



Fernando Joaquim Ganhão Pereira

Licenciado em Engenharia Eletrotécnica e de Computadores

The DS-Pnet modeling formalism for cyber-physical system development

Dissertação para obtenção do Grau de Doutor em
Engenharia Eletrotécnica e de Computadores

Orientador: Doutor Luís Filipe dos Santos Gomes

Prof. Associado com Agregação

Faculdade de Ciências e Tecnologia / Universidade Nova de Lisboa

Júri:

Presidente: Doutor Luís Manuel Camarinha de Matos

Arguentes: Doutor Ricardo Jorge Silvério de Magalhães Machado
Doutor Arnaldo Silva Rodrigues Oliveira

Vogais: Doutor Luís Filipe dos Santos Gomes
Doutor Carlos Baptista Cardeira
Doutora Anikó Katalin Horváth da Costa

The DS-Pnet modeling formalism for cyber-physical system development

Copyright © Fernando Joaquim Ganhão Pereira, Faculdade de Ciências e Tecnologia, Universidade Nova de Lisboa.

A Faculdade de Ciências e Tecnologia e a Universidade Nova de Lisboa têm o direito, perpétuo e sem limites geográficos, de arquivar e publicar esta dissertação através de exemplares impressos reproduzidos em papel ou de forma digital, ou por qualquer outro meio conhecido ou que venha a ser inventado, e de a divulgar através de repositórios científicos e de admitir a sua cópia e distribuição com objectivos educacionais ou de investigação, não comerciais, desde que seja dado crédito ao autor e editor.

ACKNOWLEDGEMENTS

I want to express my gratitude to Prof. Luís Gomes, my supervisor, for his support, guidance, knowledge, and friendship.

I would like to thank the Thesis Accompanying Committee (CAT), composed of Prof. Ricardo Machado, Prof. Anikó Costa, and my supervisor, for their recommendations.

I would also like to thank "Universidade Nova de Lisboa" (UNL) and "UNINOVA – Institute for the Development of New Technologies", Portugal, in particular to Prof. Adolfo Steiger Garção, Prof. Luis Camarinha-Matos, and Prof. João Goes.

I want to thank my team members, colleagues, and friends, for their contribution and friendship from the GRES research group: Filipe Moutinho, Rogério Campos Rebelo, Anikó Costa, João Paulo Barros, José Ribeiro, José Rocha, José Pedro Lucas, Rui Pais.

I would like to thank my work colleagues, and friends, from INOCAM, CEI/ZIPOR and ISEL for their support, releasing me from many work tasks in order to provide time to proceed with the PhD work. I would like to thank Rui Guerreiro that contributed with suggestions and helped proof read the text.

I would like to thank all my friends for their encouragement and friendship during this time.

Finally, I thank my family for their love!

This work was partially supported by project:

Petri-Rig - A Petri net based framework for embedded systems engineerInG (Petri-Rig - Ambiente de desenvolvimento de sistemas embutidos baseado em redes de Petri); Maio 2013 – Outubro 2015; Consórcio: UNINOVA, Inst. Politécnico de Beja; FCT sponsored; Funding: 109.663€; Ref.: PTDC/EEI-AUT/2641/2012 (<http://gres.uninova.pt/petri-rig/>);

Resumo

Este trabalho apresenta o formalismo de modelação DS-Pnet (Dataflow, Sinais e redes de Petri), criado para o desenvolvimento de sistemas ciber-físicos, combinando as características das redes de Petri e dataflows para possibilitar a modelação de sistemas mistos, contendo partes reativas e operações de processamento de dados. Herdando as potencialidades da classe de redes de Petri progenitora IOPT, incluindo a interface externa composta por sinais e eventos de entrada e saída, a adição de operações de fluxo de dados (dataflow) contribuiu para melhorar a capacidade de modelação para especificar a transformação matemática de dados e expressar graficamente as dependências entre sinais. Sistemas centrados em dados, que não requerem controladores reativos podem ser modelados usando apenas dataflows.

A composição de modelos baseada em componentes permite reutilizar componentes previamente desenvolvidos, criar bibliotecas de componentes e decompor hierarquicamente modelos em diversos sub-sistemas.

Foi definida uma semântica de execução precisa, tendo em conta a relação entre nós de dataflow e rede de Petri, que oferece uma abstração para definir a interface entre controladores reativos e sinais de entrada e saída, incluindo sensores e atuadores analógicos.

O novo formalismo é suportado por um conjunto de ferramentas com interface Web, IOPT-Flow, que oferece ferramentas para criar e editar modelos, simular a execução de modelos diretamente no navegador Web, para além de ferramentas de validação de modelos e geração automática de código (C, VHDL e JavaScript) que produzem hardware e software para correr em dispositivos computacionais embutidos.

Foi criado um novo protocolo de comunicação para automatizar a implementação de sistemas ciber-físicos distribuídos compostos por redes de componentes remotos que comunicam usando a Internet. A ferramenta de edição pode ser ligada diretamente a dispositivos embutidos remotos que executam modelos DS-Pnet, permitindo importar componentes remotos para novos modelos, contribuindo para simplificar a criação de aplicações distribuídas onde a comunicação entre componentes localizados em nós diferentes é especificada pelo desenho de arcos.

São apresentadas várias aplicações que foram elaboradas para validar o formalismo proposto e as ferramentas associadas, incluindo soluções implementadas em hardware, aplicações industriais e aplicações de software distribuídas.

Palavras-chave: Redes de Petri, fluxo de dados, sistemas ciber-físicos, sistemas embutidos, automação de design

Abstract

This work presents the DS-Pnet modeling formalism (Dataflow, Signals and Petri nets), designed for the development of cyber-physical systems, combining the characteristics of Petri nets and dataflows to support the modeling of mixed systems containing both reactive parts and data processing operations. Inheriting the features of the parent IOPT Petri net class, including an external interface composed of input and output signals and events, the addition of dataflow operations brings enhanced modeling capabilities to specify mathematical data transformations and graphically express the dependencies between signals. Data-centric systems, that do not require reactive controllers, are designed using pure dataflow models.

Component based model composition enables reusing existing components, create libraries of previously tested components and hierarchically decompose complex systems into smaller sub-systems.

A precise execution semantics was defined, considering the relationship between dataflow and Petri net nodes, providing an abstraction to define the interface between reactive controllers and input and output signals, including analog sensors and actuators. The new formalism is supported by the IOPT-Flow Web based tool framework, offering tools to design and edit models, simulate model execution on the Web browser, plus model-checking and software/hardware automatic code generation tools to implement controllers running on embedded devices (C, VHDL and JavaScript).

A new communication protocol was created to permit the automatic implementation of distributed cyber-physical systems composed of networks of remote components communicating over the Internet. The editor tool connects directly to remote embedded devices running DS-Pnet models and may import remote components into new models, contributing to simplify the creation of distributed cyber-physical applications, where the communication between distributed components is specified just by drawing arcs.

Several application examples were designed to validate the proposed formalism and the associated framework, ranging from hardware solutions, industrial applications to distributed software applications.

Keywords: Petri nets, dataflows, cyber-physical systems, embedded systems, design automation

Table of Contents

1	Introduction.....	1
1.1	Background and motivation.....	1
1.2	Preliminary contributions.....	5
1.3	Research questions.....	7
1.4	Research method.....	9
1.5	Overview of the IOPT-Flow framework.....	9
1.6	Contributions and publications.....	18
1.7	Document structure.....	20
2	Literature Review.....	21
2.1	Petri nets.....	21
2.2	Model checking.....	22
2.3	Execution semantics and non-autonomous properties.....	23
2.4	Low level and high level net classes.....	25
2.5	High level net execution strategies.....	26
2.6	Model composition and hierarchical structuring.....	27
2.7	Model composition based in signal and event communication.....	28
2.8	The IEC61499 standard.....	30
2.9	Automatic code generation.....	31
2.10	UML statecharts and activity diagrams.....	32
2.11	Model file formats.....	33
2.12	Reactive systems and synchronous dataflows.....	34
2.13	Matlab/Simulink.....	35
2.14	Cyber-physical systems.....	36
3	The DS-PNET Modeling Formalism.....	39
3.1	Language core.....	39
3.2	Dataflow operations.....	43
3.3	Components.....	46
3.4	Example DS-Pnet model.....	49
3.5	Model files.....	52
3.6	Execution Semantics.....	54
3.6.1	Formal definition.....	54
3.6.2	Execution semantic rules.....	57
4	Automatic Code Generation.....	63
4.1	JavaScript generated code.....	67
4.2	VHDL Generated code.....	68
4.3	C Generated code.....	70
4.4	Interface board for industrial applications.....	74
4.5	External/Foreign Components.....	76
5	Distributed DS-Pnet Models.....	79
5.1	Shared distributed components.....	87
5.2	JSON/HTTP Communication Protocol.....	91
5.2.1	User authentication and privilege levels.....	95
5.2.2	Request types.....	96
5.2.3	Server.....	100
5.2.4	Client.....	101
6	The IOPT-Flow Tool Framework.....	105
6.1	Editor.....	107
6.2	The Simulator tool.....	114

6.3 Remote Debugger.....	117
6.4 Node-Split.....	119
6.5 Automatic code generation.....	120
6.6 Import and export IOPT models.....	121
6.7 IOPT Model Checking.....	122
6.8 Component Library.....	125
6.9 Standard foreign component library.....	128
6.9.1 Arrays.....	129
6.9.2 Data file input and output.....	130
6.9.3 System time information.....	131
6.9.4 Random number generator.....	132
6.9.5 Graphical user interface.....	133
6.9.6 Audio samples.....	135
6.9.7 Industrial ModBUS Gateway.....	136
6.10 Debug And Model-Checking.....	138
6.10.1 Application example.....	141
7 Validation Applications.....	147
7.1 Bushless servo motor controller.....	148
7.1.1 Model development.....	149
7.1.2 Prototype implementation:.....	155
7.1.3 Results.....	155
7.2 Distributed multi-user game with graphical interface.....	158
7.2.1 Results.....	162
7.3 Graphical console for an industrial variable speed drive.....	165
7.3.1 Results.....	169
7.4 Distributed cyber-physical system simple application.....	171
7.4.1 Results.....	173
8 Conclusions and future work.....	175
8.1 Research question 1.....	175
8.2 Research question 2.....	178
8.3 Research question 3.....	180
8.4 Results and comparison with other technologies.....	181
8.4.1 Traditional programming languages (C/C++, Java, Python, VHDL, etc.):.....	182
8.4.2 IOPT-Tools.....	184
8.4.3 Industrial automation development languages.....	185
8.4.4 Labview and Matlab/Simulink.....	186
8.5 Future work.....	187
9 References.....	191

Index of figures

Fig. 1: Ladder diagram "emulation" model.....	10
Fig. 2: Petri net example. Places drawn as yellow circles, transitions as blue bars and arcs as arrows.....	22
Fig. 3: Incidence matrix.....	23
Fig. 4: High level Petri net example.....	25
Fig. 5: Component.....	47
Fig. 6: Example DS-Pnet model.....	49
Fig. 7: Anchor equivalence: the operation on the left is equivalent to the dataflow node on the right.....	55
Fig. 8: Micro-step and nano-step sequence numbers.....	60
Fig. 9: Automatic code generation information flow (steps 1-5).....	64
Fig. 10: A UART model (top) with the receiver(left) and sender (right) component implementation models.....	70
Fig. 11: Isolated digital I/O board w/ SPI interface (2 boards).....	75
Fig. 12: (Not)Synchronous Channel on a distributed model.....	80
Fig. 13: One publisher and multiple subscribers.....	82
Fig. 14: Client/server event driven communication.....	83
Fig. 15: Client-server communication.....	84
Fig. 16: Example: Event based communication with remote component.....	85
Fig. 17: Same example with the handshake controller Petri net encapsulated in a local component (on the right).....	86
Fig. 18: Proposed infrastructure for concurrent client access.....	89
Fig. 19: Possible CPS network topology example.....	91
Fig. 20: The IOPT-Flow Editor.....	106
Fig. 21: Editor interaction/feedback loop.....	108
Fig. 22: The Expression Editor.....	110
Fig. 23: The operation input/output editor.....	111
Fig. 24: Clipboard View.....	112
Fig. 25: View component implementation model.....	112
Fig. 26: Import DS-Pnet components from remote embedded nodes.....	113
Fig. 27: The IOPT-Flow simulator (Chrome Web browser).....	114
Fig. 28: Waveform view window (simulation history).....	115
Fig. 29 The IOPT-Flow remote debugger application (Chrome browser).....	117
Fig. 30: Code generation options.....	120
Fig. 31: IOPT Petri net view.....	122
Fig. 32: State space generation progress window.....	123
Fig. 33: The query editor (IOPT model checking).....	123
Fig. 34: A state-space graph of an IOPT model extracted from a DS-Pnet.....	124
Fig. 35: IOPT-Flow Editor tool - Library dialog (user interface widgets folder).....	125
Fig. 36: Ladder-logic library components specified as dataflow operations.....	126
Fig. 37: The t_ON timer and Up/Down counter native components. Component interfaces (left) and implementation models (right).....	128
Fig. 38: Foreign array components: vector and matrix.....	129
Fig. 39: The file input / output foreign components.....	130
Fig. 40: Using time-stamps to synchronize the position of 3 motors.....	132
Fig. 41: Reference position synchronization on the remote side.....	132

Fig. 42: Graphical user interface components.....	134
Fig. 43: User interface test application.....	135
Fig. 44: ModBUS component test model.....	136
Fig. 45: ModBUS + UI motor control application running on a Raspberry-PI 2 card, with an LCD+touchscreen hat and an USB-RS485 serial converter.....	137
Fig. 46: Concrete mixer plant.....	141
Fig. 47: Cement mixer controller model.....	142
Fig. 48: Cement mixer main model: controller + plant.....	144
Fig. 49: Cement mixer plant model.....	144
Fig. 50: Closed-loop servo motor controller top-model.....	148
Fig. 51: Quadrature encoder model.....	149
Fig. 52: Digital PID Controller model.....	150
Fig. 53: The diff_in(left), diff_out(center) and nfilter(right) models.....	151
Fig. 54: The PWM generator model (left) and the cntr_up_dn component model (right)	152
Fig. 55: The BLDC Commutation table model.....	153
Fig. 56: The Speed/Position selector model.....	154
Fig. 57: Prototype diagram on the left and photo on right: BLDC Motor, FPGA and Inverter boards.....	154
Fig. 58: Distributed dual-user «pong» game (graphical user interface).....	158
Fig. 59: The entire single-user game model fits in a single editor page.....	159
Fig. 60: The main game model: game engine + player1 interface.....	160
Fig. 61: The player2 model.....	160
Fig. 62: The game engine component model.....	161
Fig. 63: Game user interface.....	162
Fig. 64: Variable speed drive console application (running on a laptop PC).....	165
Fig. 65: Console main model.....	166
Fig. 66: ModBUS Scan-cycle model.....	167
Fig. 67: The console user interface component.....	169
Fig. 68: Example of a distributed DS-Pnet application model.....	171
Fig. 69: Implementation models of the IOX8 (left) and Kit (right) components.....	172
Fig. 70: Sub-models after node-split: Main maestro model(top), NodeA (bot. Left) and NodeB (bot. right).....	173

Index of Tables

Table 1: Petri net elements.....	40
Table 2: Dataflow nodes.....	41
Table 3: Transition firing inhibition constructs.....	42
Table 4: Expression operators.....	44
Table 5: C code generator output files.....	72
Table 6: Privilege levels.....	95
Table 7: Communication protocol request/procedure list.....	98
Table 8: Editor toolbox functions.....	109
Table 9: Component classes used in the application.....	149

1 Introduction

1.1 Background and motivation

The emergence of low cost computing platforms lead to the vast proliferation of embedded systems with increasing levels of sophistication, automating many tasks that were previously performed by human operators, with applications in the domains of industrial systems, home appliances, medical devices, automatic vending machines, security and surveillance applications, in-vehicle systems and entertainment applications, among others. The fast dissemination of the Internet and the wide availability of inexpensive networking technology brought Internet connectivity to the recent generations of embedded devices, contributing to the birth of the Internet of Things.

Applications running on mobile computing devices may be employed for the remote monitoring and operation of solutions employing distributed networks of remote devices, often taking advantage of public data provided by existing infrastructure, as smart grids and city traffic control systems. These capabilities enable the development of even more sophisticated systems, including access to automatic payment systems and connection to social media platforms.

Over the past decades, model based formalisms have been successfully used to the development of embedded system controllers, helping to cope with the increasing levels of complexity involved. With this approach, instead of directly writing software code or hardware descriptions, developers start with the design of high level models that specify the desired system behavior and data structures employed, frequently based on graphical formalisms as Petri nets [29][30][31][32][33], UML activity diagrams and statecharts [34][35][36].

With thousands of academic publications, Petri nets have been the focus of many research groups leading to the advent of a growing number of Petri net classes adapted to different fields. However, most of these classes only support autonomous systems and are only used for simulation and model-checking purposes [37][38]. In contrast, non-autonomous classes, as IOPT nets [29] and NCES [39], use inputs and outputs to communicate with the external world, going beyond the realm of simulation to allow the implementation of real controllers running on physical hardware [8][14].

Non autonomous Petri nets offer a feature set very well adapted to the design of embedded system controllers. System state can be mapped to places and the behavioral rules that define system evolution are specified using transitions. Petri nets inherently handle the concepts of parallelism, concurrency and synchronization, frequently used in embedded system controller design. The design of systems containing multiple sub-systems that compete for shared resources usually starts with the definition of independent state machines for each sub-system, without concurrency concerns. Next, the critical sections that require exclusive access to the shared resources are synchronized with additional places that work as semaphores. This method avoids possible state explosion problems that would occur trying to design entire systems using a single state machine.

To perform their job, controllers must communicate with the controlled systems and the external world, involving the transmission of different types of information, including sensor gauged data, drive mechanical actuators and communicate with users. As a result, the external interface of the non autonomous models must cover a wide range of data types to support both digital and analog signals and events. The controller models react to these events and changes in input signals, producing the consequent output responses. However, the values read from the input sensors usually require some sort of signal processing and conditioning before being ready for decision making. For example, input signals may require units conversion, noise filtering and threshold cross checking. In the same way, output values are generally calculated with basis on the system state and input values.

Although Petri nets excel in the specification of reactive systems, most Petri net classes have struggled to provide a good solution for signal processing and data manipulation. The most frequent solution relies on text inscriptions associated with places and transitions where the user may insert mathematical expressions [29] or code snippets written using the syntax of traditional programming languages [40][41][42]. Unfortunately, this solution cripples the core advantages of a graphical formalism: the relationships and dependencies between different signals are hidden inside textual code

expressions, frequently hidden from the main view to avoid screen clutter, contributing to reduce model readability.

High level Petri nets offer an interesting data processing solution, storing data inside tokens, that may be manipulated upon transition firing. Unfortunately this solution has several drawbacks. First, when a model employs interdependent calculations it imposes propagation delays, as new calculated values will only be available on the next execution step. Second, as a single place may store multiple tokens, it conducts to iterative execution semantic algorithms that increase the complexity of hardware implementation and do not guarantee a fixed step-execution time.

From another side, the functionality of the embedded systems has been constantly improving, leading to more complex controller models, raising the need for structuring mechanisms that enable the sub-division of complex controllers into several components. Using this strategy, a controller can be composed from an hierarchy of sub-systems and the behavior of each sub-system can be specified using simpler models. Component models may be individually tested and model-checked, with benefits in terms of development time. Finally, the components can be instantiated multiple times in the same project, or reused on future projects, taking advantage of the design and model-checking effort previously carried, allowing the creation of libraries containing frequently used components.

Almost all modeling formalisms, programming languages and hardware description languages include structuring mechanisms to enable the top-down decomposition of complex systems into simpler components [36][43][44], or the bottom-up composition of new applications from existing components. In all cases, the mechanisms used to pass information between components and encapsulate local data inside each component play a crucial role.

Most of the traditional Petri net classes and associated tool frameworks [40][41][45] offer structuring mechanisms, employing concepts as node fusion and macro nodes. However these mechanisms present several drawbacks relatively to the input and output signals used in electronic devices, where information flows in a unidirectional way. In these circumstances, the behavior of a system depends only on the inputs and internal state, and is not affected by external systems connected to the outputs. In contrast, the traditional Petri net structuring mechanisms permit adding external input arcs to the output nodes of module, inducing effects in the internal module behavior that conduct to results not foreseen by the original model designer. For example, an external arc may prevent a transition from firing, completely blocking a module execution. This

problem has already been presented by other authors [39][46] discussing the NCES Petri net class.

Modern embedded systems, built on top of networks combining computational sub-systems and physical devices, enter in the field of Cyber-Physical Systems (CPS) [47][48], dealing not only with classic control problems and the idiosyncrasies of communication networks and computational systems, but specifically with the problems that arise at the intersection between physical and computational sub-systems. Applications of CPS frequently listed in the literature, include distributed industrial systems, smart electrical grids, in-vehicle systems and traffic control systems, bio-medical and health-care systems, smart sensor networks and industrial robot systems, using the same network infrastructure that is usually employed in the Internet of Things (IoT) [49].

Involving both mechanical sub-systems and computational devices, Cyber-Physical Systems are viewed as an interdisciplinary field, requiring the collaborative work from different engineering disciplines, including mechanics, computer science, computer engineering, systems engineering and electronics. These disciplines approach problems from different perspectives, use different terminologies and employ different tools, raising the need for new development formalisms that may appeal to designers coming from different backgrounds. Again, model based development formalisms may be used as a common ground: high level graphical models may be used to create information systems and specify system behavior in a way that is easily understood by all people involved, hiding the low level details required by the traditional programming languages used in embedded-system design.

Around the time this work was started, the IEEE Control systems society had recently identified a list of areas in need for CPS research [50], including the need for new abstractions and architectures [51][52][53], distributed computation models and verification and validation tools, to support the rapid development of CPS applications. The work presented in this document is a contribution in that direction.

1.2 Preliminary contributions

The problems addressed in this work were identified along several years of research and development in related fields. This work resulted in several contributions to the IOPT tools framework, available on-line at «<http://gres.uninova.pt/IOPT-Tools/>», whose results were disseminated over 24 publications, including conference and journal papers, two book chapters and a user manual.

Contributions to a first-generation of IOPT support tools, include an automatic generator of debug screens for the Animator tool [3] and a DDR memory interface to support Animator graphical user interfaces on FPGA platforms [2][4].

Contributions to the IOPT Petri net class have been added to the current IOPT meta-model [5][17] descriptions, include new syntax rules for mathematical expressions, the addition of output actions associated with transition firing and definition of arrays:

a) Changes in mathematical expressions include the support for new operators and the definition of a new hierarchical syntax, to support multiple automatic code generators in a language independent way.

b) Transition actions allow the definition of output signal values using arithmetic expressions. transition output signals memorize the last affected value and are a part of the system state vector, along with place marking and signals associated with output events. In order to simplify state-space computation, the expressions used to calculate transition output signals can only contain literal values and other system-state variables.

c) Arrays are used for two purposes: First, constant arrays enable the definition of general purpose functions with one or two integer arguments, storing a table of pre-calculated function values, that can be used by both the software code generators and the hardware description code generators. Second, variable arrays, whose contents can change during model execution, enable the application of the IOPT formalism to problems that deal with large amounts of data. In order to support hardware implementations, arrays indexes are always performed using a single range variable. As a consequence, simultaneous concurrent access to different array positions must be explicitly dealt by the model designer.

Contribution to several prototypes of the cloud based IOPT tools framework [13] [16][20][21][23], including contributions to the IOPT Tools editor [24], the C code generator [8], the VHDL code generator [14], the state-space [6][7] and model-checking

subsystem [9][15], a simulator [19] and a debugger based on a remote debug and monitoring communication protocol [18][22].

Four application papers, describing FPGA based prototypes in the field of industrial electronics, consisting of a controller for a high-voltage Marx pulse generator [1][10][11] and a brush-less DC motor controller [12], played an important role in the identification of the research questions and underlying problems described in this text:

a) The controllers implemented in both prototypes presented a relatively high level of complexity and a modular approach was employed in each case. The final systems were built using the composition of smaller components, communicating with each other and the external world using input and output signals. In both cases, the component instantiation and signal connections was performed by manually writing VHDL code.

b) In both prototypes, several modules were first designed in paper using a dataflow approach and were manually translated to the chosen development language: direct VHDL in the first case and IOPT models in the second. These dataflows were even employed in the resulting papers as a simplified graphical description of some of the components (PWM generator, etc.) and as a diagram to depict the entire systems.

The work presented in this document is focused around the DS-Pnet (Dataflows, Signals and Petri nets) modeling formalism and the associated IOPT-Flow tool framework [26][27]. DS-Pnets combine the characteristics of the IOPT Petri net class with dataflows and model composition based on components. Dataflow nodes are used to specify mathematical operations and the dependencies between signals in a graphical way, replacing the expressions that were previously inserted into place and transition annotations.

