics of the Fiacre filed generated can be verified against those building properties.

An observer is based on two parts. The first is a Fiacre process which does the job of keeping track, the second is the way observers receives information from the main system component: probes. A probe is an observable event of the system, for example a process instance leaving a specified state. A basic probes is built on the path of the instantiated process that leads to such event (i.e: 'main/1/state start'). Probes can also be built upon other probes to leverage more complex events.

An observer is shown to be correct when it is innocuous (i.e: does not influences the transactions of the observed system), and valid (i.e: every sequence that violates the observer leads to a detectable state of error).

Next, we show a step by step construction of a correct observer for a timer block of a Ladder Diagram.

4.1 Making an observer

Take the most basic program with a timer,the one in Figure 4. A formal description of a black-box relationship of a1 and s1 is: 'if, and only if, a1 is true for at least N, then s1

is true'. We want an observer that is able to test this property correctly.

The first step is to define a process for the observer. What are the sequence of events that leads to a success, or a failure, of the proposition? It helps to break the behaviour in branches: one for when a1 is true and when a1 is false, that is one for the timer running or not.

When a1 is false, we have one immediate certainty: s1 can't be true, otherwise we have an error. In fact, by having a1 false at t=0, we know s1 can't be true in the future, for at least for N+1 time steps. At t=N+1, we have two options: either we observe s1=true, and observe an error, or s1=false, and this branch concludes successfully. Figure 6 is a Petri Net visualization of this branch, the initial state is *start* and [2,2] is a timed transaction standing for the N+1 time steps elapsed.

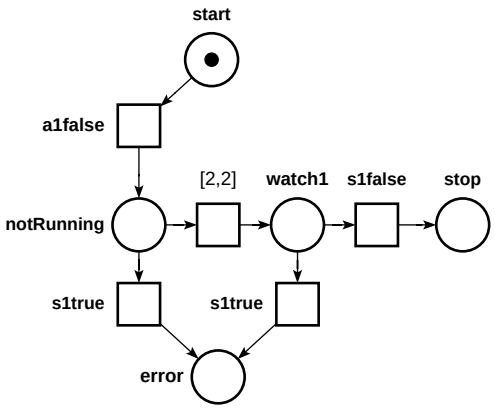

Figure 6 – Timer not running.

When a1 is true, we don't have any immediate knowledge of the output. Only when a1 runs true continuously for N time-steps that we know for certain that s1 must be true. In this case, we run a check if either a1 became false or if the time elapsed. If a1 becomes false, this branch is done - because a1 false is treated by the other branch. If the time elapsed, we check if either s1 is true or false, leading to success or failure. Figure 7 we can visualize this sequence. Note that the labels of the transactions of those Petri nets are guards: they only execute when the condition is satisfied. Later on the final Fiacre observer we show that those guards are implemented by probes.

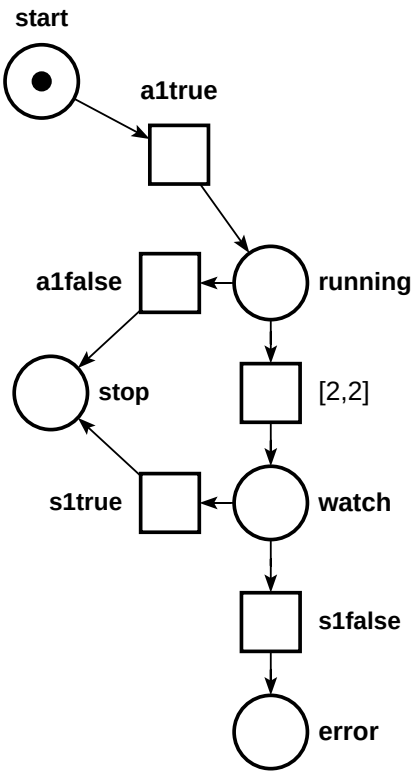

Figure 7 – Timer running.

One last branch remain: universality. If we combine the previous branches into a single observer and stop there, it has a major flaw: it runs immediately, at time zero. This limit coverage, the observer should be able to verify correctness at all possible times. While one can think this requires mixing both of the previous steps in complex ways, it suffices to delay the execution of the observer to any possible time, done by taking a branch that does nothing. In Figure 8 we see that this branch is guarded by states of the PLC cycle, so that while the cycles start-to-idle, the main process observed is not blocked.