As both the DS-Pnet modeling formalism and the IOPT-Flow tool chain inherit the results of the preliminary contributions, these contributions cannot be dissociated from the final results presented in this document. Using a simplistic approach, the DS-Pnet formalism can be viewed as the union of IOPTnets, Dataflows and model composition based on components. This way, DS-Pnets incorporate all preliminary contributions to the IOPTnet class and the design of the new tool-chain benefited from the previous experience and knowledge acquired during the development of the parent IOPT-Tools framework. Although most code was rewritten, the new tools were designed using similar algorithms and design patterns, including the editor, simulator and remote debugger.

Finally, it is important to recall that the contributions described in this section were based on previous work that started on the Uninova/CTS GRES research group, with the definition of the IOPT class [29][54] and the creation of the first-generation support tools, including a version of the Snoopy Petri net editor [55] with support for the IOPT class, an automatic C code generation tool [56][57], a VHDL code generation tool [58], an Animator tool [59] for interactive user interface design, with support for VHDL hardware implementations [60] and a Split tool [61] to support distributed execution [62].

1.3 Research questions

Based on the problems identified during the preliminary work phase, the following research questions were formulated:

Research question 1

Which modeling formalisms can be used in association with Petri nets to support the design of cyber-physical systems, including both the control logic and data operations ?

Hypothesis

a) Cyber-physical systems and embedded systems can be designed through the composition of multiple components, or function blocks, connected through input and output signals and events. The individual function blocks can be designed using IOPT nets, a low-level Petri net class designed for embedded system controller development.

b) The addition of a complementary modeling formalism to define data structures and mathematical operations, used in synergy with the Petri nets, enables the definition of complete embedded systems, including the control logic (Petri nets) and data. An higher level synchronous dataflow, describing a network of mathematical operations applied to input signals, output signals, internal signals and system state variables, can be used to define the data part of the embedded systems.

Research question 2

Which syntax rules and execution semantics must obey the complete systems, composed of multiple function blocks containing control logic (Petri nets) and data parts (dataflow) to ensure deterministic execution on a) monolithic implementations and b) distributed environments ?

Hypothesis

a) By composing the entire systems into a flat model containing all function blocks, it is possible to identify global loops inside the synchronous dataflow Petri net nodes, that would prevent deterministic execution.

b) A set of syntactic and semantic rules to regulate the bidirectional interaction between the Petri net nodes and dataflow components must be defined.

c) The loops identified in a) can be broken by inserting registered internal signals. These registered internal signals may be used to define part of the dataflow system state vector.

d) Syntactic and Semantic rules must be applied to the external interface of function blocks to enable execution correctness in distributed environments.

e) Analyze the advantages of possible automatic code generation strategies based on individual components or on flat models of the entire system, according to the target architecture (software or hardware).

Research question 3

How to model check the systems described in R.Q. 2 ?

Hypothesis

a) The construction of the state-space graph of the individual function blocks and the complete systems, enables model checking and property verification. The definition of the execution semantics and the identification of all system state variables is enough to allow state-space computation.

b) The relationships between the state-space of the entire system and state-space of individual function blocks must be studied, in order to find expedite ways to perform model checking and verify certain system properties.

1.4 Research method

In order to answer the previous research questions and verify the hypothesis, the following steps were executed:

- 1 – During a preliminary phase, contributing to the IOPT tools framework, knowledge about the state of the art was acquired, permitting the identification of gaps and unsolved problems
- 2 – Formulate research questions based on the identified problems
- 3 – Elaborate a set of hypothesis to answer the research questions
- 4 – According to the formulated hypothesis, create a new development formalism and study the respective execution semantic rules
- 5 – Create a set of support tools to enable the application the new formalism to the design of cyber-physical systems
- 6 - Define criteria to compare validation application results with other development technologies
- 6 – Using the tools created in 5, design a set of validation applications
- 8 – Analyze results and validate the hypothesis

1.5 Overview of the IOPT-Flow framework

This section presents an overview of the DS-Pnet modeling formalism and the associated IOPT-Flow tool framework, addressessing the goals that lead to the creation of the new tools and potential areas of application.

The DS-Pnet (Dataflow, Signals and Petri nets) modeling formalism [26] was designed to support the creation of distributed Cyber-Physical systems. Based on a combination of Petri nets and dataflows, it supports the design of mixed systems containing both data-processing and reactive parts. Derived from the parent IOPT net class [29], it enables the use of low level Petri nets to design the state machines employed by CPS and embedded controllers, taking advantage of the well known properties of Petri nets, with good support for concurrency and synchronization and availability of diverse model-checking tools. DS-Pnets inherit the concept of input and output signals and events from IOPT nets and the respective data types, required to create non-autonomous models that communicate with the physical world. All concepts

of the parent IOPT Petri net class may be mapped into DS-Pnets and an automatic translation tool has been created.

Composed of signals and events, the external interface of a DS-Pnet model may be used to read sensors, manipulate actuators or communicate with other DS-Pnet models, under the form of components, that may be placed locally or distributed on remote locations, enabling the creation of CPS applications over networks of distributed components.

In order to appeal to a wider base of users coming from different engineering backgrounds, the text inscriptions that traditionally have been used by other Petri net dialects to specify processing instructions, were replaced by graphical dataflows. Dataflow languages have been used in varied areas of engineering, and popular prototyping software packages as Matlab and Simulink [63] offer dataflow functionality. Regarding industrial automation, the most popular graphical languages used on programmable logic controllers, Ladder diagram [64] and Grafset [65], can be emulated respectively using dataflows and Petri nets. Developers coming from an industrial automation background may use a library «Ladder» folder containing dataflow operations implementing traditional Ladder constructs. Figure 1 presents a DS-Pnet model using dataflow operations to emulate Ladder contacts and a «T-On» timer component frequently used in Ladder diagrams.

Dataflow graphs offer several advantages, presenting a graphical representation of the dependencies between input, output and intermediate signals that contribute to improve model readability. Like functional programming languages, it restricts the number of signal value assignments to a single expression, contributing to reduce modeling mistakes. By employing a synchronous execution paradigm, assuming that all

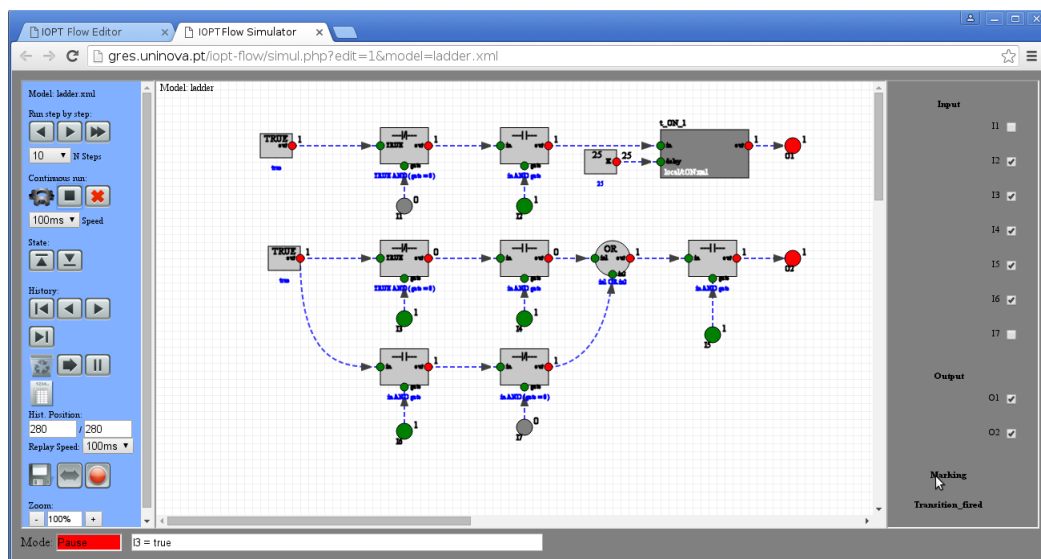


Fig. 1: Ladder diagram "emulation" model

mathematical operations are executed instantaneously, it allows signal propagation through multiple internal computational nodes in a single execution step.

The dataflow part of DS-Pnet model is composed of arcs and operations. Data flow nodes, called operations, are used to perform data processing by applying mathematical transformations to input data and producing one or more results. Arcs are used to connect signals between different nodes, including input and output signals and events, Petri net place and transition nodes, dataflow operations and components.

In order to simplify the creation of complex models, DS-Pnets support model composition based on components. Any DS-Pnet model may be used as a component to create higher level applications. In the same way as DS-Pnet models, the external interface of components is composed of input and output signals and events. Components may be used as building blocks to compose high level applications, chosen from libraries of existing components, or used for the top-down decomposition of complex systems into simpler sub-systems. As each component may contain internal data, under the form of internal signals and Petri net state variables, and processing instructions implemented using transitions and dataflow operations, components can be viewed as objects where method execution is triggered by input events, or as actors communicating with each other using input and output signals [66].

A DS-Pnet component may be native or foreign. Native components are designed using DS-Pnet models. Foreign components are used to encapsulate external sub-systems designed using other modeling formalisms and development languages. Foreign components are created using empty DS-P models, containing just the input and output signals and events that define the component interface, and selecting the «foreign» target implementation property. When the automatic code generation tools find this property, they will create a set of data structures and stub functions where the developer may insert code to initialize and execute the components.

On hardware projects, foreign components permit using existing integrated circuits and IP modules defined using hardware description languages into DS-Pnet applications. On software projects, foreign components permit using external code software inside DS-Pnet applications, including existing algorithms developed using standard programming languages, access any resources provided by computer operating systems, and communicate with legacy embedded platforms.

Components may run locally or remotely. Remote components provide an abstraction to permit the rapid development of distributed Cyber-physical systems. The controllers built using the C code generated automatically contain a minimalist HTTP server implementing a JSON/HTTP protocol for remote debug, monitoring and

operation. This way, the model edition tools can connect directly to the controllers running on the embedded computing devices to request the list of available components and download the respective DS-Pnet models. Remote components may be inserted into the new application models and used in the same way as local components. The connection between local and remote components is performed using dataflow arcs. This way, the operation of reading remote sensors or driving remote actuators is as simple as drawing arcs (after importing components from remote servers to the new application model), and all communication details are dealt by the automatic C code generation tool.

Remote components can be used to create an abstraction for physical devices, including sensors, actuators, motors and entire mechanical systems, greatly simplifying the creation of CPS applications. Remote foreign components may be also used to interface with legacy industrial devices, including programmable logic controllers, variable speed drives and numeric controlled machinery. To assist the integration of DS-Pnet applications in industrial environments, a foreign component implementing a gateway for the ModBUS [67] industrial field-bus was developed, enabling the communication and control of almost all industrial automation devices present on the market.

In order to ensure deterministic operation, the execution semantics of DS-Pnet models was analyzed, studying the bi-directional relationship between dataflow Petri net nodes. Dataflow operations may read the system state under the form of place marking and events triggered by transition firing. The evolution of the Petri net part of the models is conditioned by transition guards, transition input events and synchronous channels between transitions, defined using dataflow arcs that end at the respective transitions. A set of executions rules was defined, used as a basis for the automatic code generation tools.

The execution semantics of distributed execution of Cyber-Physical Systems composed by networks of remote components presents a different level of problems. In addition to the usual concurrency problems presented by parallel execution architectures and differences in performance between nodes running on heterogeneous hardware platforms, the choice of the Internet as a communication medium brings new concerns, including variable network latency delays, unpredictable network bandwidth and possible data loss.

These problems were solved using an approach borrowed from the IEC61499 international standard for distributed control systems [68][69]. In both cases, DS-Pnet components and IEC61499 function-blocks, the communication between distributed

modules is performed using signals and events. However, the IEC61499 function blocks usually communicate over local networks using industry standard field-buses and offer two types of function blocks for Internet communication: publishers/subscribers and master/slave [70]. In contrast, the proposed DS-Pnet communication protocol is based on HTTP and does not enforce any usage patterns, employing the same approach for both local nets and long distance Internet connections.

Four forms of remote component users were typified, the observer, the client, the master and the administrator, with details presented in the corresponding chapter. Observers just subscribe changes from remote component output signals, for example to monitor sensors or the internal state of remote sub-systems. Multiple observers may subscribe the same values without conflicts. In contrast, all other usage types may suffer from conflicts and synchronization problems.

Events play a critical role to manage synchronization and concurrency problems. In a typical use case, when an application wants to invoke a certain methods on a remote component, it first passes parameter data through input signals and then sends an event that will trigger an action on the remote side. The remote component might answer immediately with another event to acknowledge the request reception, or might just place an answer on output signals and trigger a completion event.

The responsibility to avoid synchronization errors and ensure the correct behavior lies on both developers, the component designer and application designer. As long as the parameter signals hold the correct values when the events are triggered, the communication middle-ware ensures that these values do not arrive out of order. In case of network problems and communication fails, both the component and applications are informed by setting predefined values on the signals received from the network.

The suitability of the proposed solution depends on each specific application and the type of network employed: global broadband Internet or local intranets with guaranteed network bandwidth. Real-time applications must be executed on local dedicated networks but non time-critical applications may run over the Internet.

In the near future, with the proliferation of the Internet of Things and Cyber-Physical Systems, a wide range of publicly accessible information services will become available. For example, municipalities might publish in quasi-real-time weather information, traffic control information, including data about road semaphores, airports arrival times, civil protection and emergency service information, etc. From another side, trends in industrial information systems, with projects in Industry 4.0 initiative [71], involve the interconnection between manufacturing automation sub-systems to

accounting/management systems, and even the vertical interconnection between the information systems of different partners in the same supply chain.

In all these cases, the entities publishing the data have lost control about the client applications that are using the information for different purposes. As a consequence, the same CPS component may receive simultaneous requests from multiple applications, often with conflicting consequences. In fact, the same component will be shared by multiple CPS applications, that may not be aware of each other. For example, the timing schedule information about a single semaphore might be monitored by multiple CPS applications running on in-vehicle systems or smart-phones from nearby car drivers. In the same way, each driver's CPS application might want to register the local position and desired travel destination on a central traffic control system in order to obtain the best routes and help optimize traffic management. In another example, an hospital might offer a public service to schedule doctor appointments and applications running on the patients smart-phones might try to negotiate available slots according to the patient personal schedule restrictions.

To solve this problem, an extension to the current communication protocol is proposed, allowing multiple CPS applications to share the same remote component in a transparent way. With this solution, each application views the same component as if it was being used exclusively, but the remote server assigns the component to a single application at a time. This is achieved by assigning special properties to the events of the component interface used to initiate and terminate requests. The server will put requests on a queue and store parameter input data from each application. Client users might be assigned different priorities, in order to provide different levels of quality of service.

After the definition of the DS-Pnet formalism and the corresponding execution semantics, a tool framework was created to assist the development of distributed Cyber-physical systems, called IOPT-Flow [27]. The scope of the new tools and formalisms is not restricted to CPS systems and can be employed to develop traditional embedded controllers, general purpose digital circuits, and even to software applications. The IOPT-Flow Web based tool-chain supports all development phases, including model design and edition, simulation, automatic code generation (C, Javascript and VHDL), and remote debug and monitoring.

The front end of tool framework, available at <http://gres.uninova.pt/iopt-flow>, is the model editor, with menu options to invoke all the other tools. In addition to the usual graphical edition functions, creation of Petri net and dataflow nodes, arc drawing, property edition, copy&paste, undo&redo, it offers functions to automate several tasks

that otherwise would require user attention and effort, as the creation of semaphores to lock critical zones and complementary places. For the rapid development of new applications, an hierarchical library of pre-designed components and frequently used dataflow operations is offered.

Users may store models in a public directory, download the model files to the personal computer, or create personal user accounts to store data on private folders. Multiple users working cooperatively can copy selected parts of models to a public clipboard that is shared among users. Other users may import the contents of shared clipboard and paste it into other models.

The simulator tool executes the model being edited directly on the Web browser. It greatly contributes to reduce development time, as a simulation session can start in just a few seconds after a model has been changed. In contrast, testing models on prototype hardware often requires long delays recompiling software and hardware synthesis tools typically take many minutes to generate bit-stream files. In addition, the physical devices employed in Cyber-Physical systems are prone to suffer damages due to controller mistakes, risking to cripple expensive hardware. Thus, the controller models must be extensively tested and well debugged before being executed on the real devices.

The core of the simulator employs the output of the Javascript automatic code generator, that produces new Javascript code for each model, according to the execution semantic rules. Models may be run step-by-step, or continuously, and the user may associate breakpoints to transitions or dataflow operations.

Simulation history is continuously stored and presented as graphical waveforms. For faster debug sessions, users may navigate through the saved history and replay it, or export waveform data in a spreadsheet format. To automate regression tests, users may store waveform data on the server and replay simulations after changing models, with automatic detection of changes in the resulting waveforms.

The automatic code generation sub-system employs a multiple step process. The first step, implemented in the editor, creates a flat model containing the nodes of all component sub-models, analyzes the dependencies between internal signals and define a precise execution sequence by assigning scheduling information to each dataflow node and Petri net transition. The second step produces a programming-language independent XML file containing information about the data-structures and processing instructions required to implement the model semantics. A third step employs XSL transformation to convert the XML document to the syntax of the target programming languages: C, Javascript or VHDL.

In addition to the code responsible for model execution, the output of the C software code generator also includes an optional HTTP server, to support remote debug, monitoring, integration on CPS networks, and the creation of remote user interfaces. When a model employs distributed components, located on remote servers, client code is also added to the project, automatically subscribing and transmitting events and signal changes to the remote components, according to application model topology and the arcs connected to these components, fully automating the design of CPS networks.

The underlying communication protocol, based on JSON over HTTP, was optimized to support CPS applications, using HTTP server side events to minimize latency (and connection keep-alive connections in the future). The HTTP protocol was selected due to the availability of client code for Web based applications (AJAX), the existence of libraries on many programming languages. It also benefits from easy gateway traversing, as most router policies have HTTP ports open by default and proxy services may be employed otherwise. JSON was chosen over XML due to compactness and easy integration on JavaScript applications, to produce Web based front-end user interfaces.

The ability to remotely monitor and debug distributed CPS applications has paramount importance, as components are often located at far-away locations and the computing devices often lack hardware resources to create user interfaces. The IOPT-Flow tool-chain provides a remote debugger/monitor application, with a user interface similar to the simulator, enabling the visualization of the system state in quasi-real-time, pause execution run step-by-step, reset the model state and force input values. When monitoring distributed CPS, implemented as a network of multiple nodes, the user may open several windows, attached to each node, just by directing the Web browser location to the respective node URL.

As previously mentioned, the adoption success of a development language depends as much on the set of associated libraries as on the language intrinsic qualities. The availability of well debugged components suitable for specific tasks can greatly reduce application development time. The IOPT-Flow framework currently offers a small but growing library of components, that were required to implement the sample applications presented in this text. There are folders dedicated to ladder diagram, timers, counters, PWM generators, Boolean logic, arithmetic functions, UARTs, arrays and tables, a ModBUS gateway, graphical user interface widgets, sound playback, random number generators, data file logging and access, etc.

When possible, components should be implemented as DS-Pnet models, using the internal language capabilities. However, certain features require access to external resources and must be coded using foreign/external components. This problem is common to many programming languages that resort to “native” code to implement certain functions. For example, to access operating system resources, including the system date and time information, file-system and database access, communication ports, use the graphical and sound sub-systems and generally to access hardware device-drivers. The implementation of foreign components depends on the target hardware architectures and operating systems employed, posing compatibility issues when porting applications to different hardware. The same problem arises during the creation of new automatic code generators for different programming languages. Hence, the resort to foreign components must be avoided, by identifying a minimal set that supports specific fields of application. This minimal set could then become a «standard» library that must be supported by all code generators and target architectures.

Several example applications were developed using DS-Pnets and the IOPT-Flow toolchain, used to demonstrate, test and validate the proposed concepts: a closed-loop driver for a Brushless DC servo motor, implemented on a FPGA using the VHDL code generated automatically; a simplified «pong» game implemented using C software and the graphical user interface widget components, an industrial application using the modbus protocol to access a legacy programmable logic controller and a distributed CPS industrial application. A master thesis student used the tools to implement a component library with different types of events, including atomic and compound events, defined by sequences of atomic events.

Comparing with traditional development languages, the example applications were created in a much shorter time period, due to the automatic code generation and the ability to hide distribution implementation details. For instance, the game example was completely developed in just a few hours. Communication between distributed nodes is handled transparently, just by connecting arcs to remote components, a task that would require a greater coding effort with traditional programming languages and development tools. As the low level details are hidden, it opens the field of distributed industrial automation and cyber-physical system development to a wider audience, without deep knowledge about computer programming and communication protocols.

Some of the early results have already been published, including a conference paper presenting the DS-Pnet formalism and the respective execution semantic rules, a conference paper presenting the tool framework and a controller-plant example, and journal paper presenting the tools and the servo motor driver application. Additional

publications about the new communication protocol and application to CPS development are planned for the near future. This work inherits many concepts and ideas from a preliminary work on the parent class IOPTnets and the IOPT tools framework. Several publications about the preliminary PhD work have been published during the first years and are listed in a corresponding section.

1.6 Contributions and publications

The following list presents the main contributions resulting from this work:

1 – The DS-Pnet modeling formalism, combining low level Petri net and dataflows to support the design of mixed systems containing reactive parts that evolve according to external events and a data processing part to perform signal processing, data manipulation and deal with analog sensors.

2 – The specification of the DS-Pnet execution semantic rules, used to define a precise evaluation sequence to calculate dataflow operations and transition firing, leading to the elaboration of a deterministic execution algorithm.

3 – Automatic code generation tools based on the results from 2, generating C, JavaScript and VHDL. The code generated automatically may be used to simulate the model execution, employed in the core of state-space calculation tools [7], but mainly to implement real controllers to deploy on embedded hardware and distributed CPS nodes. A multi-step code generation architecture separates semantics from the target language syntax, simplifying the future creation of code generators for different languages.

4 – The IOPT-Flow Web based integrated development environment, supporting the DS-Pnet formalism, including a graphical editor, a simulator with waveform visualization capability and test automation based on previous stored waveforms, a Petri net model checking sub-system, a remote debugger and automatic distributed code generation tools, among others.

5 – A JSON/HTTP communication protocol optimized for distributed cyber-physical system implementation and remote debug and monitoring of DS-Pnet systems. The C code produced by the automatic code generation tool includes a minimalist HTTP server to support the remote debug and monitoring and communication with other CPS nodes. In the same way, the code produced for models that employ remote components contains client networking software to communicate with the remote nodes. As a result, distributed cyber-physical systems may be created just by importing remote components into new models and connecting arcs – the communication code is generated automatically.

6 – A node-split tool to divide models into sub-models that will run on different distributed nodes, supporting workflows that start with the design of a centralized models, that are simulated and debugged before being split into distributed nodes.

7 – Propose a new mechanism to share the same component among multiple distributed applications, with the definition of two new event properties. Component sharing is transparent for the application models, as if they were being exclusively used.

8 – A growing component library containing both native components (implemented as DS-Pnet models) and foreign components created outside of the IOPT-Flow environment. The foreign library components extend the core functionality of DS-Pnets, with the addition of arrays and matrices, file input and output, random numbers, graphical user interface widgets, audio samples and a communication interface with industrial devices based on the ModBUS communication protocol.

9 – Support for foreign components (in both the C and VHDL code generators), permitting the use of virtually any existing software package or hardware device from DS-Pnet models, building an interface composed of signals and events to invoke code written using other formalisms. This way, existing algorithms, object-oriented classes and hardware subsystems may be integrated in distributed cyber-physical systems in a transparent way.

In addition to the publications listed in the preliminary contributions section, covering the IOPT-Tools framework whose results were applied in this work, new results about the DS-Pnets and the IOPT-Flow tool chain have been published:

1 - Pereira, F.; Gomes, L.; “Combining data-flows and petri nets for cyber-physical systems specification”, Technological Innovation for Cyber-Physical Systems - 7th IFIP WG 5.5/SOCOLNET Advanced Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2016, Proceedings. Vol. 470 2016. p. 65-76 (IFIP Advances in Information and Communication Technology; Vol. 470) [26]

2 – Pereira, F.; Gomes, L.; "The IOPT-Flow framework pairing Petri nets and data-flows for embedded controller development", IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, Florence, 2016, pp. 4832-4837. doi: 10.1109/IECON.2016.7794152 [27]

3 - Pereira F.; Gomes, L.; "The IOPT-Flow Modeling Framework Applied to Power Electronics Controllers," in IEEE Transactions on Industrial Electronics, vol. 64, no. 3, pp. 2363-2372, March 2017; doi: 10.1109/TIE.2016.2620101 [28]

Publication 1 presents the DS-Pnets and the respective execution semantic rules, publication 2 covers model-checking DS-Pnet models using a controller-plant strategy and the third covers the first validation application presented on chapter 7.

1.7 Document structure

Chapter 2 contains literature review about related topics.

Chapter 3 presents the DS-Pnet modeling formalism, presenting all types of available nodes, arcs and the respective attributes. Focusing on the graphical aspect of the formalism, it presents typical constructions formed by the combination of dataflow operations and Petri net nodes. This chapter also presents a formal definition of the DS-Pnet formalism and the execution semantic rules, leading to the elaboration of a pseudo-code algorithm to execute DS-Pnet models.

Chapter 4 presents details about the automatic code generation tools and algorithms, used to produce code that execute the model semantics for simulation and deployment on embedded hardware.

Chapter 5 discusses the execution of distributed DS-Pnet models containing remote components, communicating over the internet and presents the underlying JSON/HTTP protocol.

Chapter 6 presents the IOPT-Flow tool framework, containing information about the capabilities of each tool and relevant implementation details. A component library and an incipient “standard” library containing foreign components that bring enhanced functionality to DS-Pnet models. This chapter also discusses the available model-checking options and the usage of controller-plant systems to permit the verification of pertinent system properties, without incurring in state explosion problems.

Chapter 7 presents a set of validation applications and discusses the results obtained, comparing with other development languages and tools.

Finally, chapter 8 presents the conclusions and future work.

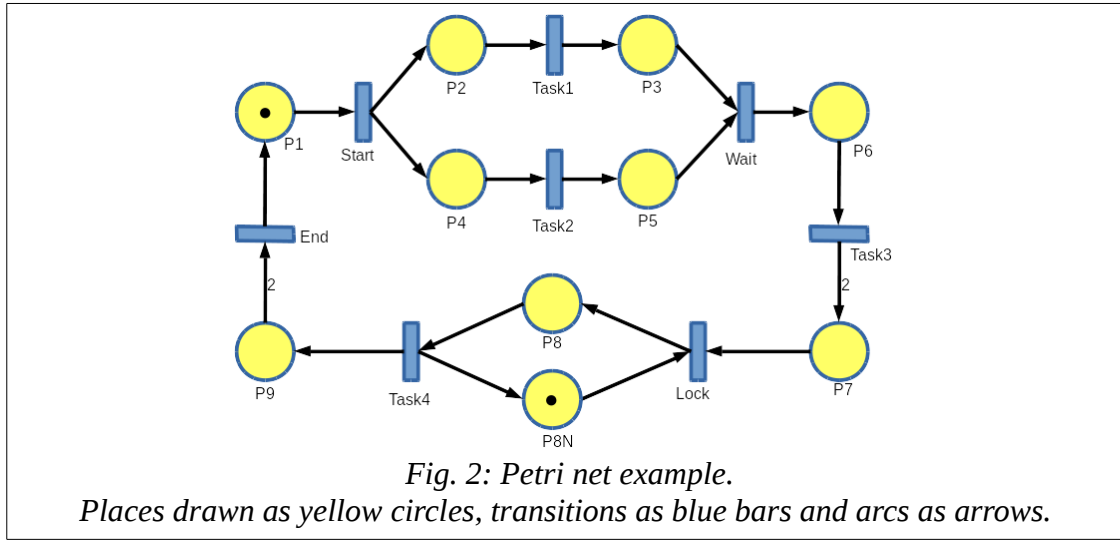
2 Literature Review

This chapter presents a literature survey about related topics. This list is complemented by related work sections on all papers published during the preliminary work and the publications focusing on the DS-Pnet class, that contain references relevant to each specific topic and application examples.

2.1 Petri nets

The Petri net [30][31][72] modeling language, proposed by Carl Adam Petri, frequently used to represent distributed systems, has been widely accepted by the scientific community, referenced by many thousands of research papers from many research areas, including biology, physics, mathematics, business processes, computer science, hardware design and industrial automation. Over the years, many Petri net classes have been created, with extensions for various applications [29][39][40][41][45][46][73].

The original Petri nets consisted of graphs containing places and transitions connected through Arcs, from which figure 2 presents an example. Arcs starting in a place and ending in a transition are called input arcs and output arcs have origin in a transition and terminate in a place. Places may hold marks, often called tokens, and the state of the system, often called the net marking, consists in the current configuration of tokens hold in all places. State changes occur when a transition fires, consuming tokens from the input places and producing new tokens, added to the output places. A transition may only fire when all places connected through input arcs hold tokens. When this condition holds true, the transition is said to be enabled. Conflicts between several enabled transitions may occur if they share the same input places: if one of the transitions fires, it will consume tokens necessary to fire the others.



However, although a transition may be enabled, nothing forces this transition to fire and when multiple transitions are enabled, any of them may (or not) fire. This way, the execution semantics of the original Petri net class is non deterministic. The majority of the net classes derived from the original Petri net class, inherit this execution semantics and are usually used only for simulation. For each Petri net class typically there is a software simulator application, that may be used to perform the so called token-game: enabled transitions are highlighted using a different color and the user may pick the next transition to fire. In alternative, the simulator may choose to fire random enabled transitions and after some time, a graph with the history of recorded markings may be obtained.

2.2 Model checking

Over the years, an entire branch of mathematics has evolved around Petri nets, a sub-set of the general graph theory, and many properties, lemmas and theorems have been studied [31][32][33][46][72][74]. For this purpose, instead of the graphical representation, vector and matrix representations of the Petri net and respective markings are often employed. Figure 3 presents an incidence matrix corresponding to the net on figure 2. Each column correspond to a place and each line to a transition. Negative numbers correspond to tokens removed from inputs places and positive numbers to tokens added to output places.

	P1	P2	P3	P4	P5	P6	P7	P8	P8N	P9
Start	-1	1	0	1	0	0	0	0	0	0
Task1	0	-1	1	0	0	0	0	0	0	0
Task2	0	0	0	-1	1	0	0	0	0	0
Wait	0	0	-1	0	-1	1	0	0	0	0
Task3	0	0	0	0	0	-1	2	0	0	0
Lock	0	0	0	0	0	0	-1	1	-1	0
Task4	0	0	0	0	0	0	0	-1	1	1
End	1	0	0	0	0	0	0	0	0	-2

Fig. 3: Incidence matrix

Using the matrix representation, many properties can be verified, including liveness, boundedness, invariants, reversibility, traps and siphons, among many others. Petri net theory textbooks [30][72][75] cover this subject with extensive detail. Using the matrix representation it is also possible to synthesize controllers to impose restrictions that prevent the reachability of undesirable states [76].

Many properties are better analysed using state-space graphs, also called reachability trees. For example, deadlocks and livelocks are easily found on the state-space graph and many reachability problems are usually checked on the state-space graphs, with the help from languages like linear temporal logic (LTL) or computational tree logic (CTL) [37][46][77]. Unfortunately, real world applications frequently produce very large state-space graphs, with many millions of states. This way, many state-space reduction techniques have been developed and concepts like strongly connected components [38], stubborn sets [78] and other partial order reduction techniques [37][46], among others.

Most Petri net classes have an associated state-space computation and model-checking tool. Popular model checking applications, include the SESA[46][79], Ina/Tina [37], PEP [80], Maria [81], Lola [82], Romeo [83] and PROD [84] tools. The Petri net tool database web page can be consulted for a detailed list [85].

2.3 Execution semantics and non-autonomous properties

As previously stated, the traditional Petri net classes exhibit a non deterministic execution semantics and are used predominantly for simulation and property analysis. However, the goals of the work proposed in this document go beyond simulation and aim the implementation of the real controllers for embedded systems running on the physical hardware platforms and demand determinism. In addition, the typical Petri net classes are autonomous closed systems that do not interact with the external world, but

embedded system controllers do communicate with the controlled systems (the plant), with the users and the surrounding environment.

To solve these issues, several Petri net classes have been proposed. Among those, the signal/nets systems (SNS) [46][86], the net condition/event systems (NCES) [39], signal interpreted Petri nets (SIPN) [87][88], and Input/Output Place/Transition (IOPT) nets [29] are more referenced in the academic literature. Other formalisms that inherit concepts from the Petri nets, as the Grafset PLC programming language [65][89] and the signal transition graphs (STG) [90][91] used for digital hardware design, also try to solve these issues.

The signal/event nets and NCES classes employ mixed execution semantics. In these classes, direct connections between transitions may be established using events, forming synchronous channels, where a master transition generates an event received by slave transitions. Master transitions exhibit a spontaneous behavior, similar to the standard Petri net transition semantics, but the slave transitions have a forced semantics, meaning that an enabled slave transition must fire immediately upon receiving an event. This way, the master transitions display an undeterministic spontaneous behavior but the slave transitions obey a maximal step semantics. Some simulation tools may offer a choice of multiple execution semantics to apply to the spontaneous transitions, including random and interleaving transition firing.

To overcome the autonomous nature of Petri nets, some tools and the underlying Petri net classes rely on code segments that may be added to places and transitions. These code segments correspond to procedures written in foreign high level programming languages, as Java [41] and StandardML [40][42], to support mathematical computations, manipulate operating-system resources and perform communication with the external world, including the creation of graphical user interfaces and network communication. Code segments may be executed whenever a transition is fired, when a place receives a new token, or continuously when a place is marked. Theoretically these code segments can read and write interface signals, enabling the creation of embedded system controllers. Unfortunately, the code segments implementing external communication fall outside the scope of the associated model checking tools, that proceed ignoring the effects of such communication operations or simply cannot be applied at all.

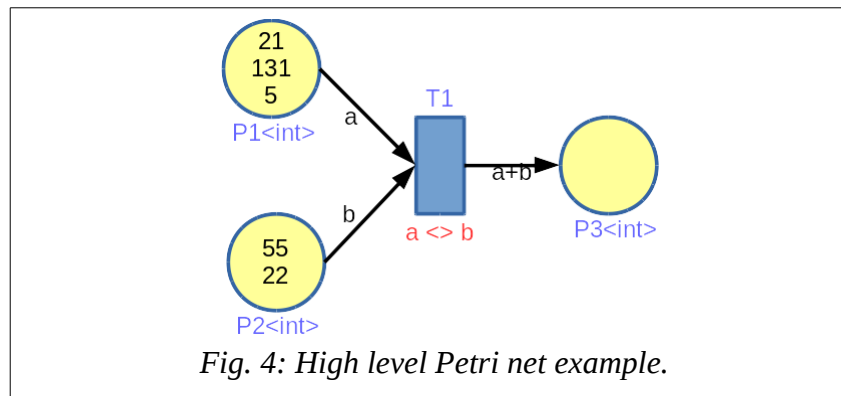
Extensions to the original Petri net class tend to keep the graphical syntax and employ inscriptions on places, transitions and arcs to implement the extended functionality. For instance, timing information [73], transition guard condition and input and output operations [29][39][40][41][45]. This way, the existing tools can ignore the

inscriptions and process only the standard Petri net components. Many properties can be verified just by inspecting the net topology and the respective matrices or state-space graphs, without considering the inscriptions.

2.4 Low level and high level net classes

Petri nets are usually classified as low level or high level nets [40][41][42], from which the Coloured Petri net class should distinguished as the most well known high level class. The main difference between both types resides on the tokens. Low level tokens carry no data, and places only need to record the number of tokens present. On the opposite, high level tokens can hold variables or even complex data structures: a token can be seen as an object with an associated data type. A low level net can be seen as a particular case of a high level net where all tokens are assigned a void data type.

In the same way as tokens, high level places also have associated data types, corresponding to the types of tokens stored inside. Arcs may be assigned names and mathematical expressions to establish relations between token values. In the same way, transition guards may be used to express restrictions on the token values. Figure 4 displays a simple Coloured Petri net.



Observing figure 4, transition T1 picks two tokens from places P1 and P2 and produces a new token for place P3 with the sum of the two values. The input arcs have inscribed names «a» and «b», used to reference tokens inside places P1 and P2 and the output arc has an expression «a+b» representing the sum of two tokens. In order to fire T1, places P1 and P3 must hold tokens satisfying the guard condition «a <> b», that are removed from the input places and a new token with the respective sum is added to P3. For example, a token with value 21 and another with value 55 may be removed from places P1 and P3, producing a new token with value 76 on P3.

Although the execution semantics of high level Petri nets continues to respect all rules of low level nets, it can be seen as a dataflow, where arcs transport data stored in

places and the transitions perform transformations to the data. This characteristic represents an enormous value for embedded system controller design, and some authors have employed high level nets for this purpose [42][56][92].

As an example, an industrial production line can be modeled using high level nets. Individual parts travel on a conveyor belt and multiple machines may apply operations on those parts. The parts can be modeled using tokens, containing information about each part, as the serial number and the types of operations that must be performed. Each machine may be modeled using a transition, that applies transformations to the tokens according to the operations performed on the physical parts. Buffer zones that store parts waiting to be processed may be modeled by places that store tokens.

2.5 High level net execution strategies

When the token data types are restricted to very small sets of colors, it is possible to unfold an high level net into an equivalent low level net [93][94], that although larger, can be processed with the standard low-level Petri net tools, including automatic code generators. However, when the cardinality of the data types employed is larger, this technique results in an explosion on the size of the resulting low level nets that would be impractical. For instance, if the integer numbers used in the figure 4 example, were limited to a simple 8bit (byte) data type, the resulting low level net would contain 65280 transitions. If 16 bit integers were used, then $2^{32}-2^{16}$ transitions would be required.

The traditional high level Petri net execution strategies employed by simulators [95], typically rely on software loops to cyclically evaluate all tokens contained in all input places of a transition, in order to evaluate guard conditions and arc expression relations. If the maximal bound of all places are restricted to a single token, or a limited number of tokens, then the maximum execution time required to evaluate any transition can be predicted. However, for transitions with multiple input places and large bounds, then the combinatorial nature of the required processing would impose large execution times, unsuitable for real-time controller implementations, or very complex hardware implementations consuming large silicon areas. Imposing restrictions on the high level nets might help solve this problem: for instance, defining place capacities to limit the maximum number of tokens (just 1), or defining FIFO or prioritized place semantics where only a single token is ready for use on any single execution step. However, with these restrictions, most of the advantages of high level nets would be lost, continuing to demand more resources. In this sense, dataflow formalisms offer a more intuitive graphical syntax to express mathematical transformations.

2.6 Model composition and hierarchical structuring

As other graphical modeling languages, complex Petri net models tend to lose readability due to the large number of nodes, long arcs with potential crossings and multiple overlapping inscriptions. In this respect, models designed using high level nets tend to become more compact and, consequently, easier to read.

Almost all Petri net dialects and the respective tools support several techniques to increase readability and simplify the design of complex models. The most simple solution consists in the subdivision of models into several pages [55], containing groups of nodes responsible for the behaviour of different sub-systems. Connection between pages is obtained through the insertion of multiples copies of the same nodes on different pages, using the concept of node-references. For example, a place on one page may be re-used on another page using a place-reference with the same identifier. Arcs connected to the node-references are interpreted as arcs connected to the original node. Any tool processing the complete net can start by joining all pages into a flat model and fusing all node-references to the original nodes.

Model composition based on hierarchical structuring techniques not only helps to enhance readability, but also contributes to simplify the design of complex systems, enabling the usage of top-down, bottom-up or even object oriented development strategies. The potential to re-use existing components, previously tested, greatly contributes to reduce development time and cost. Extensible module libraries of “prefabricated” components may be created, grouped into families for many specific scientific fields. Observing the history of computing over the past decades, it is possible to detect a pattern associated with the most successful development languages and formalisms: the availability of good libraries to support the target development fields. Notorious examples are Fortran, Java and C#, PHP, Python, Perl, VisualBasic, Matlab/Simulink [63] and LabView [96].

The node-fusion concept is also on the basis of the hierarchical structuring techniques offered by many Petri net classes [30][40][97][98]. Components are just simpler Petri net models, and the communication interface between different components is established by node-fusion: a place from the component model is fused with other place from the top container model, and the same can be applied to transitions. When multiple components are used, the same node can be shared between more than one component. Some Petri net classes use the concept of macro nodes, where a single node representing an entire sub-model [55][88][98] also rely on node-fusion, but in this particular case the communication interface consists only in a single node.

The nodes used to define the interface are conceptually viewed as input or output nodes. For instance, a component might expect the sudden appearance of new tokens on input places and produce tokens to add to output places. A similar process is used when transitions are used to establish the communication interface: arcs connected to input transitions expect receiving new tokens produced by these transitions, while output transitions remove tokens from internal places. When multiple components are connected, the same node is viewed as output by one module and must be seen as input by the other modules. However, most tools do not strictly enforce this policy, meaning that it may be possible to add external input arcs to the output transitions of a component, and there is no warranty that all tokens placed into output places will be consumed by external transitions. As a consequence, it may not be possible to properly model-check the individual component models, as output places may exhibit infinite bounds and the component execution may be blocked by external arcs added to output transitions. The model-checking tools associated with these Petri net classes will typically start by merging all components into a single flat model and property verification is only applied on the final model.

Some Petri net dialects employ the concept of reference nets [41], a form of model composition, supporting inscriptions to instantiate new nets, that stay associated with tokens in a main model. This way, when a transition containing a form of «new net:name()» inscription is fired, the execution of new model is dynamically started on a parallel thread, and a reference to this net may be associated to a new token. The opposite action may occur when a token referencing a net is consumed by other transition.

2.7 Model composition based in signal and event communication

The problems related by node-fusion based model structuring have been previously discussed by other authors [39], leading the emergence of model composition strategies based on input and output signals and events [39][46][86][87][88][99]. From these, the NCES Petri net class has been the most disseminated.

Compounded NCES models are no longer a Petri net, but are equivalent to set of parallel Petri net models. In addition to the standard Petri net arcs, NCES add the concepts condition arcs and event arcs, used to establish the communication between components.

Condition arcs, equivalent to the test/read arcs defined in other Petri net classes [29], read the number of tokens from an input place but do not remove any tokens. As a consequence, condition arcs do not cause conflicts between competing transitions and

when used to inter-component communication, do not suffer from the same problems as node-fusion. Condition arcs start in a place inside one component and will typically be connected to a transition on a different component. As the target transition does not remove tokens from the source place, the execution of the first component is not affected by the second.

The value of condition arcs is interpreted by transitions as simple Boolean condition and the formalist does not offer any syntax constructs to model the logic operators «not», «and», «or» and «xor». This way, designers have to resort on solutions based on the net topology, drawing multiple transitions to form parallel or series configurations, that greatly reduce readability and increase model size.

The other communication method offered by NCES and SNS are event arcs, used to form synchronous channels between transitions from different components. Event arcs have two fundamental differences from condition arcs. First, condition arcs start in places and finish on transitions, but event arcs connect two transitions, a master to a slave. Second, the behavior of event arcs force the firing of enabled slave transitions, while condition arcs only prevent firing. This effect has deep consequences on the execution semantics of NCES/SNS, as the master transitions - transitions that only have normal and condition input arcs - exhibit a spontaneous behavior, but slave transitions are executed in maximal steps. Transitions connected though event arcs are considered synchronous, meaning that if the master transition fires, all enabled slave transitions will fire in the same execution step.

In addition to the inter-module communication, condition arcs can also be viewed as external input signals, preventing the firing of transitions, with a semantics equivalent to a guard expression. This way, it would be possible to implement automatic code generator tools for embedded system controllers, using condition arcs to model input sensor reading. Unfortunately, the formalism also does not offer syntax constructs to perform mathematical operations, and would require an auxiliary formalism. These problems have been addressed in the signal interpreted Petri net class [87][88].

Model checking and simulation tools are available for the NCES, SNS and SIPN classes [79][100], and the applications to embedded systems and industrial automation, using automatic code generation have been presented [87][101][102].

The use of signal and event communication for inter-component communication provides additional advantages, as it permits the definition of proxy component models to encapsulate external systems designed with other development formalisms. This way, it is theoretically possible to employ virtually any existing system as a component, as long as the external interface can be specified using input and output signals and events,

ranging from existing integrated circuits, IP cores designed using traditional hardware description languages (HDL) [103], systems implemented using programmable logic controllers and almost any system designed/modelled using other languages. In the same way, component models defined using this strategy can also potentially be used by other projects. For example, the code produced by the automatic code generators can be directly inserted into other projects using the same target language [12].

2.8 The IEC61499 standard

In recent years the IEC61499 standard for distributed control and automation [68][69][70] has been assuming growing importance, as it defines the concept of function blocks with an external interface composed of signals and events, but does not impose rigid formalisms for the internal block implementations. This standard has been adopted by several automation controller manufacturers and compatible products are available on the market [104][105].

The standard defines several types of function blocks [70]. Composite function blocks used to implement the entire systems containing multiple function blocks, basic function blocks implementing the component modules, and service interface function blocks (SIFB) to encapsulate systems designed using other development languages or low level sub-systems to access operating systems resources and networking devices.

Basic function blocks are divided into two parts: an execution control chart (ECC) and an algorithmic part. The ECCs are state machines reacting to external input events that trigger the execution of algorithms to process input data. The algorithms may be implemented using many programming languages, as Java, traditional PLC graphic languages as ladder, grafcet or text based PLC languages as instruction list and structured text [64].

Each function block may be implemented on separate execution units, forming distributed topologies, or GALS systems [106][107][108]. A request to a function block typically uses a variant of the following communication pattern: a) input data is made available at the input signals of the function block, b) an input event is generated to trigger the request execution, c) the function block ECC checks pre-conditions and reads input data, d) the necessary algorithms are executed by the function block, e) result data is made available at the output signals, f) an output event is generated to inform that the results are ready. Slow requests requiring many data processing operations or long delays due to communication latencies, or even complex processes involving non trivial state machines, might require more complicated communication protocols. For example, intermediary events to acknowledge request reception and

inform the client that processing has started, might be generated before the results are ready.

As the standard offers some degree of flexibility in the formalisms used internally to specify the function blocks logic and algorithms, several authors have proposed the use of Petri nets, as NCES [109][110][111] and other formalisms [112][113][114] to implement the function blocks state machines. These solutions bring the advantages of formal verification and model-checking to the IEC61499 world.

To take advantage of the growing IEC61499 ecosystem, compatibility with this standard should be a desired goal of any new model composition proposed formalism. Compatibility can be obtained using automatic code generation techniques to translate models to the standard-sanctioned languages, creating real basic function blocks, or creating SIFBs proxy function blocks to encapsulate code produced by the normal code generators.

2.9 Automatic code generation

Traditionally, model based development formalisms have been used for simulation and formal property verification, in order to identify and correct design mistakes before reaching the low-level prototype implementation phase. After the simulation and verification phase have successfully completed, the prototype software code or hardware descriptions were typically written manually, translating model semantics to low level code for the selected architectures. As the final coding was a human task, these formalisms could omit many implementation details, and many system requirements could be specified using an informal syntax based on simple text comments. This approach is frequently found in the literature and the execution semantics of many modeling formalisms does not even ensure determinism.

From another side, the concepts presented in this work, and preliminary development, aim to support all steps of embedded systems controller development, from the early model design and edition to the final controllers deployment on hardware. As a consequence, the syntax and semantic rules of the proposed formalisms must enable the precise specification of all requirements and implementation details, in order to fully support automatic code generation tools.

Ideally, the proposed formalisms should support an incremental refinement of the implementation details, in order to enable the rapid specification of higher-level initial models that later can be improved with the addition of lower-level details. For example, a controller can be fully simulated and model checked without assigning any physical pins to the model inputs and outputs, but these assignments must be set before

generating the final code for the prototype implementation. Complex models, composed of many components can also benefit from a similar approach, by employing a top-down strategy and starting with simplified versions of each component, that later could be replaced by refined versions. Automated model-checking tools [15] can be used to perform regression tests and detect behavioral changes between versions.

However, support for deployment on physical embedded devices, either by using interpretation or automatic code generation strategies, has been offered for a long time: the languages and formalisms used by the industrial programmable logic controllers would be almost useless without it [65]. In the world of digital hardware design, the usage of high level formalisms with graphical syntax has long been used [35][91][115]. In the software world, model based development frameworks have been gaining traction, with many tools supporting UML based languages [34][35][36], as the Ecore Modeling Framework (EMF) from the Eclipse foundation [116][117]. The EMF tool-kit includes a family of meta-models and transformation technologies that support automatic code generation in arbitrary languages [118][119].

Commercial modeling applications, as the Matlab and Simulink tools [63], also support automatic code generation for both software and hardware targets, including personal computers, many digital signal processors and micro-controllers and even for FPGAs [120]. However, the licenses for the automatic code generation tools are expensive, some implementations are not standalone, requiring the presence of a companion computer running parts of the application and graphical user interfaces, and the resulting code may not be easily tailored to fit the requirements. From another side, the modeling languages implemented by these tools cover a very broad range of applications and offer a complex mixture of multiple formalisms, that is not supported by formal model-checking tools.

Several tool frameworks originated from the academic community also support interpretation and automatic code generation. From one side, the simulation tools for formalisms that support code segments, can be used for rapid prototype implementations, employing an interpreted strategy [40][41]. From another side, several automatic code generation tools have been presented in the literature, supporting Petri net based formalisms [8][14][56][87][101][102] and UML statecharts [34][35][36].

2.10 UML statecharts and activity diagrams

In addition to Petri nets, statecharts [121] are the other family of modeling formalisms that have been frequently proposed for embedded system development [115], with several tools based on the UML standards [122]. The tool-chains include

model-checking tools, simulators and automatic code generators [34][35][36]. The UML standard also defines an activity diagram formalism, that in version 1 was based on flowcharts, but version 2 adopted a Petri net semantics [123] and all the considerations about Petri nets can be applied to version 2. The conversion of statechart models to the equivalent Petri nets and the reverse, have been studied by [123][124][125][126][127].

2.11 Model file formats

The importance of the data formats used to store model files should not be neglected, as the availability of libraries and tool frameworks to process those files largely depends on the chosen formats.