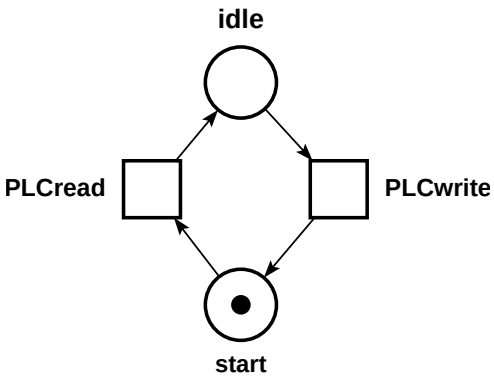

Figure 8 – Observer delayed.

The final Fiacre code for the observer can be found in Appendix A. The process, observer, is a straightforward translation of the above Petri nets, with transactions guarded by synchronizing ports. The component, obs, implements the guards by using probes, which are carefully synced to the appropriate step in the cycle of the PLC: a1, the input, must be tested after the PLC process has read (ports E1 and E2); s1, the output, must be tested after the ladder program is executed, not at the beginning (ports E3 and E5). A priority is enabled on E2 over E5: in Figure 7, it is possible that in state running both transactions are true: at the same time-step a1 became false but the timer elapsed. Since the behaviour of a timer must be deterministic, we use a priority that determines with branch to take. The chosen one implies that if a1 becomes false when the timer elapses, s1 must not become true.

To verify that the property is not violated, one only has to check if the Error state from observer is never marked. The Fiacre LTL (Linear Time Logic) code is :

```
1 property TONsafe is ltl [] not (obs/1/state error)
```

Code 4.1 – TON observer test

4.2 A problem found

During the development of this observer, it became obvious that a bug was present in the Fiacre code generated for the TON block: a sequence of transactions would leave the timer with an inconsistent behaviour. During the cycle of the execution of the PLC, it could sometimes delay the update of his internal state, thus sending the wrong state for the present cycle. In other words, the Ladder Diagram could read true as the input of the timer not in precisely N steps, but in [N,N+1]. The following code block, Code 4.2, from the TON process, generated by the translation tool, is at fault.

```
1    from running
2       select
3          portIN? IN;
4          if not IN then
5             to idle
6          else
7             loop
8          end
9       [] portTimer;
10         Q := true;
11         to elapsed
12      [] portQ! Q;
13         loop
14      end
```

Code 4.2 – TON block state running

The mechanism of this bug and the solution needed were already explained before. It arose when the observer faced a non determinist choice of transactions, which was solved by a priority between the possible transaction. In this case, the problem is that the timer expiring event from *portTimer* could be super-seeded by the *portQ!*, creating the faulty behaviour.

By forcing which transaction must be taken first with a priority, the highlighted line in Code 4.3, the bug vanish and the observer runs without a hitch.

```
1  component PLC
2      [portInputs: in arrayIn, portOutputs: out arrayOut]
3      is
4      port portTON0_IN: in out bool in [0,0],
5           portTON0_Q: in out bool in [0,0],
6           portTON0_Timer: sync in [1,1]
7      priority portTON0_Timer > portTON0_Q
8      par * in
9         Scan [portInputs,portOutputs,portTON0_IN,portTON0_Q]
10     || TON [portTON0_IN,portTON0_Q,portTON0_Timer]
11     end
```

Code 4.3 – TON running fixed

5 Generating Observers from Ladder's Cause and Effect Matrices

The previous method used to verify the transformation tool can be applied in a broader setting, in particular as a way to check that a Ladder diagram respects the specification expressed using Cause and Effect matrices. The idea of using C&E matrix as a source for formal properties was borrowed from T. Prati's work [7]. Here, I introduce a method fitted for the translation tool.

A C&E matrix is a simple relationship from inputs to outputs, as seen in Figure 9. Each cell represent logical implication from cause to effect. For example the first line of Fig. 9 has the form $A \Rightarrow S$, meaning 'S is true if A is true'.