XML [128] and JSON [129] based formats are supported by nearly every modern programming language and tool-chains. In the case of XML based formats, there exist libraries to store/parse files from/to DOM trees, syntax checking tools based in XML schemas [130] and RelaxNG grammars [131], query languages based on Xquery [132] and Xpath [133] to quickly find data inside complex documents and XSL transformations [134][135], useful to convert files between different grammars and even to implement automatic code generators [14].

The Petri net Mark-up Language (PNML) file format [136][137] defines a base syntax to represent Petri nets, with two grammar variants for low level (P/T) nets and high level nets. It defines the simple grammar for the basic nodes and arcs, that can be extended through annotations to support the particular features of each Petri net class. The annotations have a common structure, including graphical attributes and a text string, that may be visualized on any Petri net editor, independently of the respective semantic that may be completely ignored by some tools. This way, PNML models designed using any tool could, in principle, be read and visualized by other tools, and be processed by analysis tools that work only on the topological node relations.

Unfortunately, many Petri net tools do not adhere to the PNML standard and even the original members of PNML committee seem to have been moving away from it, preferring the native XSI/XMI [138] XML based formats used internally by the Eclipse Modeling framework [116][117]. However, most tools allow import and export PNML models, and, as both the PNML and XSI/XML formats are based on XML, it is possible to create XSL or MOFscript [118] transformations to perform the conversion.

2.12 Reactive systems and synchronous dataflows

The modeling languages that have been discussed in this section, mostly Petri nets and statecharts, offer very good capabilities to specify the control logic of embedded system controllers, or general discrete event system controllers, that are usually implemented using digital systems. However, real embedded systems almost always employ mixed architectures combining digital and analog subsystems, dealing with analog sensors, analog actuators, motion devices, timing and other variables that are better represented as analog values than using purely Boolean logic. Although previous work on the IOPT class [29] and preliminary contributions already provide some degree of support analog subsystems through the use of integer range signals and arithmetic expressions [5][24], the text based formalisms employed lack the intuitiveness of other graphical alternatives.

From another side, dataflow languages [139], that primarily focus on the movement of data between execution/operation blocks, provide a very intuitive way to express the dependencies between signals and enable the implementation of concurrent computation architectures that take advantage of the parallel capabilities offered by reconfigurable hardware platforms. Instead of specifying imperative sequences of calculations, dataflow languages identify a set of mathematical operations and define the relations between intermediary computed values: every time an input signal changes, all dependent signals must be immediately recalculated.

For example, the spreadsheet applications employ a dataflow semantics: when the contents of a cell are changed, any other cells that reference this cell must be recursively recalculated. Graphical dataflows are directed graphs that use nodes to express mathematical operations and data-processing modules, connected using arcs that define the signal relationships and dependencies to form data paths.

Although dataflow languages have long been abandoned for general purpose computation, the underlying concepts have been recognized as an effective solution for digital signal processing, linear systems control and digital circuit design [140][141]. The recent availability of multi-core personal computers, may one day bring these concepts back from the shadows, due to the ability to deal with concurrency and parallelism.

Synchronous dataflows (SDF), a particular type of dataflows designed for digital signal processing applications, where each node always produces and consumes a fixed number of tokens, were presented in [142], covering many aspects ranging from computation scheduling, parallel implementations, automatic code generation and

correctness verification [143]. The synchronous data flow concepts have been implemented in the languages LUSTRE [144], Signal [145] and Esterel [146]. Applications for embedded system development have been presented in [147], and the association to state machines in [148].

Reactive systems were defined as systems that react to external events, as opposed to transformational systems that continuously apply computations over input signals to produce outputs [149][150]. A synchronous approach has been proposed for both kinds of systems, using two syntax styles: statecharts and dataflow [151], employing an instantaneous execution paradigm where computation times are considered negligible. Applications of the synchronous paradigm to model real-time reactive systems have been presented, including problems related to model composition and detection of signal loops, calculation scheduling in composite models, correctness verification, code generation [152][153][154][155][156], and implementation of IEC61499 function blocks [157].

2.13 Matlab/Simulink

As previously mentioned, the Matlab and Simulink commercial tools [63][120], have been used to model embedded system controllers [158]. These tools support a vast mix of different modeling formalisms, including block diagrams, flowcharts, Petri nets, multiple solver algorithms to support different execution semantics, and even support automatic code generation for software and hardware targets. Requirements based verification tools are available at [159]. Simulink offers blocks for TCP/IP communication that may be used to implement distributed applications that have been used to model cyber-physical systems.

Although Matlab/Simulink can be viewed as a technology that competes with the proposed framework, used to build the same type of solutions, including automatic code generation tools, both solutions can also be used in a complementary way. During the preliminary work a new code generator to produce Matlab code from IOPT models was created, that builds Matlab-system objects (<http://gres.uninova.pt/IOPT-Tools/>) to be used in Simulink. The resulting Simulink blocks offer an external interface composed of the same input and output signals and events as the original IOPT model, applies the same execution semantics and lets the user define the initial marking of each instance. The system objects may be connected to other Simulink blocks and used for simulation and submitted to the Matlab code generation tools.

A similar code generator for the new formalism might be created for the new formalism. This way, DS-Pnet models might be used in Simulink projects and Simulink

might also be used to simulate and validate DS-Pnet systems. This is specially important when a controller-plant strategy is used to model the interactions between the controllers and controlled systems, as the controlled system models may involve dynamic systems that require the employ of the mathematical solvers offered by Matlab. In the same way, there are thousands of available Matlab/Simulink well-debugged models that may be immediately used to model the system plants, without the need to design new models. The code generated automatically may be combined with the existing plan models for simulation and detect design flaws.

2.14 Cyber-physical systems

Cyber-physical systems (CPS) is a multi-disciplinary field that studies systems composed of networks of physical and computational sub-systems, often exhibiting closed feedback loops between both types of sub-systems, covering the areas of software and hardware development, mechatronics, communications, and automation, among others. Although there is no standard definition of CPS, the most common definition found in the literature is «CPSs are defined as the systems that offer integrations of computation, networking, and physical processes» [48][51].

CPS are often considered an evolution of embedded systems. However, the previous generation of embedded systems were envisioned as part of a single equipment or machine and were often contained inside a physical system. In contradiction, CPS are often implemented as distributed heterogeneous systems containing multiple sub-systems that may extend through the Internet, mixing physical systems as sensors, motors and actuators, computing nodes located on the cloud and user interfaces running on mobile computing devices.

Over the last 10 years CPS have been identified as a target for future research, indicating challenges and future roadmaps [50][51][52][53]. A frequent concern is the need for new modeling formalisms capable to offer a new abstraction able to express timing and spacial restrictions and deal with the networking communications and interconnections between physical and computational devices in a transparent way. This work is a step in that direction.

As a result, CPS have been the subject of extensive research work from the academic community, that resulted in numerous publications, covering the areas of modeling, simulation, verification, tool frameworks, hardware platforms, security, reliability, real-time requirements, privacy and multiple fields of application. The development formalisms proposed to implement CPS systems range from the traditional programming languages to modeling using differential equations, synchronous

dataflows, actors and aspects, Ptolomy II [160], Matlab and Simulink, among others. Good surveys about these publications can be found in [47],[48] and [161].

A 5 level guideline for the implementation of CPS solutions has been proposed [71] with the following levels: 1) Smart connection level; 2) Data-to-information level; 3) Cyber-level; 4) Cognition level; and 5) Configuration Level. Although the solutions proposed in this work do not enforce any type of guidelines or specific workflow, the developers are free to employ the mechanisms offered by the new formalism and associated tools to follow any desired guidelines.

In the particular case of the 5C architecture proposed by [71], the new communication protocol and the networking layer included by the automatically generated code provide a good solution to implement the first level, as communication between distributed physical devices and computational nodes is modeled in a transparent way. Level 2 can be implemented using dataflow transformations, used to condition, filter and pre-process data read from local sensors in order to extract relevant information that is forwarded to higher-level computing nodes. In the same way, the cyber level can be implemented using higher-level DS-Pnet models and algorithms implemented using foreign components. The user-interface component library used to implement remote used interfaces and data visualization, in association with the remote debug and monitoring tools help the creation of decisions support systems in level 4. Finally, the same tools can be used to implement supervisory control systems, that will be further augmented in the future with the addition of dynamic reconfiguration capabilities.

The concept of CPS has been identified by the American National Science Foundation as an infrastructure to build the smart systems on the 21th century and dedicated numerous initiatives to the subject [162]. However, the concept has since been adopted by other entities and lead to several research projects [163][164][165][166] and major initiatives as the European Industry 4.0 [167].

3 The DS-PNET Modeling Formalism

3.1 Language core

The DS-Pnet (Dataflows, Signals and Petri nets) modeling language [26] was designed to support the development of cyber-physical systems, employing a mixed approach combining Petri nets [30][31] and dataflows [139][142]. Petri nets are used to model the reactive part of the controllers whose state evolves according to external events, and dataflows are used to specify data processing operations, used to perform mathematical transformations on input signals, and calculate output values. Model composition based on components, communicating with each other using input and output signals and events, enable the creation of reusable component libraries and the implementation of distributed cyber-physical systems containing networks of remote components communicating through the Internet.

A DS-Pnet model can be divided in two parts. The Petri net part of a DS-Pnet model is a non-autonomous low level Petri net, inheriting the main characteristics of the parent IOPT Petri net class [29], including a maximal step execution semantics, transition priorities, transition guards and transition input events. The dataflow part inherits principles from synchronous dataflows, where the execution time of dataflow operations is considered instantaneous, with no propagation delays. The external interface of DS-Pnet models (and components) also inherits the characteristics of IOPT-nets, including the input and output signals and events and the Boolean and integer range data-types.

Dataflow and Petri net nodes interact with each other in a bidirectional way. From one side, dataflow operations may be used as guard expressions or as transition input events to prevent transition firing. From another side, dataflow operations may use

place marking and events triggered by transition firing to calculate output values. To ensure deterministic execution, a set of semantic rules has been defined, specifying the relationships between the Petri net nodes and dataflow nodes, presented at the end of this chapter.

Tables 1 and 2 present the Petri net and dataflow elements available in DS-Pnets.








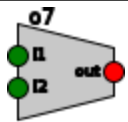
Place		<p>Low-level Petri net place that may hold a positive integer number of tokens</p> <p>Characteristics:</p> <ul style="list-style-type: none"> - Unique identifier - Initial Marking: Zero or positive integer number - Name: Text String - Comment: Text String - Graphical position {X,Y}
Transition		<p>Low level Petri net Transition</p> <p>Characteristics:</p> <ul style="list-style-type: none"> - Unique Identifier - Priority: Integer number - Name: Text String - Comment: Text String - Graphical position {X,Y}
Petri net Arc		<p>Traditional Petri net arc used exclusively to connect places and transitions</p> <p>Characteristics:</p> <ul style="list-style-type: none"> - Unique Identifier - Inscription/weight: natural number - Source and target node identifiers - Visualization mode: Graphic / Symbolic

Table 1: Petri net elements

Signal	 	<p>Signal – Variable used to convey information whose value varies with time</p> <p>Characteristics:</p> <ul style="list-style-type: none"> - Signal name: valid unique identifier - Data-type: Boolean or Integer range (min, max) - I/O Mode: Input (green), output (red) or internal (gray) - Comment: Text String - Graphical position {X,Y}
Event	 	<p>Event – Represents an instantaneous happening that occurs during a single execution step</p> <p>Characteristics:</p> <ul style="list-style-type: none"> - Event name: valid unique identifier - I/O Mode: Input (green), output (red) or internal (gray) - Comment: Text String - Graphical position {X,Y}
Operation		<p>Dataflow node, called operation, calculates one or more output values using mathematical expressions combining inputs and literal values. Contains a set of anchors to connect read arcs. Input anchors are drawn as green and outputs as red. Model designers may choose trapezoid/arrow, circle or rectangle shapes. A collapsed visualization mode, where only the math-expressions are visible, helps reduce clutter.</p> <p>Characteristics:</p> <ul style="list-style-type: none"> - Unique identifier


		<ul style="list-style-type: none"> - Name: text string - Comment: text string - Shape: Trapezoid, Rectangle or Circle - Size: Natural number - Inputs: Name, data-type, dynamic-type, dynamic-name - Outputs: Name, math-expression, data-type, dynamic-type, dynamic-mode - Visibility: Graphical / collapsed - Lock: Locked / Editable - Graphical position {X,Y}
Read Arc		<p>Read arc - used to transmit data between nodes: Transmit data when connecting dataflow nodes, input and output signals Transmit events when connecting events and transitions Also used to read place marking without removing tokens from the input place (test-arc)</p> <hr/> <p>Characteristics:</p> <ul style="list-style-type: none"> - Unique Identifier - Visualization mode: Graphic / Symbolic - Source and target nodes

Table 2: Dataflow nodes

The Petri net elements of a DS-Pnet model behave as traditional low level P/T net nodes [30]. The state of the Petri net part of a model is defined by the number of tokens in each place, called the net marking, whose evolution depends on the transition firing sequence.

Transition firing is controlled by the set of arcs ending at the respective transition, that may be Petri net arcs or read arcs. A transition is enabled when all input places hold enough tokens to satisfy the weights inscribed in the respective arcs. However, transition firing may be subsequently inhibited by read arcs, in form of guards, input events, test arcs and synchronous channels. Table 3 presents a list of possible constructs built using read arcs and Petri net arcs. It is important to notice that Petri net arcs are drawn with an arrow near the target node, read arcs transmitting events finish with a diamond and read arcs transmitting Boolean guard-condition values finish with a circle. This graphical format was chosen for compatibility with the existing NCES Petri net class that used a similar graphical notation [39].

Execution of a DS-Pnet model is performed in discrete steps, called execution-steps. In each step all dataflow nodes and transitions are evaluated. Due to the bi-directional interactions between transitions and dataflow operations and master-slave dependencies imposed by synchronous channels, a deterministic evaluation sequence was defined, attributing a set of micro-step and nano-step numbers to each transition and dataflow node. The simulation and automatic code generation tools employ these numbers to schedule the calculations of dataflow nodes and to evaluate the transition firing, ensuring that all values required to calculate a certain result were previously obtained.

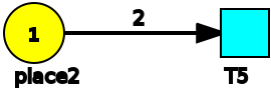

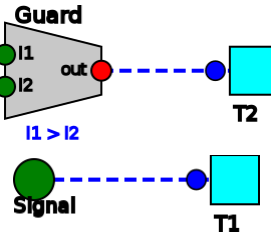
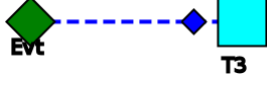
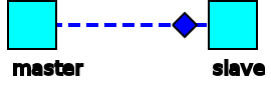
Petri net arc		Traditional Petri net arc – firing is only possible when the number of tokens on the input place is equal or exceeds the arc weight inscription. Input tokens are removed from the input place upon firing. When all input places hold enough tokens, the transition is said to be “enabled”.
Test Arc		A read arc tests if a place is marked and holds at least one token. Firing is inhibited when the place has no tokens. Upon firing no tokens are removed from the input place. The read arc transmits the value of the place marking on the previous execution step. As read arcs have no weight inscription, multiple-token marking conditions may be specified using dataflow comparative operations.
Guard condition		Read arcs starting on dataflow operations or starting directly on signals define transition guards. A transition may only fire when all guard conditions hold true.
Input Event		Read arcs starting on a event or a dataflow operation that produces an event result, define transition input events. A transition may only fire when the event was triggered during the present execution step.
Synchronous channel		A read arc connecting two transitions defines a synchronous channel. When the master transition fires, it emits an event received by the slave. The slave transition can only fire if the master has fired on the same execution step. However, the master transition may fire independently of the slave.

Table 3: Transition firing inhibition constructs

Cyclic dependencies between dataflow nodes or transitions, considered modeling errors, are detected during micro-step/nano-step assignment, discussed at the end of this chapter. For example, the inputs of a dataflow operation may not depend on other values that were calculated based on its own results. In the same way, the master-slave relationships between transitions connected through synchronous-channels cannot form loops. To avoid cyclic dependencies, a delay operator, inherited from synchronous dataflows [142], may be employed in mathematical expressions to refer values calculated on previous execution steps. As the values from previous execution steps are memorized, they can be used anywhere in the model without creating calculation loops and thus preserving the synchronous paradigm.

Transition firing evaluation follows a maximal step semantics, meaning that all transitions that are enabled and ready to fire must fire on the next execution step. Conflicts between transitions may occur when multiple enabled transitions compete for the same tokens from shared input places, but the number of existing tokens is not

enough to fire all of them. In this situation, transitions are sorted using a triple criteria (micro-step number, priority and identifier) and are sequentially evaluated by this order. This way, synchronous-channel masters are evaluated before slaves, the transitions with lower priority numbers will be evaluated first, and finally, the transitions with the same micro-step and priority values will be sorted by identifier. In case a conflict occurs, the transitions evaluated first will grab the disputed tokens. To assist conflict resolution, the editor tool calculates the evaluation sequence, presented as numbers inside each transition.

3.2 Dataflow operations

Data transformation is performed by dataflow nodes, called operations, that apply mathematical expressions to input data and calculate output values (output expressions). By default, each operation has only one output, but it is possible to add multiple outputs. In this case, each output is associated to a data type and mathematical expression. An operation with multiple outputs is equivalent to multiple single-output operations sharing the same inputs, thus reducing the number of arcs on screen.

Output expressions may be composed of a single mathematical expression or multiple conditional expressions using WHEN/OTHERWISE constructs. When more than one expression is specified, all expressions, except the last, must contain a WHEN condition. Evaluation is performed starting from the first expression and stops when the first valid WHEN condition is found. In the following two examples, o10.out uses a single-expression and o11.out employs multiple conditional expressions:

```
o10.out =      (i1 + i2 + i3 ) / 3
o11.out =      i1 WHEN (i1 > i2 AND i1 > i3)
                i2 WHEN (i2 > i1 AND i2 > i3)
                i3 OTHERWISE
```

Mathematical expressions are composed of operands and operators, as listed in table 4. Operands may consist of literal values, expressed as decimal or hexadecimal numbers (0x prefix), or operation input-anchor names. Mathematical expressions may only refer to local names of the operation input anchors. This way, all the relationships between nodes and data dependencies are explicitly visible through read-arcs. This restriction also brings advantages in the reuse of existing operations, permitting the creation of libraries containing frequently used operations. As all dependencies between dataflow operations are expressed using arcs, it is possible to duplicate or copy&paste parts of other models without the risk of losing hidden dependencies to external signals.

Arithmetic operators		
Addition	+	
Subtraction	-	
Multiplication	*	
Division	/	(VHDL requires a power of 2 as 2 nd operand) A 1/N data table may be employed for different denominators.
Modulus / Remainder	MOD	(VHDL requires a power of 2 as 2 nd operand)
Unary - / Symetric	-	
Comparative operators		
Less than	<	
More than	>	
Less or equal	<=	
More or equal	>=	
Equal	=	
Different	<>	
Logic operators		
Logical and	AND	
Logical or	OR	
Logical xor	XOR	
Logical NOT	NOT	
Bitwise operators		
Bitwise AND	ANDB	
Bitwise OR	ORB	
Bitwise XOR	XORB	
Bitwise NOT	NOTB	
Other operators		
Sub-expression	()	
Delay operator	input[-n]	Fetch the past value of an operation input from a previous execution step (n steps go)
Table/Array index	table[in]	
Conditional operator	WHEN	
Default condition	OTHERWISE	

Table 4: Expression operators

In addition to the standard arithmetic, comparative and logic operators, DS-Pnet defines a delay operator, used to access data from previous execution steps. This operator may be used for different purposes, including the detection of events caused by changes in input signals and the implementation of signal filters in the time domain, but it also may be used to avoid cyclic dataflow dependencies. Signals associated with the delay operator imply the existence of memory elements to store the values of this signal on the previous execution steps, implemented in a shift-register fashion.

However, a side effect of this operator may also be used to optimize hardware implementations: As the hardware code generator implements dataflow operations using combinatory logic, models with long chains of dependent operations would produce long chains of combinatory logic, that would impose restrictions in the maximum clock-frequency. The delay operator breaks these chains, splitting the calculations through several consecutive clock cycles, conducting to pipelined implementations: dataflow operations before a delay operator are calculated in clock cycle N , but the operations after this operator are calculated in the next clock cycle ($N+1$).

Each dataflow operation may hold a bi-dimensional array¹ used to store constant data tables, for instance containing the values of general purpose functions of one or two integer arguments. Individual table elements are accessed in mathematical expressions using the “[]” operator. The editor tool offers functions to fill the table contents from mathematical expressions and has the ability to import and export data in spreadsheet compatible formats (CSV). Tables of precalculated data simplify the implementation of general purpose functions using re-configurable hardware and low-end micro-controller devices.

Operations without any input anchors may be used to specify constant values. To simplify constant definition, the editor has a tool that automatically creates a dataflow operation from a numeric value.

All values used in mathematical expressions must have well defined data-types. The available signal data-types, inherited from the IOPT Petri net class, are Boolean and integer ranges. Fixed point data-types with 8, 16 or 24 fractional bits are planned for a future implementation. These data-types were selected to support code generation for very low end hardware architectures, as 8 bit micro-controllers and re-configurable hardware. However, in the future, the automatic code generation tools may implement fixed point arithmetic operations using floating-point hardware on targets that support it.

In addition to the Boolean and integer range data-types, an event data-type was defined, representing instantaneous happenings that typically hold true for only a single execution step. In this way, the DS-Pnet event node may be seen as a signal associated to an event data type. Operation input and output anchors may be assigned the event data-type, that is treated inside mathematical expressions as Boolean values.

In the same way as the input and output signals, each dataflow operation input and output anchor is assigned to a data-type. This assignment may be performed explicitly by the designer or automatically by the system. As the manual assignment of data-types

¹ For uni-dimensional tables, the 2nd dimension may consist of a single column.

and names to individual anchors may become a time consuming task, an algorithm to dynamically define these values was created. By default, all input and output anchors have dynamic names and data-types, that change dynamically whenever an arc is attached to an input anchor. However, the designer may choose to define fixed names or data-types that remain unchanged when connecting arcs. In this case, the tools must verify data-type compatibility between the source and destination nodes of read-arcs. Users must also check for data-type mistakes, as any integer range value connected to Boolean signals or input anchors are automatically converted to true or false.

The dynamic data-type of an input anchor is copied from the source node whenever an arc is attached. In the same way, the source node names are used to dynamically define input anchor names. A set of heuristic rules are employed to define the data-type of output anchors:

- Boolean: when the output expression contains comparative and logic operators (except inside WHEN conditions)
- Boolean: when all input anchors are also Boolean values
- Integer range otherwise. The range limits are obtained the minimum and maximum of all input anchor ranges ².

Usually these heuristic rules produce the expected results in most cases, but the user can manually change the data-types and disable dynamic type assignment. In the same way, the anchor names assigned during arc attachment may not produce the desired naming. For instance, some operations have typical names, as «reset» or «enable» that may not match the arc's source node names.

When the data-type of a signal or an anchor suffer changes, these changes will propagate through read-arcs and mathematical expressions through the dataflow nodes that depend on this value, as long as the respective anchors have dynamic data-types.

3.3 Components

The external interface of a model is composed of a set of input events, input signals, output events and output signals defined in the model. Signals and events defined as «internal» are provided as a convenience to give explicit names to internal values, and help reduce the number of long arcs crossing the models. A model may be used to implement an entire application, or may be used as a component to build more complex systems, or both things simultaneously. This way, whenever the user saves a

² A better heuristic would calculate the minimum and maximum values of the output expression for all combinations of input values, however the computation time would impose interactivity drawbacks during model edition.

model, a component symbol with the corresponding interface is automatically created.

Almost all programming languages and modeling formalisms offer some form of structuring mechanisms to enable the division of large models into smaller parts or sub-systems, or the composition of new application models based on existing modules. DS-Pnets support model composition based on components that communicate with each other (and to the external world) using an external interface defined by input and output signals and events.



Fig. 5: Component

A component, as presented in figure 5, may be viewed as an object in object-oriented languages, containing its own internal data and algorithms. It may be implemented using another DS-Pnet model or designed using any other modeling formalism, as long as the external interface may be specified as a set of input and output signals and events. For example, a component could be implemented using an existing IOPTnet model [29], an IEC61399 function block [69][70], a digital integrated circuit or an IP core, even if the internal details of these components are now known.

Components may also be used to interface with existing software code, used to invoke algorithms and encapsulate objects defined in object-oriented languages. In this case, the component may contain a set of input events, used to invoke the object methods and a set of input signals used to pass parameters to these methods. Output signals may be used to pass results and expose status data and output events to acknowledge data reception, mark the completion of algorithms or error exceptions. The DS-Pnet “C” code generator tool supports external components, defining data-structures and function-calls to implement the glue-logic used to communicate with the external code.

A DS-Pnet application may be constructed using components implemented with different technologies or distributed through different devices, as long as it is possible to establish a communication layer between the components and the top model. For example, a co-design solution might employ a main model implemented as software and some components as hardware³. In this case, the glue-logic just transfers input and output data through the interface from the CPU running the main application to the hardware where the components were synthesized, that depends on both the operating system and the tool-chain offered by the hardware vendor. In the same way, a DS-Pnet

3 To employ with the existing FPGA PCI cards for personal computers, or system-on-chips containing both CPU cores and re-configurable hardware, as the Xilinx Zinq platform.

model may be implemented using a network of distributed components, located at remote Internet locations. To implement these solutions, a communication protocol based on JSON/HTTP has been specified [22][129], and client/server software added to the automatic C code generators, permitting the design of distributed cyber-physical systems in a transparent way, as described in chapter 5.

Components are represented graphically as a dark-gray rectangle with a set of input and output anchors. Contrary to dataflow operations, the names and data-types of the anchors are fixed and cannot be changed, unless the underlying implementation model is changed. Component symbols are generated automatically from the list of signals and events of a DS-Pnet model (or IOPTnet model). The names and data-types of each input/output are fetched from the original DS-Pnet model, and are sorted according to the Y coordinate, with events always on top, inputs on the left side and outputs on the right.

In addition to the list of input and output anchors, each component is also characterized by an identifier, a name, a comment string, a class name and information about the implementation and target hardware. The component implementation refers to the formalism used to develop and specify the component behavior, that may be a DS-Pnet model, an IOPT-net model or a Foreign component. Foreign components are designed with other tools and development languages, whose code will be linked/added to the output of the automatic code generator tools.

The interface of a foreign component may be specified creating an empty DS-Pnet model, just adding a list of input and output signals and events, and setting the implementation property as “foreign”. The target property of a component refers to the code generators employed; default, software, hardware or remote. This way, it is possible to support co-design solutions where some components are implemented as software and other as hardware and, in the future, the glue logic for specific hardware platforms may be added to automatic code generators.

Two additional string parameters, «resource-location» and «param-string» are used to support the implementation of foreign components and distributed Cyber-physical systems. The resource location is used to pass information to the “C” code generation tools about the location of remote components, including an internet address or a logical node address and the component identifier on the remote model. When applied to foreign/external component implementation, the resource location property may be used to specify the location of data-files, including user interface icons and sound wave files, data-base tables, communication ports, etc. The «param-string»

property may be used to pass application related parameter data, as communication port parameters, user interface text strings and keyboard accelerator shortcuts, etc.

Distributed Cyber-physical systems usually employ networks of remote components. These components may be designed and developed by third parties and be used simultaneously by several applications. For example, a traffic sensor located on a road may be used by multiple traffic control applications running on in-vehicle systems or mobile computing devices. To achieve this, the external interface of these components must be accessible from the Internet, even if the implementation model/code is hidden.

3.4 Example DS-Pnet model

Figure 6 presents an example model illustrating common DS-Pnet constructs, including the relationship between dataflow and Petri net nodes. This model mixes both types of nodes side-by-side. However, if desired, the dataflow nodes could be drawn apart from the Petri net nodes and symbolic-view arcs used to connect both parts, thus separating the data-processing part from the reactive controller part of the model.

This model employs 3 input signals, where «Btn» is a Boolean signal and «Input1» and «Input2» have an integer range data type. Another input, «Start» is an event that represents an instantaneous happening. In a similar way, the model has two output signals, a Boolean «LedOut» and an integer range «Counter» and also produces an output event «OutEvent».

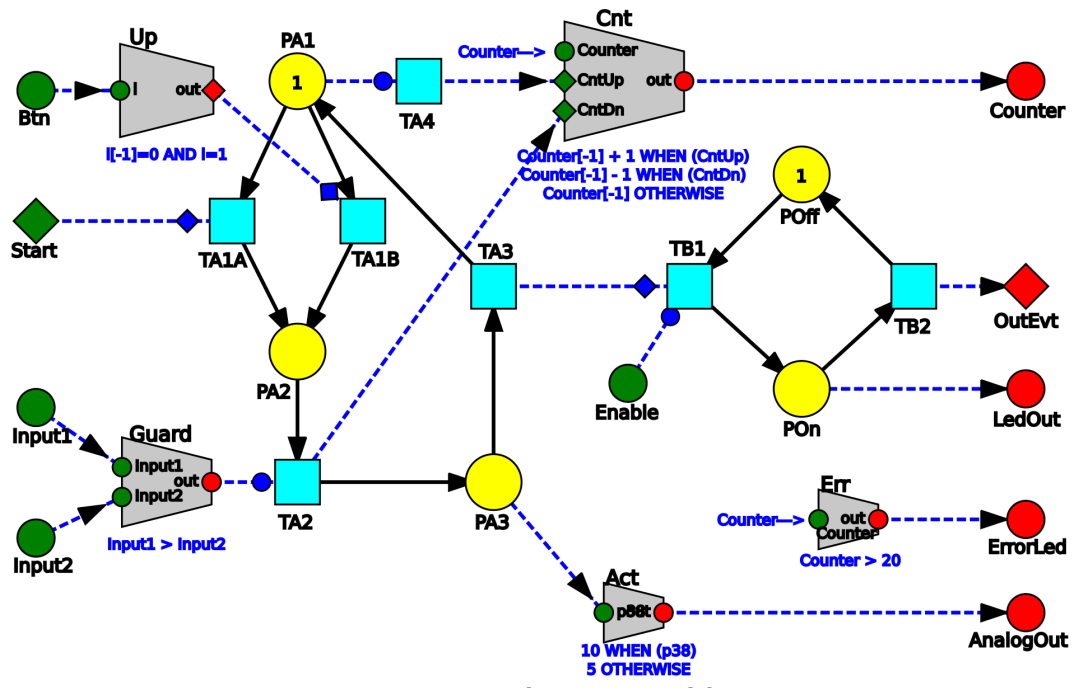


Fig. 6: Example DS-Pnet model

The execution of this model is centered on the behavior of the transitions, whose firing is conditioned by both the place marking and dataflow nodes, with guards and input events coming directly from input signals and from dataflow operations.

When model execution starts, the places «PA1» and «POff» are marked and, as a result, transitions «TA1A», «TA1B» and «TB1» are enabled. In addition, the transition «TA4» is also enabled. In fact, this transition is always considered enabled as it does not have any input Petri net arc. It is connected to place «PA1» using a test arc that reads the place marking but does not consume any tokens.

Transitions «TA1A» and «TA1B» are both inhibited using input events, as «TA1A» is connected directly to the «Start» external input event and the firing of «TA1B» is controlled by an event generated internally by the «Up» dataflow operation. The event is calculated using the output expression « $I[-1]=0$ and $I=1$ » that employs the delay operator « $[-1]$ » to fetch the value of the input signal on the previous execution step, detecting positive-edge signal changes from zero to one, corresponding to the instant when the user presses a button.

The previous expression produces an event based to a simple edge-up detection. However, using different expressions it is possible to define more complex event semantics. For example, one problem that often arises with mechanical switch buttons is called «bounce»: instead of producing a single edge-up transition, a button might produce a very fast train of pulses while the metallic contacts are approaching each other but are not fully connected yet. In order to overcome this problem, an expression like « $I[-4] = 0$ and $I[-3] = 1$ and $I[-2]=1$ and $I[-1]=1$ and $I=1$ » will employ the past four input values to filter bounce pulses and ensure that the signal is stable during the past four execution steps⁴. Similar expressions may be used for other purposes, to filter high frequency noise from digital input signals.

As previously referred, when execution starts, both transitions TA1A and TA1B are simultaneously enabled. However, place «PA1» has only one token, that is not enough to fire both transitions. As a result, if the events inhibiting the firing of these transitions happen during the same execution step, a conflict between the transitions will arise. Conflicts are commonly solved by assigning different priorities to each transition: in this case, only the transition with lower priority number will fire. When both transitions have the same priority number, firing is sorted according to the respective unique identifiers, meaning that the oldest (firstly drawn) has priority.

4 As the execution step frequency is often much larger than the of bounce pulse frequency (MHz vs Hz), a more robust bounce filter solution might employ a counter component.

After one of these transitions fire, place «PA2» will be marked and transition «TA2» will be enabled. However, «TA2» is inhibited by a guard operation that compares «Input1» and «Input2». When the value of «Input1» exceeds «Input2», TA2 will immediately fire in the next execution step and «PA3» will be marked.

Place «PA3» is connected to a dataflow operation «Act» that calculates the value of an output signal «AnalogOut». The output expression uses a WHEN/OTHERWISE construct to calculate the output value: 10 when PA3 is marked and 5 otherwise⁵. However, place «PA3» is only marked during a single execution step, as the transition «TA3» is not inhibited by any guard or input event and will fire as soon as it is enabled. This way, the output of «AnalogOut» will consist on a steady value of 5 with sporadic spikes with value 10. After «TA3» fires, place «PA1» will be marked and the left part of the model (PA1, PA2, PA3) will return to the original state.

Transition «TA3» is connected to transition «TB1» using a read-arc, forming a synchronous-channel where «TA3» is the master and «TB1» is the slave. This means that transition «TB1» may only fire if «TA3» also fires in the same execution step. However, «TB1» may not be able to fire when «TA3» fires, as «TB1» may not be enabled: «Poff» unmarked or the guard signal «Enable» may not hold true.

When «TB1» fires, place «POn» will be marked during exactly one execution step, as transition «TB2» will immediately fire. This way, the output «LedOut» connected to place «PB2» will blink during one execution step, while the place is marked and the event «OutEvt» will be triggered on the next execution step, caused by the firing of «TB2». As this point, the right side of the model (POff, POn) will also return to the original state with «POff» marked.

The complete state of this system is composed of the place marking and also by the «Counter» output. The value of this output must be memorized by the system as it is used in association with the delay operator «[-1]» in the «Cnt» operation. The respective value is incremented when «TA4» fires (while «PA1» is marked), is decremented when «TA2» fires and remains constant otherwise. The real values of the «Counter» output are limited by the integer range limits defined in the data type.

5 The word OTHERWISE is implicit on the last expression of a WHEN construct and is optional.

3.5 Model files

DS-Pnet models are stored using a XML file format. XML was chosen due to the wide support across almost all programming languages, the availability of many parsing libraries and processing and validation tools, including dictionary based syntax validation (DtD, RelaxNG), query languages (Xpath, Xquery) and transformation engines (XSLT).

Although a standard XML file format to represent Petri net models (PNML) exists, it was not used. First, it would require non-standard extensions to represent input and output signals and the dataflow part of DS-Pnet models. Second, the PNML syntax is too verbose, where node properties are usually stored inside children nodes, requiring increased parsing effort to process model files, with a negative impact on tool development. Finally, the chosen XML data format can be easily converted to PNML using simple XSL transformations. In fact, the IOPTflow framework includes a transformation that extracts the Petri net part of a DS-Pnet model and converts it to an IOPTnet model, stored as a PNML file. This model may be subsequently processed with the IOPT model-checking tools.

The following XML document contains an excerpt of the model file presented in figure 6, truncated to include just one example of each type of DS-Pnet nodes:

```
<?xml version="1.0"?>
<net name="tst3" type="iopt-flow">
  <place id="p001" x="210" y="125" init_marking="1">
    <name off_x="-10" off_y="-10" text="PA1"/>
    <comment off_x="0" off_y="20" text="-"/>
  </place>
  <transition id="t006" x="150" y="200" priority="0">
    <name off_x="-10" off_y="-10" text="TA1A"/>
    <comment off_x="0" off_y="20" text="-"/>
  </transition>
  <event id="Start" x="50" y="200" mode="input"/>
  <signal id="Input1" x="50" y="315" mode="input" type="boolean" min="0" max="1"/>
  <signal id="LedOut" x="725" y="285" mode="output" type="range" min="0" max="255" dynamic="type"
frac="0"/>
  <arc id="a010" type="normal" source="p001" target="t006"/>
  <arc id="a021" type="read" source="Start" target="t006"/>
  <operation id="o028" x="120" y="140" rot="0" shape="arrow" size="16">
    <name off_x="-11" off_y="-16" text="Up"/>
    <input off_x="-16" off_y="0" name="i" id="o028.i" type="range" min="-32768" max="32767"
frac="0"/>
    <output off_x="16" off_y="0" name="out" id="o028.out" type="event" min="0" max="1"
dynamic="none" frac="0">
      <expression>
        <text>i[-1] = 0 AND i = 1</text>
        <operand type="signal" idRef="i" delay="1"/>
        <operator type="equal"/>
        <operand type="literal" value="0"/>
        <operator type="and"/>
        <operand type="signal" idRef="i"/>
        <operator type="equal"/>
        <operand type="literal" value="1"/>
      </expression>
    </output>
  </operation>
</net>
```

Listing 1: Part of the DSP-net XML document from the model presented on fig. 2.

Each node contains the XML representation of the respective DS-Pnet node characteristics presented in tables 1 and 2. In order to simplify parsing, most of these characteristics are encoded as XML node properties, except for items that may have multiple instances in the same node, encoded as child nodes. The name and comment characteristics, not employed by the execution semantic tools⁶, are also stored into children nodes.

Mathematical expressions are encoded as an hierarchical XML tree, containing a sequence of operands and operators. A list of available operators has been presented in table 4. Operands may consist of literal values, operation input names and sub-expressions. This format was chosen to simplify the implementation of the automatic code generation tools, simplifying the translation to the syntax of the target languages, that may have different rules.

```
<component id="c1" class="protocols/modbus_if.xml" x="630" y="585" width="170" height="180" rot="0"
implementation="iopt-flow" target="external" res_location="/dev/ttyUSB0" param_string="19200,8,N">
  <name off_x="-85" off_y="-95" text="modbus"/>
  <source_model file="files/modbus_if.xml"/>
  <input id="c1.ReadInput" name="ReadInput" type="event" off_x="-85" off_y="-"/>
  <input id="c1.WriteCoil" name="WriteCoil" type="event" off_x="-85" off_y="-30" />
  <input id="c1.WriteReg" name="WriteReg" type="event" off_x="-85" off_y="-10" />
  <input id="c1.m_id" name="m_id" off_x="-85" off_y="10" type="range" min="0" max="255" />
  <input id="c1.s_id" name="s_id" off_x="-85" off_y="30" type="range" min="0" max="255" />
  <input id="c1.Addr" name="Addr" off_x="-85" off_y="50" type="range" min="0" max="65535" />
  <input id="c1.WrValue" name="WrValue" off_x="-85" off_y="70" type="range" min="0" max="65535" />
  <output id="c1.RecvAns" name="RecvAns" type="event" off_x="85" off_y="-70" />
  <output id="c1.Error" name="Error" type="event" off_x="85" off_y="-50" />
  <output id="c1.Ready" name="Ready" off_x="85" off_y="-30" type="boolean" />
  <output id="c1.RdID" name="RdID" off_x="85" off_y="-10" type="range" min="0" max="255"/>
  <output id="c1.RdValue" name="RdValue" off_x="85" off_y="10" type="range" min="0" max="65535" />
</component>
```

Listing 2: XML representation of a component instance.

Component instances have a XML encoding similar to dataflow operations, with a list of input and output signals and events. Listing 2 presents an example instantiation of a foreign component implementing an interface to communicate with industrial devices using the ModBUS field-bus protocol. Each component instance contains a reference to the source model implementing the component, that are be used by code generation tools to build flat models of entire systems. Two additional attributes, resource location and parameter string, are used by the automatic code generation tools to pass information to foreign components, coded using external programming languages. In this example, are used to select a serial port adapter and define the serial communication parameters.

6 Although the comment attribute may be used by foreign components.

3.6 Execution Semantics

The DS-Pnet modeling formalism combines concepts from Petri nets and dataflows. Although both fields have been extensively studied in the past [30][31][32][33][75][139][142][144][145][146], the interaction between dataflow operations, input and output signals and Petri nets must be studied. In order to ensure deterministic execution, the semantic rules resulting from such interactions must be well specified. An early version of the DS-Pnet formal definition has been presented in [26]. Later, during the implementation of the automatic code generation tools, additional requirements were added, needed to obtain coherent execution behavior between software and hardware implementations, described in [28].

3.6.1 Formal definition

A DS-Pnet model is a directed graph, combining Petri net nodes, dataflow operations and input and output signals and events. The Petri net nodes are used to model the system state and reactive behavior, dataflows nodes are used to perform data processing and signals and events define the external interface.

Definition 1: A DS-Pnet model is described as a tuple $DS-Pnet = (P, T, S, E, O, A, R, m_0, s_0, w, pt, ex, st, ot)$ satisfying the following requirements:

- 1) P is a finite set of places
- 2) T is a finite set of transitions
- 3) S is a finite set of signals
- 4) E is a finite set of events
- 5) O is a finite set of dataflow nodes, called operations
- 6) $P \cup T \cup S \cup E \cup O = \emptyset$
- 7) A is a finite set of Petri net arcs with $A \subseteq (P \times T) \cup (T \times P)$
- 8) R is finite set of read arcs with
$$R \subseteq (S \times S) \cup (S \times O) \cup (S \times T) \cup (O \times S) \cup (O \times O) \cup (O \times T) \cup (P \times T) \cup (O \times E) \cup (E \times O) \cup (E \times E) \cup (E \times T) \cup (T \times T)$$
- 9) $\forall s \in S, \# \{(x \times s) | (x \times s) \in R\} \leq 1$ (signals have no more than one input arc)
- 10) $\forall e \in E, \# \{(x \times e) | (x \times e) \in R\} \leq 1$ (events have no more than one input arc)
- 11) m_0 is the initial place-marking function with mapping $m_0: P \rightarrow N_0$
- 12) s_0 is the initial signal values partial function with mapping $s_0: S \rightarrow N_0$
- 13) w is the Petri net arcs weight function with mapping $w: A \rightarrow N$
- 14) pt is the transition priority function with mapping $pt: T \rightarrow N_0$
- 15) ex is a function applying operations to mathematical expressions
(where non-literal operands(nlop) are the source of input arcs)
 $ex: O \rightarrow exp$, where $\forall nlop \in exp(O), nlop \in \{x | (x, O) \in R\}$
- 16) st is a signal type function with mapping $st: S \rightarrow t, t \in \{Boolean, Range\}$
- 17) ot is an operation result type function with mapping $ot: O \rightarrow t, t \in \{Boolean, Range, Event\}$

For improved readability, this definition has been simplified in two ways:

a) Components have not been considered. However, a model containing components may be transformed into a flat model, without components, where the component anchors are converted into signals and events, adhering to the previous definition. Components implemented as DS-Pnet models are absorbed into the main model and the respective anchors are converted into internal signals and events. Inputs and outputs of foreign components, that may consist of physical devices or systems implemented using external tools, are respectively transformed into output and input signals that are appended to the external interface of the main model.

b) Operation input and output anchors have been omitted. However, anchors are only used as an edition aid to attach arcs, simplify the writing of mathematical expressions and allow the copy&paste of model sections. Anchor names may be replaced in mathematical expressions by the identifier of the respective driver arcs, as presented in figure 7. In addition, operations with more than one output may be split into multiple single-output operations, cloning the corresponding input arcs. This way, the operation identifier may be used to refer the respective output, leading to an equivalent model without any anchors. Finally, the restrictions applied to signals must also apply the input and output anchors: input anchors may not be driven by more than one input arc and output anchors may not have input arcs.

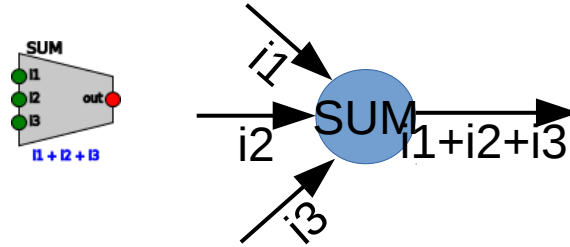


Fig. 7: Anchor equivalence: the operation on the left is equivalent to the dataflow node on the right.

The external interface of a DS-Pnet system is defined by a set of input and output signals and events, that are a subset of the system's signals and events.

Definition 2: The external interface of DS-Pnet system is a tuple $EIF = (IE, IS, OE, OS)$ satisfying the following requirements:

- 1) $IE \subseteq E$
- 2) $IS \subseteq S$
- 3) $OE \subseteq E$
- 4) $OS \subseteq S$
- 5) $IE \cap IS \cap OE \cap OS = \emptyset$
- 6) $\forall s \in IS, \{(x \times s) | (x \times s) \in R\} = \emptyset$ (input signals have no input driver arcs)
- 7) $\forall e \in IE, \{(x \times e) | (x \times e) \in R\} = \emptyset$ (input events have no input driver arcs)

- 8) $\forall s \in OS, \# \{(x \times s) | (x \times s) \in R\} \leq 1$ (output signals have no more than one driver arc)
- 9) $\forall e \in OE, \# \{(x \times e) | (x \times e) \in R\} \leq 1$ (output signals have no more than one driver arc)

In addition to the external interface, a DS-Pnet system may be decomposed in two parts, the state control logic and the data-processing parts:

Definition 3: The state control part of a *DS-Pnet* is a low level Petri net defined by a tuple $PN = (P, T, A, m_0, w, tp, R^-)$ where:

- 1) P is the DS-Pnet set of places
- 2) T is the DS-Pnet set of transitions
- 3) A is the DS-Pnet set of Petri net arcs
- 4) m_0 is the initial place marking mapping $m_0: P \rightarrow N_0$
- 5) w is the DS-Pnet arc weight mapping $w: A \rightarrow N$
- 6) tp is the DS-Pnet transition priority mapping: $T \rightarrow N_0$
- 7) R_p is a subset of the DS-PNet set of read arcs such as $R_p \subseteq R \wedge R^- \subseteq (P \times T) \cup (T \times T)$ (test arcs and synchronous channels)

Definition 4: The data processing part of a *DS-Pnet* model is a dataflow $DF = (O, S, E, R^+, s_0, ex, st, ot)$ where:

- 1) O is the DS-Pnet set of dataflow operation nodes
- 2) S is the DS-Pnet set of signals
- 2) E is the DS-Pnet set of events
- 4) R_D is a subset of the DS-Pnet read arcs $R_D = R - R_p$
- 5) s_0 is the initial signal values partial function $s_0: S \rightarrow N_0$
- 6) ex is the DS-Pnet operation expressions function
- 7) st is the DS-Pnet signal types function
- 8) ot is the DS-Pnet operation results type function

The results of operation mathematical expressions (ex in definition 1) must belong to one of the data types: Boolean, event or integer range. A fixed point integer range data type is planned for the future, but has not been implemented in the current version.

Operation mathematical expressions are constructed using the following items:

- Literal operands: decimal values or hexadecimal values (prefixed with «0x»)
- Variable operands corresponding to the graph nodes directly connected through input arcs (graphically attached to input anchors)
- The arithmetic operators +, -, *, / and MOD, plus the unary operator -
- The comparison operators <, <=, >, >=, <> and =
- The logical operators AND, OR, XOR and the unary operator NOT
- The bitwise operators ANDB, ORB, XORB and NOTB
- Sub-expressions inside curly parentheses

- The delay operator ($[-N]$) used as suffix to variable operands, to refer past values from previous execution steps
- The array index operator ($[+i]$) used as suffix to tables of constant values stored in operations⁷
- The conditional operators WHEN and OTHERWISE to build «case» constructs

The state of a DS-Pnet system is composed of the Petri net place marking and the previous values of signals, events and operations, used in association with the delay operator, memorized as shift-registers:

Definition 5: The state of a *DS-Pnet* model is tuple $MS = (m, ps, pe, po)$ where:

- 1) m is the place marking, $m: P \rightarrow N_0$
- 2) $ds \subseteq S$ is the subset of signals associated with the delay operator
- 3) $de \subseteq E$ is the subset of events associated with the delay operator
- 4) $do \subseteq O$ is the subset of operations associated with the delay operator
- 5) $ns: ds \rightarrow N$ is a function mapping signals to the index of the oldest value accessed with the delay operator
- 6) $ne: de \rightarrow N$ is a function mapping events to the index of the oldest value accessed with the delay operator
- 7) $no: do \rightarrow N$ is a function mapping operations to the index of the oldest value accessed with the delay operator
- 8) ps is a function mapping signals to shift register arrays storing the previous signal values $ps: ds \rightarrow [SR]_{ns}$
- 9) pe is a function mapping events to shift register arrays storing the previous events values $pe: de \rightarrow [SR]_{ne}$
- 10) po is a function mapping operations to shift register arrays storing the previous operation values $po: do \rightarrow [SR]_{no}$

3.6.2 Execution semantic rules

The execution semantic rules of DS-Pnet models, combining characteristics from low level Petri nets [30][31] and synchronous dataflows [139][142], reflect the formalism heritage. In the same way as the parent formalisms, execution is performed in discrete steps, but the computation of each step is considered instantaneous, with no propagation delays between nodes. In typical implementations steps occur at a fixed frequency, with a predefined time interval between consecutive steps, but variable frequency implementations are not discarded.

Although the computation of each execution step is considered instantaneous, observing the synchronous paradigm [142], the evaluation of net nodes, including dataflow operations and Petri net transitions, must be performed under an exact sequence, that is defined using the concepts of micro-step and nano-step numbers.