The observer defined in Chapter 4 does not verify the inner implementation of the ladder diagram, or the generated model; indeed the "code running on the PLC" is seen as a black box. The observations made by the Fiacre observer are, on purpose, limited to the outputs and inputs of the program at the reading and writing steps of the PLC cycles. This means that we can easily change the Ladder diagram without modifying the observer. Better yet, we can use the behavior of the observer to answer a different question, namely: does the program implements the semantic tested by this observer ? In other words, we can test the behavior of an unknown, complex Ladder diagram with the same observer that was used to test only the TON block. In this chapter, following the same approach that was used in Chapter 4, we show how to define an observer that can test whether (the Fiacre specification of) a Ladder program is compatible with the behavior described by a given Cause and Effect Matrix.

A simple observer that can be used to check a single relation of a C&E matrix (a

Cause \ Effect	S	T	V	W	...
A	X				
B		X	X		
C	X	X	X	X	
...					...

Figure 9 – A general form for C&E Matrix

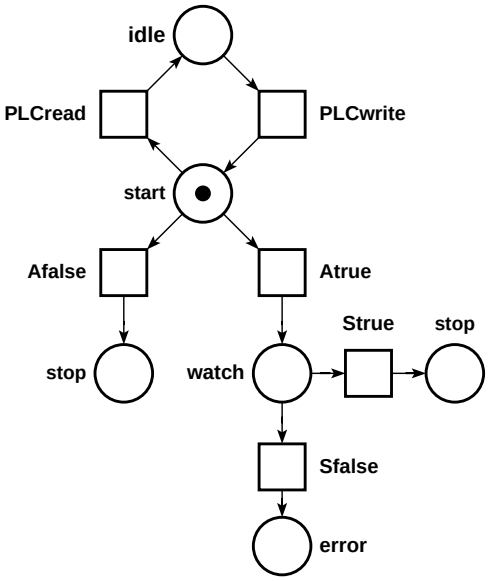

Figure 10 – Observer for $A \Rightarrow S$

cell in the matrix) can be derived following the method from Chapter 4. The resulting Petri net is given in Figure 10, and code in appendix B. In fact, since observers can be freely composed together without introducing unwarranted side-effects, we can test with the same approach the validity of the whole C&E matrix.

If such granularity is not needed, the observers may be generated for each line in place of each cell. The only modification needed is to combine the probes used in state watch: the ones for Strue must be ANDed together, those for Sfalse must be ORed together. A draw back of an observer for the entire line is that one has to infer which column is at fault by searching the error trace.

6 Some Limitations of our Approach

Although we have been able to use our tool successfully in different use cases, there are several limitations. These limitations originate either from the tool itself, the technology used or the theory.

First and foremost, although MDE methodology does indeed facilitates the design and handling of data, it is far from enough. This becomes clear during the integration of all three steps, each one using a different subset of tools from EMF. While they all interface each other with the same Ecore model, it is far too permissive: what is an optional field for one may be a requirement for the other, or vice-versa. That is, the valid generated data for one may be invalid for the other.

Checking for conformity to a model only happens at run-time, with non descriptive errors. This leaves a programmer with no leverage or indication of progress to completely conformity of the generated data. This also means that the transformation tool can fail midway: even if the first two transformation steps are valid, an error in the last step produces no output, not even a partial translation.

Because of the mixture of multiple frameworks, maintainability goes down. Paradigms are widely different, each one with it's own idiosyncrasies. Mixed with the non-exhaustive and run-time check, a programmer must wade through the different frameworks and not reason about the problem at hand.

The tool has multiple design questions to be posed: where does error handling happens ? Or how is it exposed to the user ? Should checks be added at the first step, TC6 to LD ? Should it be spanned across all steps ? Should the tool try to fix malformed ladder programs and generate a partial Fiacre model ?

At the moment the tool provides some checking of input at the very first stage of the transformation, that is, if the line of a rung ends with a single coil and starts on a power rail. Warnings for badly formed ladder programs, lacking definition and label of variables, inclusion of more Complex Blocks into the translations are improvements to be made.