On reactive systems, that respond to external events and changes in input signals, the most important part of the system state is the Petri net place marking. Previous

⁷ May be used for software or hardware implementation of mathematical functions of 1 or 2 integer arguments, based on tables of values.

values stored in shift registers perform a secondary role, to generate internal events associated with the crossing of certain thresholds, or to calculate values that evolve continuously in the time-domain, as counting, differentiation and integration. Thus, the controllers are designed around a Petri nets, that evolve according to transition firing. A transition may only fire (and must fire), when it is simultaneously enabled and ready. The following definitions cover the rules that govern transition firing:

Definition 6: A transition is enabled when every input place, connected through Petri net arcs, hold a number of tokens that is equal or larger than the respective arc weights. For a transition t : $\forall(p \in P \mid (p,t) \in A), m(p) \geq w(p,t)$

Definition 7: A transition guard-condition is defined by an input read arc, originating on a node containing a Boolean value. Range values are evaluated as true when different from zero.

Definition 8: A transition input event is defined by an input read arc, originating on node of type event: an event, a transition or an operation producing a result of type event.

Definition 9: A transition is ready when all guard conditions and input events hold true.

Definition 10: Maximal step execution semantics - all transitions simultaneously enabled and ready are forced to fire on the next execution step.

Definition 11: Conflict – two or more transitions are in conflict when all of them are simultaneously enabled (and ready), but the number of tokens on the shared input places is not enough to fire all of them.

Definition 12: Conflict resolution – Conflicts between transitions may be solved assigning priorities to each transition. Firing priority criteria is defined by: 1) execution micro-step, 2) transition priority and 3) transition unique identifier.

A read arc starting on a place transmits the number of tokens on that place. If this read arc ends on a transition, forms a special type of guard function, called a test arc, meaning that this transition will only fire when the place is marked, but no tokens are removed from the place. To test if a place has multiple tokens requires a different guard condition, created using a dataflow operation to compare the number of tokens with the desired value.

In the same way, read arcs starting on transitions transmit events, called transition output events. These events, triggered when the transition fires, may be used as input to dataflow operations, to trigger actions on other components or may be connected to

other transitions. When an event triggered by a transition is used as input for other transition, a synchronous channel is formed. The master transition, emitting the event, will fire as soon as it is ready and enabled, independently of the slave transition. However, the slave transition can only fire when the master has triggered an event. This way, both transitions will fire on the same execution step⁸.

Synchronous channels are often used to synchronize transitions located in different components. Systems designed using multiple components may contain long chains of master-slave transitions, but these chains may not create cyclic dependencies. As all transitions from these chains may fire on the same execution step, the corresponding transition firing semantics rules must be evaluated on the same execution step. However, as the firing of the slaves depend on the firing of the masters, the execution semantic rules must ensure that the masters are evaluated before the slaves. In the same way, any dataflow operation that receive events from a transition must also be evaluated after deciding if the transition is about to fire. To resolve this problem, the concepts of micro-steps and nano-steps were introduced, permitting the definition of a precise evaluation sequence to decide transition firing and compute dataflow operations.

Definition 13: Micro-step number assignment:

- 1) Nodes with no input read arcs are assigned to micro-step 1
- 2) Nodes with input read arcs are assigned a micro-step number corresponding to the maximum micro-step associated with these input read arcs, according to the following rules:
 - a) Read arcs used inside mathematical expressions in association with the delay operator «[-n]», are assigned to micro-step 1, as the expression uses memorized values from previous executions steps.
 - b) Read arcs starting on a transition, propagating transition output events, are assigned a micro-step number equal to the transition micro-step plus 1.
 - c) Read arcs starting on non-transition nodes, are assigned the same micro-step number as the source node.

Nano-steps are used to sequence the dependencies between dataflow operations evaluated on the same micro-step:

Definition 14: Nano-step number assignment:

- 1) Nodes with no input read arcs are assigned to nano-step 1
- 2) Nodes with input read arcs are assigned a nano-step corresponding to the maximum nano-step number of the input read arcs source nodes, according to the following rules:
 - a) Read arcs used in inside mathematical expressions in association with the delay operator «[-n]», are assigned to nano-step 1
 - b) Read arcs starting on nodes from past micro-steps, including all places and transitions, are assigned nano-step 1

⁸ Assuming both components are running locally on the same time domain. On distributed implementations the slave may fire later due to communication delays and differences in execution step clocking. In that case the channel is no longer synchronous.

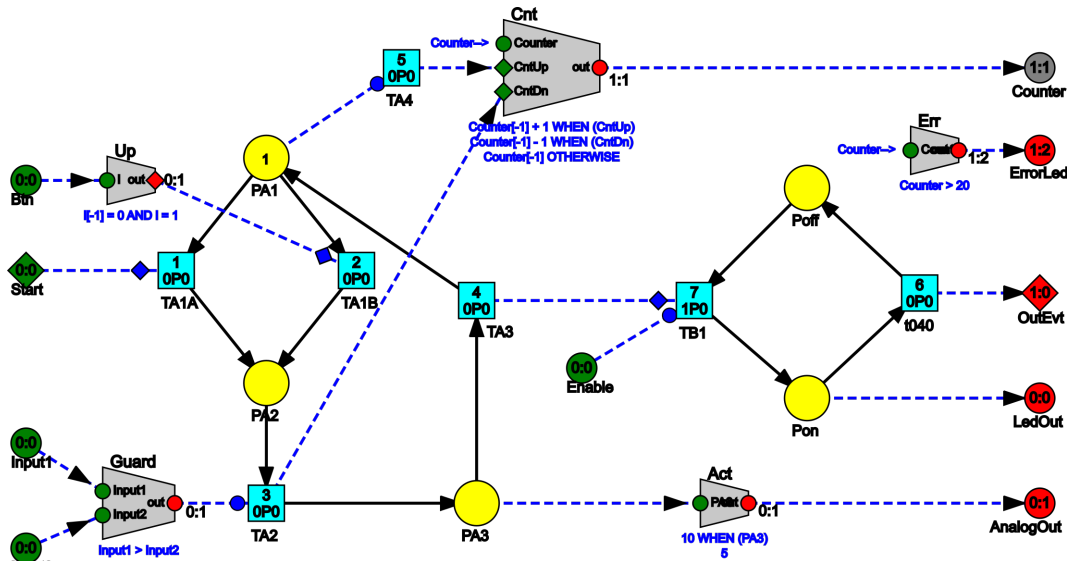
c) Read arcs starting on nodes from the same micro-step, are assigned 1 plus the source node nano-step number

Lemma 1: Dataflow operations nodes with the same micro-step and nano-step numbers can be evaluated by any execution order or may be executed in parallel.

The first stage in the execution of DS-Pnet model consists in the assignment of micro-step and nano-step numbers to all transitions and dataflow nodes, according to definitions 13 and 14. As a result, these numbers are subsequently used by the automatic code generation tools to schedule all calculations. However, this task has another important effect: any cyclic dependencies between dataflow operations and synchronous-channel transitions are immediately detected, even if the loops form across multiple components. This corresponds to a common modeling mistake that is usually solved using the delay operator. When a model contains components, a flat model with all nodes from all components must be built. The assignment of micro-step and nano-step numbers, and the detection of cyclic dependencies is performed on the flat model.

Figure 8 displays the same example model as figure 6, with sequence numbering visualization enabled. Signals and dataflow operations present the respective micro-step and nano-step numbers. Transitions show an evaluation sequence number, used for conflict resolution, followed by the respective micro-step number and priority.

In addition to the micro-step and nano-step numbers, it is also necessary to filter which nodes are required for transition evaluation and which nodes are required for output signal evaluation. These filters are required by the automatic code generation tools for software targets, to immediately update the value of output signals after transition firing, before waiting for the next execution step. For instance, output values depending on place marking may have to be immediately recalculated after place marking changes.



In order to optimize the system execution, computations are split into two stages. On a first stage, only the nodes required for transition evaluation are calculated. Next transitions are fired and a new marking is calculated. Finally, all other nodes are calculated, but the nodes that are simultaneously used by transition evaluation and output signal computations, may have to be calculated twice in the same execution step. This is required to ensure consistent behavior between software and hardware targets, and the last implements dataflow operations using combinatory logic.

It is important to notice that the two stage computation process might cause effects unexpected at first sight, where the value of an output calculated on the first stage triggers the firing of a transition that immediately changes the value of the same output on the second stage calculation. As this process is instantaneous, due to the synchronous paradigm, an external observer would only see a unique change from a value on the previous step to the final value, and may not understand why the transition fired. As a result, an execution step of a system described by a DS-Pnet may be implemented with the algorithm presented in listing 3.

```

read input-signals, input-events
for-each place do
  avail-marking[place] = marking[place]
  add_marking[place] = 0
done
for micro-step = 1 to n-micro-steps do
  // Comment: Stage 1
  for nano-step = 1 to n-nano-steps[micro-step] do
    if required-by-transition-evaluation
      execute data-flow-operations[micro-step][nano-step]
    end if
  done
  for-each transition[micro-step] (sort by priority,identifier)
  do
    if transition-is-enabled and transition-is-ready
      then
        for-each input-place[transition] do
          avail-marking[place] = marking[place] - arc-weight
        done
        for-each output-place[transition] do
          add-marking[place] = add-marking[place] + arc-weight
        done
      end if
    end if
  done
  for-each place do
    marking[place] = avail-marking[place] + add_marking[place]
  done

  for micro-step = 1 to n-micro-steps do
    // Comment: Stage 2
    for nano-step = 1 to n-nano-steps[micro-step] do
      if not(required-by-transition-evaluation) or required-by-output-evaluation
        execute data-flow-operations[micro-step][nano-step]
      end if
    done
  done

  for-each signal do
    if use-delay-operators(signal) then shift-registered-values(signal)
  done
write output-signals, output-events

```

Listing 3: DS-Pnet execution step algorithm pseudo-code

4 Automatic Code Generation

The main goal of the DS-Pnet modeling formalism is the creation of controllers for embedded systems and build distributed cyber-physical systems (or general purpose digital systems), running on physical hardware devices. In order to execute the models on these devices, the corresponding semantic execution rules must be either interpreted or translated to the native programming languages of the target devices. A compilation strategy was chosen, generating code that implements the model behavior on several programming languages. Currently only C, JavaScript and VHDL are supported, but other languages may be supported in the future, as Java, Matlab, or IEC61131-3 Structured text [168][169], to support programmable logic controllers.

The C programming language was chosen to run models on micro-controllers, industrial PCs, and small computing boards as the Arduino and Raspberry PI. VHDL was selected to implement models on hardware devices as FPGAs or ASICs. The JavaScript code generator is currently only used by the DS-Pnet simulator to run models directly on the Web browser. It generates code to efficiently run a single execution step, invoked by the simulator to execute models step-by-step or continuously run. In the future, the JavaScript code generator may have additional uses, for example to implement remote Web user interface for embedded devices. The JavaScript code can perform the computations required for data visualization and user input validation in the browser, releasing the embedded devices from these tasks, with communication bandwidth savings.

Co-design solutions for hybrid systems containing both reconfigurable hardware and microprocessor units may be implemented by selecting hardware or software targets for each component. However, at this point, the code generator must be called

separately for the software and hardware parts, and the communication between hardware and software components must be coded manually, as it is highly dependent on the hardware details of the target architectures.

In a typical co-design scenario, a software main model runs on the processing system and hardware accelerated components are synthesized using VHDL. The hardware component inputs and outputs used to communicate with the main model must be connected to the bus of the processing system and the main model will read and write these I/Os using memory mapped variables.

The automatic code generation tools employ the algorithm presented in listing 3 and the definitions presented on chapter 3. Code generation is processed into six steps:

- 1 – Generate a flat model containing the elements of all components
- 2 – Define the evaluation sequence, assigning micro-step and nano-step numbers to the flat model elements
- 3 – Create a language independent XML file with the model execution semantics
- 4 – Convert the XML file to the selected target language: C, JavaScript or VHDL
- 5 - Add client/server code for distributed execution and remote debug (C only)
- 6 – Pack all source files and support files into a single compressed file

Figure 9 displays the flow of information from step 1 to 5, omitting step 6 that consists on file packaging and the addition of support files (data-files, makefile, etc.). In addition to the language independent semantic XML code, the code generators obtain auxiliary information directly from the original model: The communication layer of the

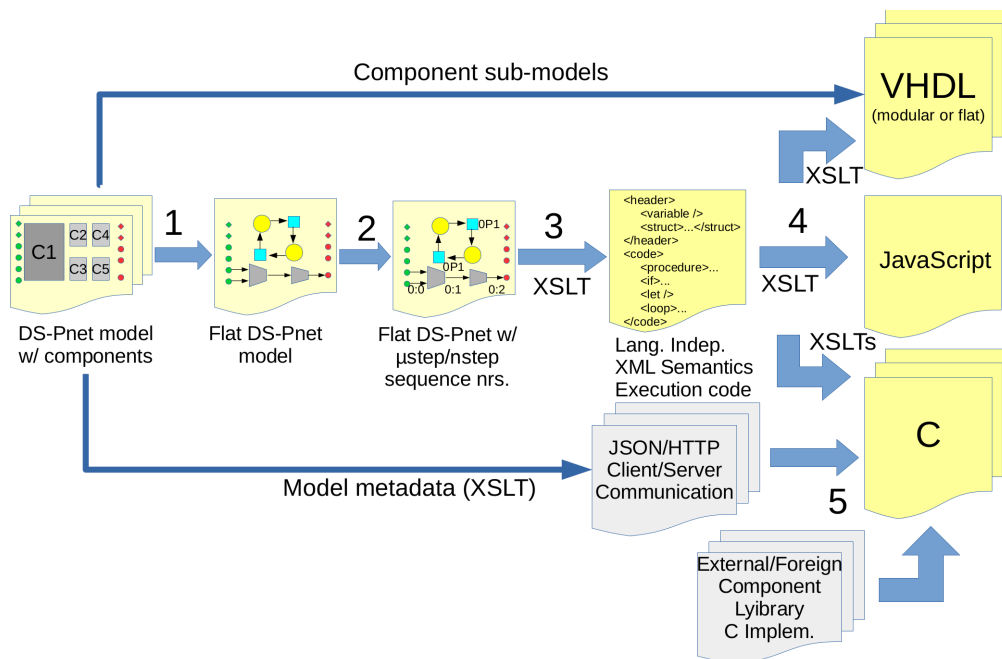


Fig. 9: Automatic code generation information flow (steps 1-5)

C code employs meta-data information extracted from the main model and the modular VHDL code generator applies the code generation algorithm to each individual component.

The first step recursively merges the elements of each component implementation model into the main model, producing a flat model without components, except for foreign/external and remote/distributed components. Element identifiers are constructed appending the original element identifier to the component identifier (ex: «comp1.in3»). The input and output signals and events of the component interface are converted to internal signals of the flat model, and the arcs attached to the corresponding component anchors are reattached to these internal signals.

Step two calculates an execution sequence to compute dataflow operation results and evaluate transition firing. The dependencies between dataflow nodes and transitions are evaluated, assigning micro-step and nano-step numbers to each node, according to definitions 13 and 14. Different instances of the same component element may receive different micro/nano steps due to input dependencies. Cyclic dependencies, possibly crossing multiple components, are detected during this phase.

Next, a XSL transformation is applied to the flat model to generate an intermediary XML document containing instructions to execute the model behavior, according to the algorithm presented in listing 3. The intermediary XML format, independent of any programming language syntax, contains a set of directives, including an header section with data structure and variable declarations and a procedural section with computational instructions. Mathematical expressions maintain the same hierarchical XML format as the original DS-Pnet models. Listing 4 presents an excerpt of this XML code, but complete documents may be viewed using the editor tool.

In the fourth step, other XSL transformations are used to translate the intermediary XML document to the syntax of the target programming language. Currently there are transformations to produce C, VHDL and JavaScript.

XSL transformations [135] had previously been used on a preliminary work [6][8] [14], to generate code from IOPT Petri net models. The new work builds on the experience previously obtained that conducted to the development of the new multi-step code generation strategy. The introduction of the intermediary XML document permits the separation of the code generation in two stages, one dealing only with the semantic execution rules and other with the syntactic details of each target language. This separation contributes to ensure behavioral coherency between all code generators, as the first steps of the code generation are common to all languages. In addition, it also

greatly simplifies the creation of new code generators for different languages, as it just requires translating the intermediary XML files to the new syntax.

```
<execution-semantics model="sample">
  <header>
    <variable name="Input1" orig-node="signal" mode="input" type="boolean"/>
    <variable name="Input3" orig-node="signal" mode="input" type="range" min="0" max="100"
def_value="0" io_pin="0"/>
    <variable name="Input4" orig-node="signal" mode="input" type="range" min="0" max="100"
def_value="0" io_pin="0"/>
    <variable name="Counter" orig-node="signal" mode="output" type="range" min="0" max="1023"
shift-register-depth="1" def_value="0" io_pin="0"/>
    <variable name="InEvent" orig-node="event" mode="input" type="boolean"/>
    <variable name="OutEvent" orig-node="event" mode="output" type="boolean"/>
    <struct name="marking">
      <field name="p1" type="range" min="0" max="255" node-name="P7" def_value="0"/>
      <field name="p2" type="range" min="0" max="255" node-name="P1" def_value="1"/>
      <field name="p3" type="range" min="0" max="255" node-name="P2" def_value="0"/>
      <field name="p4" type="range" min="0" max="255" node-name="P4" def_value="0"/>
    </struct>
  </header>
  <code>
    <procedure name="executionStep">
      <!--Transition T4-->
      <if>
        <condition>
          <operand type="struct" idRef="avail_marking" field="p4"/>
          <operator type="more-or-equal"/>
          <operand type="literal" value="1"/>
        </condition>
        <then>
          <let struct="transition_fired" field="t11">
            <expression>
              <operand type="literal" value="1"/>
            </expression>
          </let>
          <let struct="avail_marking" field="p4">
            <expression>
              <operand type="struct" idRef="avail_marking" field="p4"/>
              <operator type="sub"/>
              <operand type="literal" value="1"/>
            </expression>
          </let>
          <let struct="new_marking" field="p6">
            <expression>
              <operand type="struct" idRef="new_marking" field="p6"/>
              <operator type="add"/>
              <operand type="literal" value="1"/>
            </expression>
          </let>
        </then>
      </if>
      <let variable="o9_out" microstep="1" nano-step="1">
        <expression>
          <operand type="variable" idRef="Counter" delay="1"/>
          <operator type="add"/>
          <operand type="literal" value="1"/>
          <operator type="when"/>
          <operand type="struct" idRef="transition_fired" field="t10"/>
        </expression>
        <expression>
          <operand type="variable" idRef="Counter" delay="1"/>
          <operator type="sub"/>
          <operand type="literal" value="1"/>
          <operator type="when"/>
          <operand type="struct" idRef="transition_fired" field="t12"/>
        </expression>
        <expression>
          <operand type="variable" idRef="Counter" delay="1"/>
        </expression>
      </let>
      <let variable="Counter" microstep="1" nano-step="1" min="0" max="1023">
        <expression>
          <operand type="variable" idRef="o9_out"/>
        </expression>
      </let>
    </procedure>
  </code>
</execution-semantics>
```

Listing 4: Language independent intermediary XML code excerpt.

Cyber-physical systems are frequently designed as networks mixing physical devices and computational nodes that may be located on different locations. Hence, the code produced must have the ability to communicate with distributed components deployed on remote nodes. As components may be located far away, physically placed in inaccessible locations, or running on inexpensive hardware without user interface capabilities, remote debug and monitoring assumes paramount importance.

The automatic C code generator employs a fifth step that appends a minimalist HTTP server to the code generated automatically. This server includes a static part, independent of the model, that implements a simplified version of the HTTP protocol, and a dynamic “reflection” part used to obtain meta-information about the DS-Pnet model, including the names and values of signals, events, internal nodes, system status and trace information. For simplified parsing and reduced network bandwidth, data is transmitted in JSON format, using a protocol that will be described later. The server code may be optionally removed from the final executable program, but is required to interface with the Web based remote debugger application and to support distributed applications that must access components running in the generated code.

In a complementary way, a model may use distributed components located on remote node locations. When this happens, client JSON/HTTP code is added to the generated C code to open connections with the remote nodes and automatically manage the communications of events and signal-changes in a bi-directional way.

Finally, the sixth step prepares a compressed ZIP file containing all source code files and additional support files.

4.1 JavaScript generated code

The JavaScript code generator was originally created as part of the IOPT-Flow simulator tool, to enable the execution of DS-Pnet models on Web browsers. However, in the future it may be used to create components for distributed applications where some components may run on remote physical devices and other components on Web browsers, enabling the creation of Web user interfaces to remotely monitor and control cyber-physical systems or build SCADA (supervisory control and data acquisition) applications. To achieve this, the user interface component library currently supported by the C code generator, must be ported to JavaScript and HTML. A communication layer over JSON/HTTP to talk with remote components, must also be added to the code automatically generated. A similar communication layer written in JavaScript, is already part of the IOPT-Flow remote debugger application.

The output of the code generator is a single JavaScript file containing two functions, responsible for data initialization and the execution of a single step. It also contains data structures with information about the model, including input and output signals and events and internal data as place marking, transitions fired and the current value of dataflow operations.

The simulator/debugger application user interface displays the model data and provides a graphical interface to change the values of input signals and events. Before calling the execution step function, the simulator application must set the appropriate values on these data structures, including input values selected by the user and system state variables. The step function operates on these data structures, updates the state variables (ex. place marking) and defines new output values that are subsequently displayed by the simulator application.

4.2 VHDL Generated code

Two variants of the VHDL code generator were created: one produces a single module containing a monolithic implementation of the flat model, and another produces modular code, generating a main module and additional modules for each class of DS-Pnet component instantiated in the main module. In the second case, multiple VHDL files are generated. The main module (and any module containing sub-components), includes component declarations, instantiation and the respective port mappings.

The external interface of the VHDL modules starts with three input signals: clock, reset and enable, followed by the list of input and output signals and events present in original DS-Pnet models. These VHDL modules may be synthesized into hardware and used as standalone applications, or as components of larger DS-Pnet applications. However, these modules can be used as any other VHDL entity and may be applied in other projects developed using traditional hardware description languages.

Internal signals are used to hold the values of the DS-Pnet elements, including place marking, dataflow operation results and shift-registers (to store values from previous execution steps). As external input signals, coming from the outside world, may change at any point during a clock cycle, potentially leading to random execution errors, the code generator creates internal copies of these signals, sampled at the begin of each clock cycle. To avoid the propagation of glitches produced by combinatorial logic, the external outputs of the main module are also synchronized with the clock signal, resulting in an additional delay of one clock cycle.

Execution semantics is performed by a VHDL process whose syntax offers some similarities with an imperative programming language, simplifying the translation from

the XML intermediary code to VHDL. However, contrary to an imperative language that sequentially executes instructions, a VHDL process is synthesized as a series of hardware registers, multiplexers and combinatory logic. In the specific case of the code generated automatically, the dataflow operations are synthesized as combinatory logic, and only the system state variables are stored in registers: place marking and the shift registers holding past values, whose attribution directives are synchronized with the clock.

Execution is performed at one step per clock cycle. The maximum allowable clock frequency depends on the hardware employed and the length of the chains of consecutive dataflow operations present in the models, that may sometimes cross multiple components. Long chains of dataflow operations that reuse values from previous nodes may impose a limit to the maximum clock frequency. Fortunately, the HDL synthesis tool-chains supplied by reconfigurable hardware manufacturers usually provide analysis tools that calculate the maximum frequency.

When the desired frequency is exceeded, the delay operator may be used to break these chains of dependencies between dataflow operations. This way, instead of using a value calculated in the present clock cycle, the dataflow operations will use registered values calculated in the previous execution steps, corresponding to a pipelined implementation where the operations before the delay operator are calculated in parallel with the operations after. The final results will be delayed by one clock cycle, but the operating frequency may be increased.

When designing models for hardware implementation, special attention should be paid to integer ranges used to store the results of arithmetic operations, as the hardware synthesizers do not generally deal well with situations where the number of bits of the result differs from the «natural» number of bits produced by the arithmetic operators involved. For example, VHDL arithmetic additions and subtractions will produce a result with the same number of bits as the larger operand, and the hardware synthesizer may ignore additional bits on the result. This way, if the signal used to store the result is larger than the operands and the results are negative, the signal may not propagate to the most significant bit. The same problem happens with multiplications and divisions, that produce results whose size corresponds to the addition/subtraction of the operand sizes. Although the generated code employs VHDL integer range types, the problem still happens, and future versions of the code generator may emit warnings to the user.

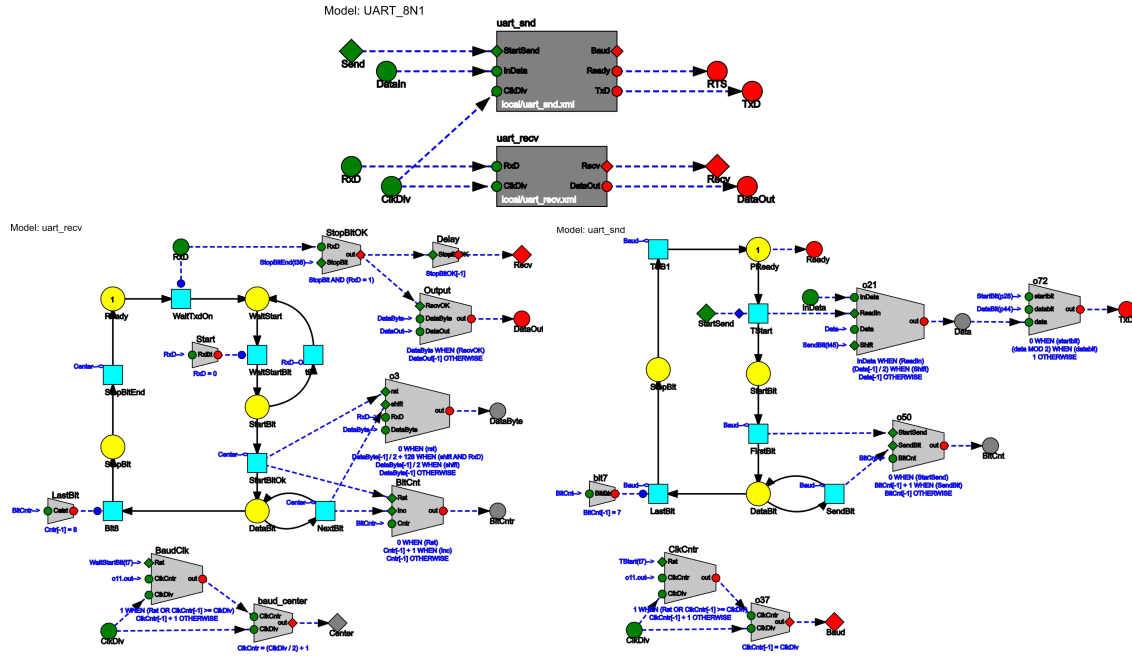


Fig. 10: A UART model (top) with the receiver (left) and sender (right) component implementation models

Figure 10 shows UART model used to test and debug the VHDL code generator. The implementation of the receiver and sender components are shown at the bottom. The resulting VHDL code was synthesized in hardware and tested on a Xilinx Spartan-3AN FPGA board to communicate with a personal computer. The communication baud rate is calculated dividing the main clock signal by the value of the «ClkDiv» input.

4.3 C Generated code

The C code generator plays a central role in the implementation of distributed cyber-physical systems. In addition to the model execution semantics, the output code also contains the infrastructure responsible to establish the communication between distributed nodes, including a JSON/HTTP based client/server code. At present, the other code generators lack this communication infrastructure, but in the near future, the JavaScript code generator might borrow the client part of the remote debugger application, to enable the creation of applications with user-interface components running on Web browsers.

Regarding hardware implementations, there are no plans to directly add HTTP communication capabilities to the VHDL generated code, as TCP/IP based protocols are usually implemented using a software layer running on microprocessor units. A more efficient strategy to access hardware components from distributed cyber-physical networks would employ a co-design solution, with an interface top model implemented in software, containing the hardware components. With this solution, the hardware components would be implemented using VHDL, but the interface model would be

implemented using the C code generator, automatically providing remote access to the hardware components. The software part of the solution could be implemented as a “soft” processor core, embedded in an FPGA, or using hybrid processor/FPGA chips like the Xilinx Zynq [170].

The output of the C code generator is composed of three parts: the model execution semantics, a low-level digital input/output interface and an optional communication layer. The execution semantics code was designed to run on small devices, including 8 and 16 bit micro-controllers with reduced memory capabilities. This way, all data structures are encoded as bit-fields, where the number of bits was optimized according to the range of the integer signals, or maximum place bounds. However, the communication layer was designed to target the IoT devices available today, as the Intel Edison or Raspberry PI, that employ 32 bit architectures and run an operating system with a proper TCP/IP stack. The client/server code was tested on embedded Linux platforms, but the code employs only standard sockets function calls and porting to other embedded TCP/IP stacks or operating systems as Windows or IOS, should be straightforward.

Table 5 lists the generated files and the respective purpose:

Sub-system	File	Description
Model semantics execution	model_types.h *	Constant and data structure type definitions, plus function declarations
	model_exec_step.c *	Data initialization and execution step functions
	model_main.c *	Hardware setup, execution loop and main functions
	model_io.c *	Input and output functions to interface with physical hardware. May be edited by the user.
External / Foreign components	extern_comp.c *	Function stubs to interface with external components. Two functions init() and step() are created for each class of external components, whose body must be manually coded
	comp_lib.c	Standard library components implementation (only used components are linked)
GPIO	dummy_gpio.c	Simulated GPIO using text files. Inputs are read from a file and outputs written to other file
	linux_sys_gpio.c	Digital input and output operations for embedded Linux boards, using the /sys file-system
	raspi_mmap_gpio.c	Digital input and output for raspberry PI family of boards, using faster memory mapped IO
Model meta-data [optional req. by client/server]	model_info.h	Data structures and function declarations to extract meta-model information about the model
	model_info.c	Part of the meta-model information subsystem that is independent of the models, common for all models
	model_metadata.c *	Part of the meta-model information subsystem that depends of each model (node names, etc.)
	dist_comp.c *	Meta-data information about remote/distributed components in use (for HTTP client).

JSON/HTTP communication (server) [optional]	http_server.h	Data-structures and function prototype definitions for the http mini-server
	http_server.c	HTTP server main functions
	http_req.c	HTTP server remote procedure call request implementation
	conn_auth.h	Connection and user authentication data structures.
	conn_auth.c	Connection and user authentication code
	sha1.h/sha1.c	Public domain SHA1 cryptography hash implementation
JSON/HTTP communication (client) [only if model uses remote components]	http_client.h	HTTP client data-structures and function prototypes
	http_client.c	HTTP client implementation (requests/answers)
	cps_net.h	ComPonentS-network – connection to remote components – data structures
	cps_net.c	ComPonentS-network – connection to remote components – implementation code
Data files	user_db.txt	User data-base (IDs, user-names, passwords, privilege-levels and priorities)
	node_db.txt	Distributed node database (map logical node names to network-addresses/user/port)
Utility programs	create_user.c	Utility program to add a new user to user_db.txt with password encryption
	chg_pass.c	Utility program to change a user_db.txt password
Project	Makefile	Compilation «makefile» to build executable files (on Unix/Linux systems)

Table 5: C code generator output files

NOTE: All files marked with «*» are created dynamically using XSL transformations. All remaining files are common to all models.

The model execution semantics code, found in file «model_exec_step.c» is the result of a direct translation of the language independent XML code produced in step 3 to C. The data-structures declared in «model_types.h» include the model inputs, model outputs, internal signals and dataflow operation data, place marking, fired transitions and the array shift-registers used to store past values used with the delay operator. When the models employ foreign components, a new data structure is created for each class of foreign components, with the respective inputs, outputs and parameters.

File «model_main.c» includes three functions. A setup function used to initialize internal values, perform hardware setup and initialize remote communications, a loop function that runs a complete execution step, including reading hardware inputs, call the semantics execution code, write outputs and process remote JSON/HTTP requests and subscriptions. The loop function may be employed directly on Arduino systems that have a builtin main function. For other systems, a main function is provided. The loop function includes a minimalist solution to pause execution or trace step by step based on a single trace_control step counter, to support the remote debug and monitoring application.

All input and output operations that depend on the target hardware were isolated on a single file «model_io.c», that may require manual edition. This way, when a model suffers changes, it is recommended to replace the other files by the new generated code, except the «model_io.c» file that contains manually written code.

Two functions are used to read and write inputs and outputs from the hardware, called respectively at the begin and end of every execution step, and may require manual implementation. When the model input and output signals and events are assigned to specific hardware pin numbers, then the code generator automatically fills these functions with the corresponding read/write function calls. In the same way, pins are also initialized with the corresponding input/output direction. When pin assignment is omitted, then a default value is assigned and the user may manually write the input and output code.

Pin operation is performed using the `digitalRead(pin)`, `digitalWrite(pin, value)` and `pinMode(pin, mode)` function calls, native from the Arduino programming library. For non Arduino systems, three different re-implementations of these functions are provided in the `dummy_gpio.c`, `linux_sys_gpio.c` and `raspi_mmap_gpio.c` files. The first, not completely dummy `dummy_gpio.c`, is useful to run the generated code under simulated environments where the user defines input values by editing an `inputs.txt` file and inspects output results on an `outputs.txt` file. The second uses the Linux «sys» file-system to perform GPIO operations and is compatible with virtually all embedded Linux distributions. Finally, a third implementation uses memory mapped IO for the Raspberry PI boards, offering higher performance than the «sys» implementation. Upon compilation, the user must edit the makefile and un-comment the desired I/O implementation.

At the end of input reading, a function `ioptf_applyForcedSignalValues()` is called. This function call is necessary for remote debugging: it allows the networking subsystem to change the value of input signals and events, according to the JSON/HTTP requests received. The input signals and events left unconnected to hardware pins will be automatically assigned to default values. However, remote clients can attach to the local HTTP server and force different values. There are multiple applications for this feature: For example, the unconnected inputs may be associated to buttons, scroll-bars and other widgets from remote user-interface applications; or may be driven by other components from distributed cyber-physical system applications. Even when the inputs are assigned to physical hardware pins, the remote debugger application has the ability to force different values. Input forcing is frequently used on industrial environments

when a sensor is damaged and reads wrong values: carefully forcing the correct value avoids stopping production until a replacement sensor is installed.

Two additional functions `delayPause()` and `finishExecution()` are used to define the execution speed and determine when executions should stop. The first function may employ timing hardware, or equivalent operating system functions, to set an execution pace, for example a step per millisecond. The second function may be used to stop execution when certain objectives were accomplished or when serious error conditions were detected.

The model meta-data subsystem is used to add information about the model elements to the code produced automatically, including the external interface of the model composed of signals and events, the list of components employed and the respective interface, and also internal signals, dataflow operations and Petri net places and transitions. The meta-data information is composed of lists of data-structures with information about each model element, including the name identifier, data-type, node type, allowable value range, default value, current value, etc. Two functions are responsible to exchange information between the internal variables and the meta-data data-structures, enabling the observation of the current status and forcing different values.

Based on the meta-data sub-system, the client/server communication layer was designed in a model independent fashion, that simply invoke the information exchange functions and scan the resulting lists of meta-data. In turn, the HTTP server publishes this information in JSON format, enabling the creation of model independent applications, as the IOPTflow remote debugger. The IOPTflow editor has the ability to connect to remote servers running DS-Pnet models, extract meta-data information and automatically import the components running on those servers to build distributed cyber-physical applications.

4.4 Interface board for industrial applications

As a general rule, the computational boards available for embedded system development and IoT applications, as the Arduino, Raspberry PI, Intel Edison, Red Pitaya, and even the reconfigurable hardware development boards offered by FPGA vendors are not suitable for industrial applications, requiring the help of dedicated interface boards. For example, these boards offer voltage levels for digital input and output signals in a range of 1.8V to 5V, but industrial systems usually operate at 24V. In addition, the digital outputs can typically only drive loads of 10 to 25mA, clearly insufficient to drive relays and pneumatic valves. From another side, industrial safety regulations require the presence of a normally-open enable signal that cut off all output

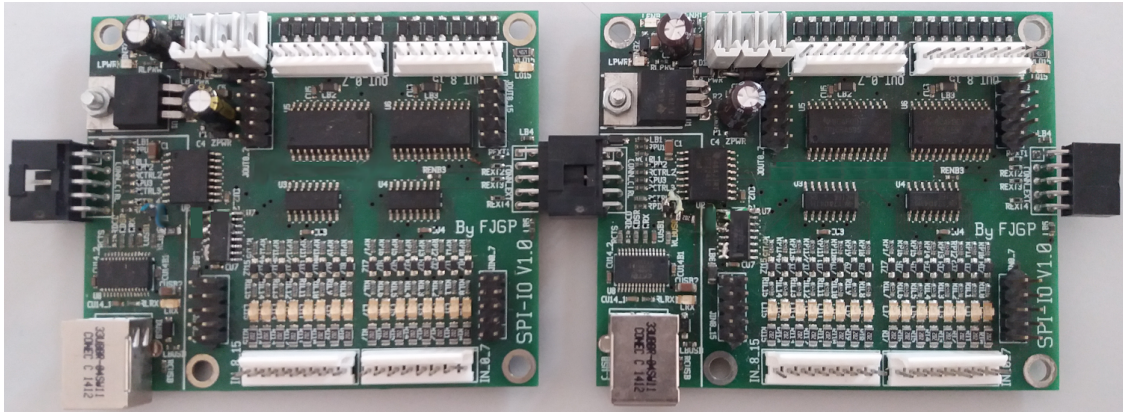


Fig. 11: Isolated digital I/O board w/ SPI interface (2 boards)

signals when a «emergency» circuit is not cleared. Finally, industrial equipment are frequently subject to high voltage spikes and electromagnetic interference, that require galvanic isolation to prevent possible damaging the controller boards.

Although there are commercial interface cards used to drive relays, most these cards do not satisfy the requirements described above, and each of these cards were designed to interface with specific boards, as the Arduino shields and Raspberry Pi hats, requiring the creation of a different driver software for each type of card.

The board presented in figure 11 was designed to provide an universal input/output interface, that can be connected to any development board, FPGA board, or even to personal computers via USB. Digital inputs and outputs are transmitted as shift-registers, suitable for SPI communication ports, where a chip-select signal is used to synchronize the parallel loading of input signals and perform output updates.

The proposed interface card offers the following features:

- 1 - SPI communication interface requiring only 4 IO pins (SCK,SDI,SDO and CS/PL/WE up to 10Mb/s)
- 2 – Optional USB front-end for SPI interface, using the synchronous bit-bang mode of a FTDI FT232R chip
- 3 – Galvanic isolation on both USB and SPI interface ports
- 4 – 16 digital inputs (24V)
- 5 – 16 digital outputs (open collector, 200mA, 1Apeak, 50V max), ready to drive relays and pneumatic valves
- 6 – Daisy chain multiple cards connected to the same SPI/USB interface
- 7 – Normally-connected «enable» input to inhibit all outputs when not enabled
- 8 - 24V Power supply (15V to 100V, 24V nominal)
- 9 – Isolated SPI interface power supply of 1.8V to 5.5V.

By serial chaining multiple boards to the same SPI bus, it is possible to read and write hundreds of digital input and output signals, that can be quickly updated at the start and end of each execution step. The isolated SPI interface occupies just four I/O

pins, permitting the usage of long cables and avoiding the formation of parasitic ground loops, responsible for high levels of electromagnetic interference. The clocking sequences of the SPI protocol may be manually programmed or driven by the hardware SPI master modules offered by many micro-controller devices. An optional FTDI USB/serial converter chip may be installed on the board, providing an USB front-end for the SPI bus, that can be used from any PC or Raspberry-PI USB port. The used FT232R chip offers a bit-banging feature that is frequently employed for the serial programming of micro-controllers, EPROMS and FPGA boards (JTAG). Regarding analog input/output, as most ADC and DAC commercially available chips also offer an SPI interface, it should be possible to create an analog I/O card that may be plugged to the same serial bus as the digital I/O card.

Two surface-mount prototype boards were manufactured and tested. The SPI interface was tested using the processing unit of Xilinx Zedboard FPGA+ARM card, and the USB interface was tested using the FTDI bit-bang library. In both cases the test software was developed in C.

A fourth implementation of the input/output interface code supporting this board may be added to the C code generated automatically. This way, any DS-Pnet (or IOPT) model may be immediately applied on real-world industrial applications, independently of the chosen computational/FPGA development card, or even using a standard PC.

4.5 External/Foreign Components

Foreign components, corresponding the existing physical devices or components implemented using other development languages, must be integrated with the code generated automatically. For hardware implementations, the more effective strategy consists in the definition of an empty component with the exact same external interface of the existing IP module, and just replace the instance of the empty component with the existing IP module before applying the synthesis tools.

On software targets, the usage of foreign components require the manual writing of glue code to interface with these components. For each class of foreign components, the C code generator automatically creates a data structure with the component interface data and two empty functions to initialize data and execute a single step of the component code. The data structure contains the following fields:

- A text string with the component id.
- A text string with the component class name
- The component comment string
- A resource location string parameter

- A «param-string» parameter
- Fields for each component input event and input signal
- Fields for each component output event and output signal
- An auxiliary pointer to hold private data of each instance

The initialization function is called only once before starting the model execution, after the identifier and parameter fields have been defined. This function will typically initialize variables, allocate memory, reserve system resources or open connections to physical devices. By default there is no termination function to release allocated resources, but a global execution finalization function may be used to perform these actions.

During execution, the step function is invoked on every step. When evaluation reaches the correct micro-step/nano-step numbers (the higher combination of the component inputs), the fields corresponding to input signals and events are set with the corresponding values read from the driver arcs, and the step function is invoked. The step function parses the input fields and acts accordingly (preferentially in a non-blocking way), running a single component execution step, and putting the results on the output fields. Finally, the automatic generated code reads the output fields and propagates the new values through the corresponding output arcs, continuing the model execution.

In some cases the same component may be executed twice during the same execution step, once to evaluate the firing of certain transitions and after transition firing to calculate the value of outputs depending on place marking. In addition to a pointer to the instance data structure, the step function receives another argument indicating if the execution is being repeated on the same step.

The IOPT-Flow library includes a set of foreign components, whose code is added to the output of the automatic C generator and may be used as an example to create new foreign components. It includes components to implement arrays and matrices, file IO, random numbers and user interface widgets, among others.

5 Distributed DS-P_{net} Models

Advances in mobile computing and networking technology have reached a point where distributed controller implementations can compete with monolithic solutions with advantages in terms of hardware cost and easy of deployment. However, the traditional programming languages and the respective support tools are not usually well suited for distributed implementations, requiring the use of low level APIs to manually write code to deal with communication and concurrency problems.

In contrast, the DS-Pnet modeling formalism was designed to simplify the development of distributed systems. The model editor tool offers the ability to import components from remote embedded devices running DS-Pnet models and the communication between distributed components is specified just by drawing arcs. All low level communication details are dealt by the automatically generated C code. Using the new formalism, a distributed application is constructed by designing DS-Pnet models containing a network of components, that may run locally or on remote network nodes, whose input and output signals and events are interconnected through read-arcs.

A DS-Pnet distributed model is a GALS (globally asynchronous, locally synchronous) system, composed of multiple nodes connected through the Internet using the JSON/HTTP communication protocol described in the next section. The execution of each node is synchronous and all internal components share the same execution step clock. However, the entire model, formed by multiple components is an asynchronous system, as the nodes employ different execution clocks that may run at different speeds.

This fact has implications in the execution semantics of distributed models. As the Petri net arcs are not allowed to cross component boundaries, the Petri nets contained on any DS-Pnet model are always restricted to a single component. A model composed of multiple components may have many independent Petri sub-nets, not connected by Petri

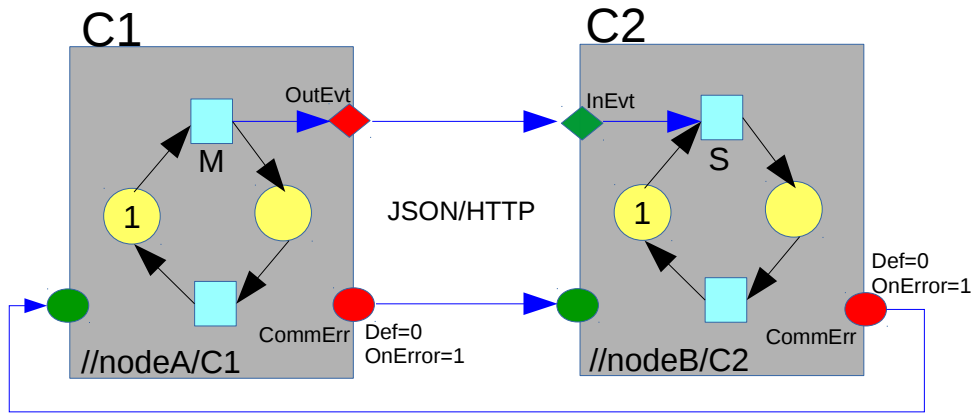


Fig. 12: (Not)Synchronous Channel on a distributed model

net arcs. This way, distributed execution does not affect the execution semantics of each individual Petri net.

However, different Petri nets may be connected using synchronous channels, as presented in figure 12. In this example, the transition M on component C1 fires an event «OutEvent» that is received by component C2 where it is used to inhibit the firing of transition S. When both components run on the same node and share the same execution clock, then the transitions M and S are connected using a synchronous channel and both fire in the same execution step, assuming that both were enabled. In contrast, when the components C1 and C2 are implemented on different nodes, not subject to the same execution clock, the channel is no longer synchronous. The execution semantics of the master transition M is not changed: it fires when enabled and ready, and triggers the «EvtOut» event. Then the communication layer transmits the event to the other node, but the transmission is subject to network latency delays. When the event arrives at the destination node it will be used on the next execution step to fire the slave transaction S, if enabled.

As a consequence, any distributed models employing synchronous channels, relying on the fact that both transitions fire simultaneously, may not behave as expected. The same consideration may be applied to models that expect instantaneous transmission of signal values. To solve these problems, a client-server use pattern borrowed from the IEC61499 should be employed.

Applications may be built using bottom-up or top-down approaches, or employing a mixed strategy. Both local and remote components may be implemented with DS-Pnet models or using traditional development languages. Foreign components may be used to transparently integrate existing legacy code into distributed component networks, without the need to manually write any networking code.

An application constructed from bottom-up may use remote components available on the Internet as building blocks, designed by third parties and already running on existing hardware. For example, in-vehicle navigation systems could collect information from street semaphores in the nearby crossings, to select the best route and adapt speed to arrive at the next semaphore without needing to stop.

In contrast, a designer may choose to develop an entire application using a top-down approach, starting with the design of centralized models composed of multiple components. The centralized application may be debugged using the simulator and model-checking tools, before assigning components to different network nodes (by editing the resource location parameter). Later, a node-split tool can be used to automatically divide centralized models into several sub-models to run on each node. The final distributed solution is built by applying the automatic code generation tools to each sub-model, creating different executable applications for each network node.

The automatic node splitting tool applies the following rules:

- Create a list of nodes according to the different resource locations found in all components, filtering only virtual node names and ignoring components previously assigned to real network addresses
- DS-Pnet elements and arcs that are only connected to components from a single node, will be implemented on that node
- DS-Pnet elements and arcs connected to components assigned to different nodes, are implemented on a main node, called the maestro or application node

The automatic node splitting algorithm always creates a main sub-model (application sub-model), to manage the communication between all the other nodes. However, this solution does not ensure the best performance. For instance, an arc starting on a node and ending on a different node will force the transmission of two messages whenever the value transmitted by the arc changes, one from the first node to the main model and another from the main model to the final node. To optimize performance, a developer may choose to manually design the node sub-models, inserting direct arcs between the nodes. Yet, this solution may bring additional problems, as different network connections may suffer from different latency delays and the inter-node connections may drop, resulting in situations that would be easily managed with a single main model.

The final validation application discussed in chapter 7 presents a small cyber-physical system that was developed using the top-down approach, starting with a centralized model that was split into three node sub-models.

When a top-down approach is employed, it is still possible to reuse existing components, included into the IOPT-flow framework library or available on the Internet. In addition, components that are already being used by other applications may be reused, leading to the formation of complex networks of cyber-physical systems, as presented in figure 19.

When a model is designed using a bottom-up strategy, assembling distributed applications from existing components, it is necessary to assign the correct resource location parameter of each remote component. Components imported when the editor connects directly to the remote nodes are automatically configured with the correct resource location. However, when these components are not yet running, or when using components from the library, the resource location must be edited manually. The format of a component resource location parameter has the following syntax:

user@address:port/comp-id

Where «user» is the user-name for authentication on the remote node, address is an Internet address or a numeric IP address, port is an IP port number that may be omitted (default 9000) and «comp-id» is the component identifier of the remote node. When the definitive address of a component is not known during the modeling phase, then the «user@address:port» part of the resource location can be replaced by a single word to define a virtual node address. A database of virtual nodes is kept in a file «node_db.txt», that is parsed at runtime to postpone the selection of users, addresses and ports until the deployment phase on real hardware. When a component uses the same identifier on both sides, the application model and the remote node model, then the «comp-id» part of the resource location may be omitted. This situation happens with models that were automatically split.

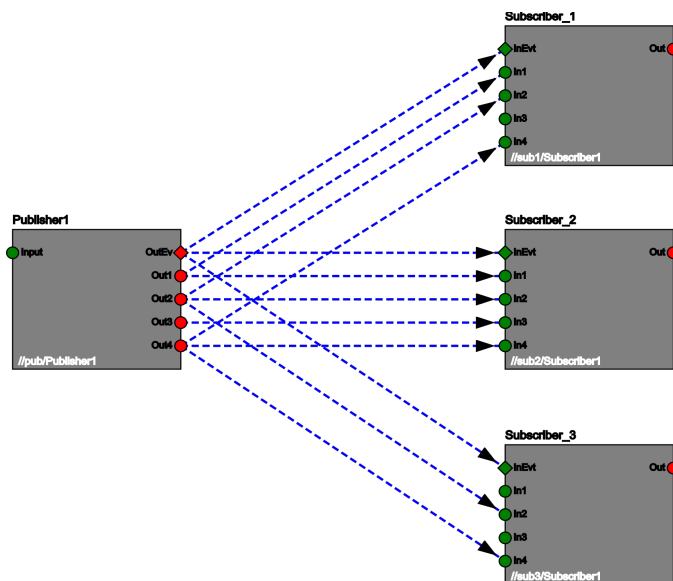


Fig. 13: One publisher and multiple subscribers

The communication between distributed nodes may be signal or event driven. Signal driven communication is typically employed in publisher-subscriber patterns of use, where a publisher makes a set of output signals available and one or more clients subscribe the published information (or just a subset). Although the published information is usually composed of signals, events may also be transmitted.