Also, I note that the original use case, to substitute manual testing realized by field technicians, requires a way to backtrack the transformation: giving an error detected by Selt, how can it be exposed to the a user without requiring training in formal methods

At the moment, the error provided is a trace leading to an Error marking. This format is far too dense in information for a quick overview of the problem, as you can observe in Code 6.1

```
state 0: InputGlue_1_sgetInput__a1 Input_1_svarFalse Input_2_svarFalse Output_1_svarFalse
    Scan_1_sinitial TON_1_sidle observer2_1_sstart Input_1_vstates Output_1_vstates

  Firing : observer2_1_t0:
  state 1: InputGlue_1_sgetInput__a1 Input_1_svarFalse Input_2_svarFalse
      Output_1_svarFalse Scan_1_sinitial TON_1_sidle observer2_1_sidle Input_1_vstates
      Output_1_vstates

  Firing : Input_1_t1_InputGlue_1_t2:
  state 1489: InputGlue_1_sgetInput__a2 Input_1_svarTrue Input_2_svarFalse
      Output_1_svarFalse Scan_1_sinitial TON_1_sidle observer2_1_sidle InputGlue_1_va1
      Output_1_vstates

  (...)
```

Code 6.1 – Error Trace from SELT

Also, it would be helpful to have an overview of the Input/Output state of the Ladder program when we want to look at some example (timed) traces of its execution. A graphical compact form, like seen in Code 6.2, could be achieved by post-processing the trace generated by the model-checker (or the player). This can be done by filtering out intermediary states, that is, keeping only states where Scan_1_rung1 or Scan_1_initial are marked, and reading the associated marking for the input and output states. This tool would be a step into creating a model checking toolkit that can be used by a technician with limited knowledge in formal methods.

```
#Inputs
a1 _-_----------
a2 _____----------
.
.
.
#outputs
s1 _____-----------
s2 _____----------
.
.
.
```

Code 6.2 – A suggested temporal PLC trace representing the state (low or high) of the input and output for each execution cycle

7 Conclusion and Perspective

During my internship, I have implemented a streamlined and integrated tool for translating Ladder Diagrams into Fiacre specifications. I have also defined a Fiacre specification for the timer blocks (TON), used in Ladder, as a mean to check the correctness of part of our transformation. This work helped me find a problem with the temporal semantics of generated models. Finally, I have used the notion of observers as a mean to model the properties that need to be checked on a Ladder programs.

Personally, this work gave me the occasion to learn technologies, like Model-Driven Engineering development, with EMF, Eclipse and Java, or Domain Specific Language (DSL) technologies with Xtext. After gaining familiarity with Frac and Tina, I developed a better understanding and became more proficient in formal methods.

Also, the usefulness of model-checking became evident throughout my work: not only does it provide a way to automatically check properties, it is also a powerful paradigm to model and study the dynamic behavior of systems. I have also found a lot of interest in the possibility to use a component-based approach, with the ability to weave together simple processes to create a more complex system, and to reuse the same model both for verification and for simulation.

While the tool itself can be improved over the limitations of the previous chapter, at this point I believe to be more beneficial to explore the user experience of tools for the resulting fiacre model. For example, an user should develop a more intuitive understanding of the resulting Fiacre specification. if handed a special purpose Play for Ladder, or a post processed temporal trace like the one in Chapter 6.

Bibliography

1 CARPES, A. M. M. *Properties of LD Programs: Expression and Verification.* Dissertação (Projeto Final de Curso) — Universidade Federal de Santa Catarina, 2015. Cited 5 times in pages 5, 7, 14, 24, and 29.

2 BERTHOMIEU, B.; VERNADAT, F. Time petri nets analysis with tina. In: *Proceedings of the 3rd International Conference on the Quantitative Evaluation of Systems*. Washington, DC, USA: IEEE Computer Society, 2006. (QEST '06), p. 123–124. ISBN 0-7695-2665-9. Disponível em: <http://dx.doi.org/10.1109/QEST.2006.56>. Cited in page 14.

3 CLARKE JR., E. M.; GRUMBERG, O.; PELED, D. A. *Model Checking*. Cambridge, MA, USA: MIT Press, 1999. ISBN 0-262-03270-8. Cited in page 18.

4 BERTHOMIEU, B. et al. Fiacre: an Intermediate Language for Model Verification in the Topcased Environment. In: *ERTS 2008*. Toulouse, France: [s.n.], 2008. Disponível em: <https://hal.inria.fr/inria-00262442>. Cited in page 18.

5 FARINES, J. et al. A model-driven engineering approach to formal verification of plc programs. In: *Emerging Technologies Factory Automation (ETFA), 2011 IEEE 16th Conference on*. [S.l.: s.n.], 2011. p. 1–8. ISSN 1946-0740. Cited in page 24.

6 ZALILA, F. *Methods and tools for the integration of formal verification in domain-specific languages*. Tese (Doutorado), 2014. Thèse de doctorat dirigée par Aït-Ameur, Yamine et Crégut, Xavier Sûreté de Logiciel et Calcul à Haute Performance Toulouse, INPT 2014. Disponível em: <http://www.theses.fr/2014INPT0092>. Cited in page 25.

7 PRATI, T.; FARINES, J.; QUEIROZ, M. de. Automatic test of safety specifications for {PLC} programs in the oil and gas industry. *IFAC-PapersOnLine*, v. 48, n. 6, p. 27 – 32, 2015. ISSN 2405-8963. 2nd {IFAC} Workshop on Automatic Control in Offshore Oil and Gas Production {OOGP} 2015Florianópolis, Brazil, 27–29 May 2015. Disponível em: <http://www.sciencedirect.com/science/article/pii/S2405896315008721>. Cited in page 33.

Appendix

APPENDIX A – Timer Observer

```
process observer [S0,S1,E1,E2,E3,E4,E5: sync] is
    states start, idle, run, notrun,
           watch, watch2, error, stop
    from start
    select
       S0; to idle
    [] E2; to notrun
    [] E1; to run
    end
    from idle
    S1; to start
    from notrun
       select
           E5; to watch2
    [] E3; to error
       end
    from watch2
    select
       E3; to error
    [] E4; to stop
end
    from run
    select
       E5; to watch
    [] E2; to stop
    end
    from watch
    select
       E3; to stop
    [] E4; to error
    end
```

```

component obs is
    port S0: sync in [0,0] is (wTON/1/1/state rung_1),
         S1: sync in [0,0] is (wTON/1/1/state initial),
         E1: sync in [0,0] is (wTON/2/1/2/state varTrue)
                           and (wTON/1/1/state rung_1),
         E2: sync in [0,0] is (wTON/2/1/2/state varFalse)
                           and (wTON/1/1/state rung_1),
         E3: sync in [0,0] is (wTON/2/2/1/state  varTrue)
                           and (wTON/1/1/state  initial),
         E4: sync in [0,0] is (wTON/2/2/1/state varFalse)
                           and (wTON/1/1/state  initial),
         E5: sync in [2,2]

    priority E2>E5
    par
       observer[S0,S1,E1,E2,E3,E4,E5]
    end
```

APPENDIX B – Cause & Effect Observer

```
process observer [S0,S1,E1,E2,E3,E4: sync] is
    states start, idle, watch, error, stop
    from start
    select
       S0; to idle
    [] E2; to stop
    [] E1; to watch
    end
    from idle
       S1; to start
    from watch
    select
       E3; to stop
    [] E4; to error
    end
```

```
component obs is
    port S0: sync in [0,0] is (aTos/1/1/state rung_1),
         S1: sync in [0,0] is (aTos/1/1/state initial),
         E1: sync in [0,0] is (aTos/2/1/2/state varTrue)  and (aTos/1/1/state rung_1),
         E2: sync in [0,0] is (aTos/2/1/2/state varFalse) and (aTos/1/1/state rung_1),
         E3: sync in [0,0] is (aTos/2/2/1/state  varTrue) and (aTos/1/1/state  initial),
         E4: sync in [0,0] is (aTos/2/2/1/state varFalse) and (aTos/1/1/state  initial),
    par
       observer[S0,S1,E1,E2,E3,E4]
    end
```

APPENDIX C – Generated Fiacre

```
1
2 type indexOut is 0..0
3
4 type arrayIn is array 1 of bool
5
6 type arrayOut is array 1 of bool
7
8 process Input
9     [sendVar: out bool]
10    is
11    states varTrue, varFalse
12    init
13       to varFalse
14    from varFalse
15       select
16          sendVar! false;
17          loop
18       []
19          sendVar! true;
20          to varTrue
21       end
22    from varTrue
23       select
24          sendVar! true;
25          loop
26       []
27          sendVar! false;
28          to varFalse
29       end
30
31 process InputGlue
32    [writeInputs: out arrayIn, syncPort: in arrayOut, a1Port: in bool]
33    is
34    states writing, synchronizing, getInput_a1
35    var irrelevantArray: arrayOut,
36        a1: bool,
37    init
38       to getInput_a1
39    from writing
40       writeInputs! [a1];
41       to synchronizing
42    from synchronizing
43       syncPort? irrelevantArray;
44       to getInput_a1
45    from getInput_a1
46       a1Port? a1;
47       to writing
48
49 process Output
50    [receiveVar: in arrayOut]
51    (arrayIndexVar: indexOut)
52    is
53    states varTrue, varFalse
54    var outputsVarsArray: arrayOut
```

```
55      init
56         to varFalse
57      from varFalse
58         receiveVar? outputsVarsArray;
59         if outputsVarsArray[arrayIndexVar] then
60            to varTrue
61         else
62            loop
63         end
64      from varTrue
65         receiveVar? outputsVarsArray;
66         if outputsVarsArray[arrayIndexVar] then
67            loop
68         else
69            to varFalse
70         end
71
72  process Scan
73      [portInputs: in arrayIn, portOutputs: out arrayOut, portTON0_IN: out bool, portTON0_Q
            : in bool]
74      is
75      states initial, writing, final, rung_1, rung_11
76      var varsIn: arrayIn,
77          a1: bool := false,
78          s1: bool := false,
79          TON0_IN: bool := false,
80          TON0_Q: bool := false
81      init
82         to initial
83      from initial
84         portInputs? varsIn;
85         a1 := varsIn[0];
86         to rung_1
87      from writing
88         portOutputs! [s1];
89         to final
90      from final
91         wait [1,1];
92         to initial
93      from rung_1
94         TON0_IN := a1;
95         portTON0_IN! TON0_IN;
96         to rung_11
97      from rung_11
98         portTON0_Q? TON0_Q;
99         s1 := TON0_Q;
100        to writing
101
102 process TON
103     [portIN: in bool, portQ: out bool, portTimer: sync]
104     is
105     states idle, running, elapsed
106     var IN: bool := false,
107         Q: bool := false
108     init
109        to idle
110     from idle
111        select
112           portIN? IN;
113           if IN then
```

```
114             to running
115          else
116             loop
117          end
118       []
119          portQ! Q;
120          loop
121       end
122    from running
123       select
124          portIN? IN;
125          if not IN then
126             to idle
127          else
128             loop
129          end
130       []
131          portQ! Q;
132          loop
133       []
134          portTimer;
135          Q := true;
136          to elapsed
137        end
138    from elapsed
139       select
140          portIN? IN;
141          if not IN then
142             Q := false;
143             to idle
144          else
145             loop
146          end
147       []
148          portQ! Q;
149          loop
150       end
151
152 component PLC
153    [portInputs: in arrayIn, portOutputs: out arrayOut]
154    is
155    port portTON0_IN: in out bool in [0,0],
156         portTON0_Q: in out bool in [0,0],
157         portTON0_Timer: sync in [10,10]
158    priority  portTON0_Timer>portTON0_Q
159    par * in
160       Scan [portInputs,portOutputs,portTON0_IN,portTON0_Q]
161    || TON [portTON0_IN,portTON0_Q,portTON0_Timer]
162    end
163
164 component Inputs
165    [writeInputs: out arrayIn, readOutputs: in arrayOut]
166    is
167    port a1Port: in out bool in [0,0]
168    par * in
169       InputGlue [writeInputs,readOutputs,a1Port,a2Port]
170    || Input [a1Port]
171    end
172
173 component Outputs
```

```
174     [readOutputs: in arrayOut]
175     is
176     par * in
177        Output [readOutputs] (0)
178     end
179
180 component Plant
181     [writeInputs: out arrayIn, readOutputs: in arrayOut]
182     is
183     par * in
184        Inputs [writeInputs,readOutputs]
185     || Outputs [readOutputs]
186     end
187
188 component wTON
189     is
190     port portInputs: in out arrayIn in [0,0],
191          portOutputs: in out arrayOut in [0,0]
192     par * in
193        PLC [portInputs,portOutputs]
194     || Plant [portInputs,portOutputs]
195     end
196
197 wTON
```