Figure 13 contains an example where multiple subscribers receive information from the same publisher. In this kind of situations there are no concurrency and synchronization concerns and the published information may be disseminated by multiple clients. As soon as a signal value suffers changes it is immediately forwarded to the subscribers, but there are no timing guaranties and no verification that these signals arrive at destination (except for attempts to reopen broken connections by the networking code).

However, most distributed applications require handshaking negotiations between distributed components to manage synchronization and concurrency problems, ensuring that there is no information loss, data and requests arrive at the destination nodes in the correct order and results are always received.

Signal driven communication could result in data loss if a slow node is receiving information from a faster node. For example, if a signal value changes twice (0 to 1 and back to 0) during the execution of a single step of the subscriber model, the subscriber model will not be able to detect any signal change and possible events will be missed.

These problems are addressed with event driven communication, used to specify handshaking negotiations between remote components, based on principles borrowed from the IEC61499 standard [69][70]. With this strategy, input and output signals are used to pass parameters and receive results from distributed components, sending events to request the invocation of specific methods on the remote side and receiving other events to signalize the request reception, deny requests or to receive result notifications.

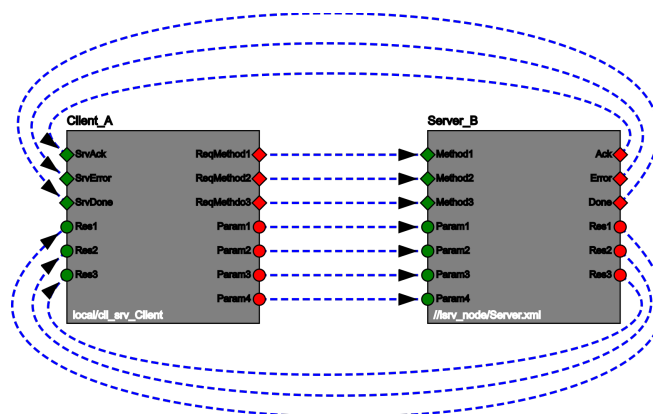


Fig. 14: Client/server event driven communication

The most frequent use case employs a client/server pattern, presented in figure 14, where an application (client) requests the execution of specific methods on a remote component (server) and waits for an answer. As presented in the figure, a server component may implement multiple methods that are triggered by different input events.

The diagram on figure 15 shows the interaction between client and server, that typical perform the following steps:

- 1 - The client assigns parameter values to the remote component inputs
- 2 - The client sends an event to request the execution of a remote method
- 3 - The remote component acknowledges the reception of the event
- 4 - The remote component executes the method and the client waits for the results (while waiting, it may execute other tasks in parallel)
- 5 - The remote component places the results on the respective output signals
- 6 - The remote component sends an event to notify that results are ready
- 7 - The client parses the results and proceeds

The implementation of this usage pattern is not rigid and many variants may occur: For example, when the parameters are inappropriate or a request cannot be executed, the server may produce denial events. When the execution of a request is fast or immediate, the third step may be skipped and the server omits the acknowledge event.

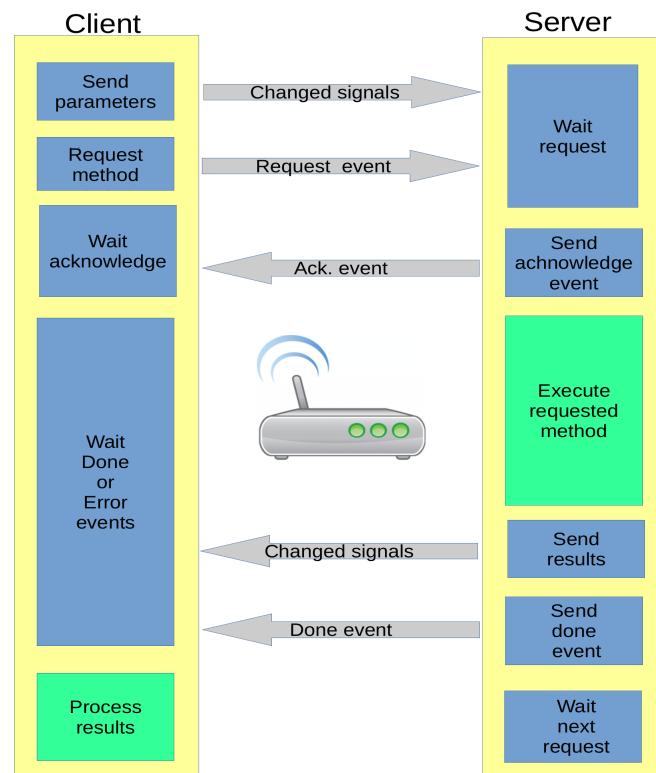


Fig. 15: Client-server communication

Frequently the distributed nodes are continuously executing certain tasks or applications and are not just waiting for client requests. For example, remote nodes may be running real-time industrial machine controllers that cannot be interrupted. In this situation the relationship between the two peers is bidirectional: the application may send commands to the remote node, but the remote side may also spontaneously initiate communication in an asynchronous way. Requests to the remote node usually correspond to high level commands, as start and stop production, set new operating parameters or request statistics. On the opposite direction, the other side may need to report errors and exceptions, notify the lack of bulk materials, or request data required to continue operating, etc.

The proposed formalism does not enforce any type of usage patterns. The system designers are free to implement the appropriate communication handshakes for each specific application, using signal driven or event driven strategies. The communication layer just ensures that events are not lost and the parameters and result signals values arrive at destination on the same step (or before) as the corresponding event.

An important attribute of distributed component input and output signals is the «on-error» parameter, used to define a value that is automatically assigned by the networking layer when a communication error is detected. This value is used to notify models about dropped connections. Two approaches may be employed: a) Define a default neutral value that does not cause malfunctions; b) Define an error value to force an immediate model response, taking the appropriate actions.

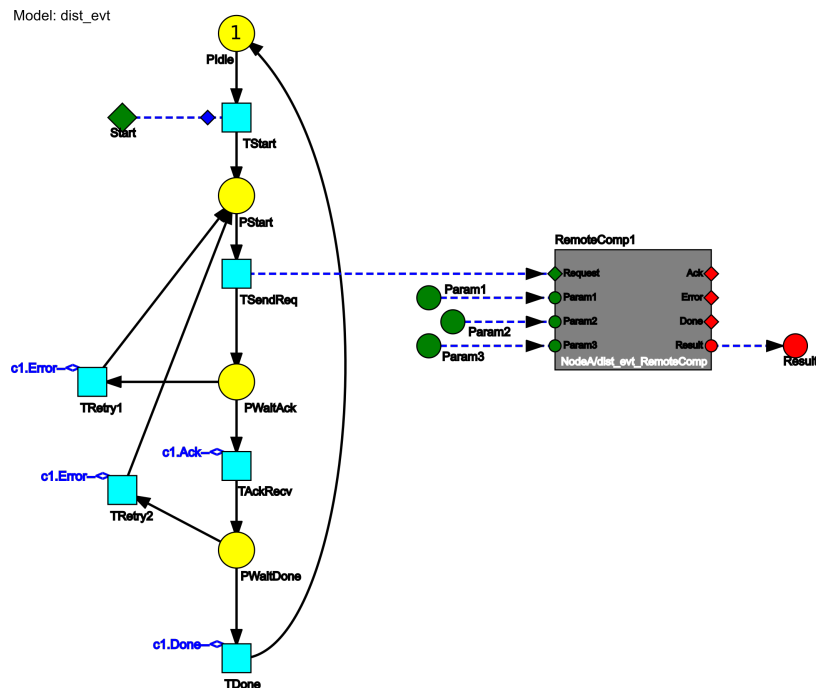


Fig. 16: Example: Event based communication with remote component

The example on figure 16 presents a distributed model using event driven client-server communication to talk with a remote component.

The communication handshake is controlled by a Petri net state machine. This state machine starts with place «PIdle» marked, waiting for a «Start» event that fires transition «TStart». At this point the three parameter inputs must already hold the correct values. In the next execution step transition «TsendReq» is fired, triggering an event that is forwarded to RemoteComp1 (c1). After sending the request, place «PWaitAck» is marked and one of two events may be returned «Ack» or «Error» meaning that the component has started to process the request or the request cannot be processed and one of the transitions «TackRecv» or «TRetry1» will fire. When no errors happen, the system evolves to a state where «PWaitDone» is marked and is waiting for a «Done» or «Error» event. The «Done» event marks the successful completion of the requested action, and the state machine returns to the initial state. When an error occurs, one of the «TRetry1» or «TRetry2» transitions will fire and the «Request» event is sent again.

This example presents a Petri net state machine directly on the main model. However, this state-machine is used frequently to implement client-server handshakes and can be considered a design pattern [171]. This way, the Petri net can be encapsulated into a new component that can be instantiated whenever this pattern is necessary. The handshake controller component requires four input events: «Start», «Ack», «Error» and «Done», where the first is used to initiate the communication and the remaining to receive answers from the server component. Two output events, «Request» and «OK» are used to send requests and inform completion. Figure 17 displays an equivalent model using the handshake component.

The second version of the model hides the complexity of the underlying Petri net state-machine. In real world applications, the handshake controller model would probably be more complex. For example, instead of a single error event, the remote component could return multiple type of error exceptions to differentiate between invalid request parameters, resources temporarily unavailable, authorization violations,

Model: dist_evt_hs

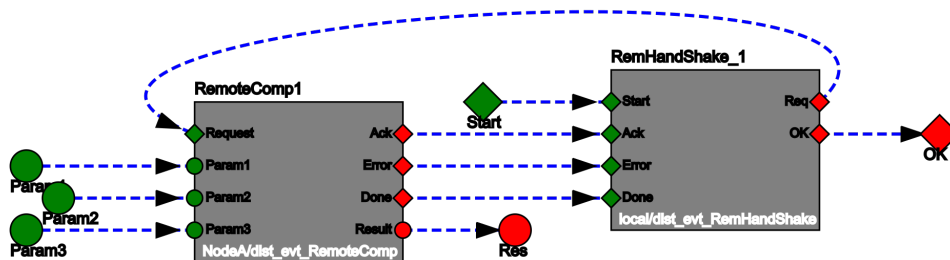


Fig. 17: Same example with the handshake controller Petri net encapsulated in a local component (on the right)

hardware malfunctions, etc., that should be treated differently by the handshake controller. Finally, the model presented in figure 17 could also be used as a component, hiding all communication details with the remote component and providing a simpler interface to access distributed resources. The resulting component may be reused multiple times.

5.1 Shared distributed components

Note: This section describes an extension to allow the shared use of the same components by multiple distributed applications. However, although all steps of the proposed methods seem feasible, none of the concepts presented here has been implemented or tested and are planned for a future IOPT-Flow version.

Public services available on the Internet are generally accessed by a large number of users, sometimes using different client software applications that may access the same service simultaneously. This happens with traditional Internet services as Web (HTTP), Email (SMTP/POP/IMAP), file transfer (FTP, NFS, SCP) and database servers, among many others. In the near future, the components offered as public services to build cyber-physical systems will suffer from the same usage pattern, having to deal with multiple concurrent clients. As a consequence, the development formalisms aiming to provide an infrastructure for cyber-physical systems must also support concurrent component usage.

This section proposes an extension to the existing communication interface to support shared component access by multiple applications. The proposed solution is transparent for the client applications, that continue to apply the same client-server handshaking techniques used for dedicated components, and is almost transparent for the shared component implementation models, that only require two new attributes assigned to input and output events. All the low-level concurrency and synchronization details will continue to be implemented by the networking layer of the C code generated automatically.

In this section, the term «client» refers to an application model using a remote component available on a public or private network. The term «server» applies to components that accept requests from the network, to execute specific methods/tasks. A «transaction» corresponds to a time interval when the server is processing a single request from a client.

Using the event-driven handshaking for remote component access, presented in the previous section, a client application always initiates a communication transaction by sending an event to request a certain resource or method execution. Next, the client waits for an answer event to acknowledge the request reception or notify the successful

request execution. When there are multiple client applications accessing the same component, the component model cannot serve multiple requests simultaneously⁹ and the requests must be processed sequentially. This way, when a client application issues a new request and there are already other requests pending, the client will have to wait longer, until the preceding requests are served, but it still can continue to use the same handshake state-machines, as if there were no concurrent clients.

From a modeling point of view, the proposed solution implies the creation of two new attributes of input and output events, used in the implementation of server component models:

Begin-transaction: A begin-transaction event requests a new transaction (applicable only to input events)

End-transaction: An end-transaction event terminates a running transaction or cancels a pending transaction request. It can be used by the client to cancel requests or by the server to close processed requests, to deny requests, or to notify error exceptions

When designing a new component for concurrent use, the input events used to start requests must be assigned the «begin-transaction» property. The output events used to notify processing termination, or the output events events used for error notification, must be assigned with the «end-transaction» property. Input events with the «end-transaction» attribute may be used by clients to terminate or cancel transactions.

After a transaction starts being processed, the server component is fully dedicated to a single client and the communication between the server and client can flow in a bidirectional way, using both signals and events, until one of the sides triggers an end-transaction event. After a transaction ends, the server can proceed to the next pending transaction request.

From the implementation point of view, the networking layer of the C code generated automatically will have to support the following infrastructure, also presented in the figure 18 diagram:

1 – A prioritized request queue for each shared component. Priorities are already part of the authentication subsystem and may be used to define application profiles, for instance to differentiate between real-time and non critical applications. In a multi-level

9 A multi-thread/multi-core architecture could launch new instances of the component to simultaneously process multiple requests, but parallel solutions are out of the scope of this work

queue, connections with high priority enter immediately to the higher levels that are served first.

2 – Buffered component interface for each client connection: Storing a private copy of the input and output signals and events of the shared component interface for each client. This way, each client sees a different version of the component interface. However these copies remain unconnected to the real component until a transaction starts, and are immediately disconnected after a request ends, avoiding information leaks between clients.

3 – A shared version of the «grab» HTTP request, discussed in the next chapter, used to initiate the component private interface buffers.

Using this solution, when a client assigns new values to the server component input signals, or triggers events without the start/end transaction property, these values are only stored in the private interface copies and are not immediately propagated to the real component. When the component starts serving a client request, these values are copied to the component interface before the transaction starts, maintaining bi-directional updates while the transaction continues. When a transaction ends, the private copy will store the last values assigned during the transaction.

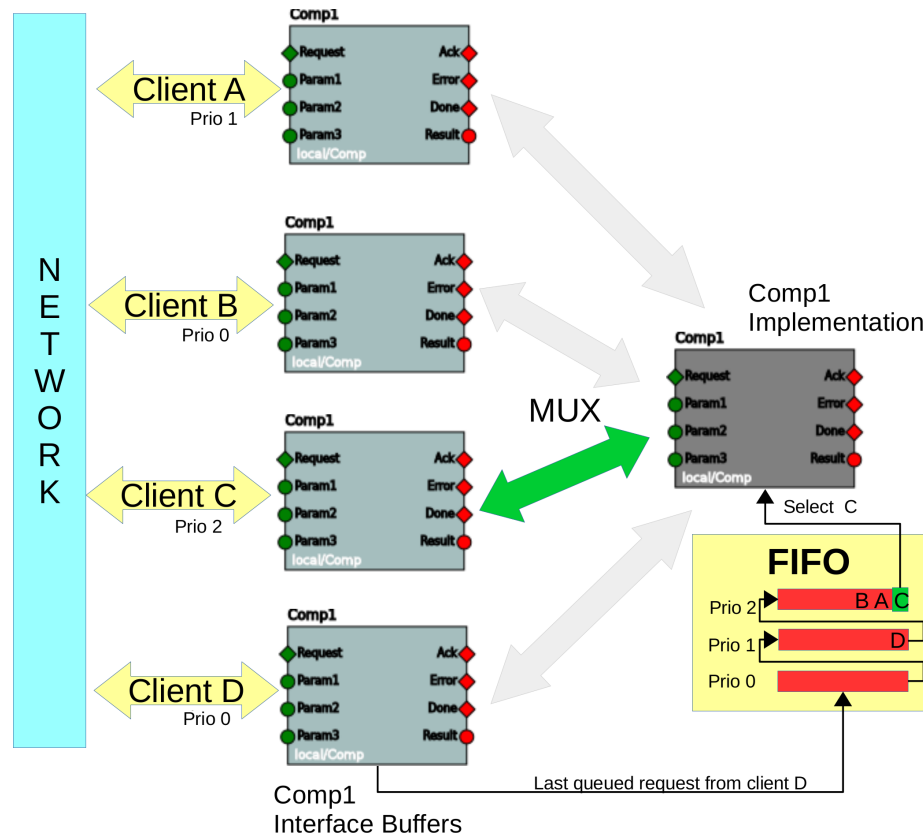


Fig. 18: Proposed infrastructure for concurrent client access

The proposed solution is similar to the implementation of traditional TCP services, where requests are buffered in the operating system TCP/IP socket FIFOs and the server application sequentially extracts one request at a time from the queue. However, these requests are usually atomic, consisting only of a request and an answer message. In contrast, the new solution comprehends a time interval where both sides can dialog, being able to change signal values and send events as would happen in a single client configuration, until a transaction ends. The request FIFO presents similarities with the mailboxes employed by actor systems [66], used to store messages received by an actor in a FIFO, that may employ priorities to dequeue messages from the mailbox. However, in this case the FIFO only contains request events and the messages are stored in each client component interface buffers.

As previously stated, none of these concepts has been implemented and tested. As a consequence, any potential technical difficulties found during the implementation phase might require changes in the proposed solution. For instance, the addition of special-purpose virtual inputs and outputs, whose values are automatically defined by the networking layer could bring improved functionality: a client-id input, automatically set with the user identifier of the current transaction, would enable the implementation of new services that track information about each user. On the opposite direction, a virtual output, automatically set with the number of pending client requests, could be useful to implement load balancing applications.

5.2 JSON/HTTP Communication Protocol

The output of the C code generator contains a JSON/HTTP communication layer to support the interaction between distributed components over the Internet and also to permit the remote debug and monitoring of entire applications or individual nodes. This way, the underlying communication protocol must provide mechanisms to propagate events and changes in signal values through a network of distributed nodes containing DS-Pnet components. These components may interface with physical devices, perform logic and computational operations or provide user interfaces for remote operation, forming distributed cyber-physical systems.

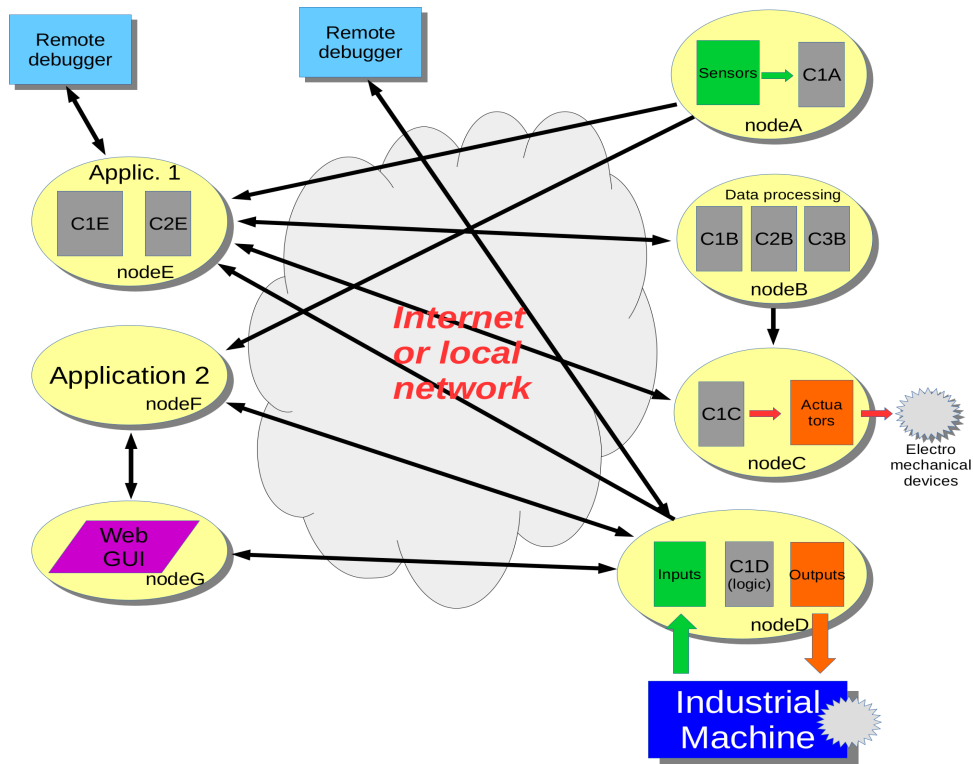


Fig. 19: Possible CPS network topology example

Figure 19 presents an example of a possible cyber-physical system network topology, composed of multiple nodes that execute DS-Pnet components. Some nodes perform computational tasks and other nodes form an interface to read sensors and drive actuators. For instance, «nodeA» contains sensors and the outputs of the component C1A offer a remote interface to read sensed values. In the same way, «nodeC» is attached to mechanical actuators and the inputs of the component C1C provide a remote interface to drive these actuators. At the bottom, «nodeD» represents a typical industrial machine controller, including inputs to read machine sensed data, outputs to drive machine parameters and a logic/processing component to implement real-time control. Usually this controller is responsible for the low level control of the machine, receiving

high level directives from the exterior, as start/stop production, operating parameter settings and statistics gathering. Other nodes may perform only data-processing operations, including the execution of computational intensive algorithms, data logging and storage, or interface with other information system applications, as ERP commercial information systems, factory management software or possible communication with the information systems of customer and supplier companies from a vertical production/commercial collaboration network.

On the left side, two application models communicate with the nodes on the right, receiving notifications about subscribed sensed values, commanding actuator values, accessing data and algorithms running on remote nodes and sending operation directives to remote machinery. A Web based graphical user interface may connect to applications or component nodes to provide control, monitoring and supervision user interfaces for the distributed systems. Finally, a remote debugger application may attach to any of the nodes, either applications or physical nodes, to monitor the internal state of these nodes in quasi real-time and help debug possible mistakes of hardware malfunctions.

As previously stated, although the communication layer has only been added to the C generated code, it may be employed in co-design solutions to bring connectivity for VHDL hardware components. From another side, existing JavaScript communication code from the IOPT-Flow remote debugger application may be easily added to the JavaScript generated code, to access remote nodes from models being simulated and permit the implementation of Web based remote user interface applications.

Communication between nodes is based on the HTTP protocol and information is encoded in JSON notation. The HTTP protocol was chosen for several reasons. First, it does not suffer from traffic routing restrictions as this protocol is normally open on most firewall configurations and routing can be assisted by proxy services, easily traversing intranet/Internet barriers. It is well supported by libraries available on most programming languages and is the natural protocol used by Web based applications. Although HTTP headers may reach hundreds of bytes, these headers do not impose a large overhead, as it can be minimized using server side events and connection keep-alive settings to maintain open connections, that also contribute to increase bandwidth and reduce latency.

An earlier version of this communication protocol [18][22], employed by the IOPT-Tools framework remote debugger, transmitted information using a XML encoding, but was later converted to JSON [129] with large bandwidth savings and simplified parsing on Web based applications.

Based on the TCP/IP protocol, every HTTP request is served using a different TCP connection and broadcasts/multicasts are not supported. In contrast, the internet connectivity offered by IEC61499 publisher/subscriber function blocks [70] is based on IP multicast packets. This option contributes to reduce bandwidth consumption, as only a single message is sent to all subscribers, but may pose user authentication problems and suffers from routing difficulties, as multicast packets are frequently blocked by routers and firewalls. Using the new protocol, a possible alternative solution could be implemented using a dedicated proxy server that connects to remote servers and share the subscribed data to multiple clients on local networks.

In the same way as the second version of IOPT-tools remote debug protocol [22], the connection between client application models and server nodes containing DS-Pnet components (or even other applications), is maintained using two simultaneous connections, referred as two channels. One channel is used to send requests to the remote server nodes and other channel is used to receive notifications from subscribed events and signal changes. Event and change notifications are transmitted using HTTP server side events, sending a stream of event JSON objects over a single TCP/IP connection, avoiding HTTP header overheads. Currently the request channel opens a new HTTP connection for every request, but the HTTP keep-alive option might be used in future versions to use a single persistent connection and further contribute to optimize bandwidth consumption and minimize authentication related traffic.

The applicability of the new protocol to each specific application greatly depends on the available communication infrastructure and the respective performance. As a general rule, long distance Internet connections do not guarantee sustained performance levels, that may vary according to the time of day and the density of network traffic. As a result, real-time applications may require dedicated intranet local networks, while other non critical applications may be implemented across larger networks that spread across the Internet.

As any other Internet connectivity application, security problems must be addressed. Although the main goal of this work does not focus on security issues, a simple security infrastructure was implemented, including user privilege and authentication databases, based on an encrypted challenge-response mechanism to avoid transmitting passwords in open text. This minimal infrastructure was considered enough during the proof-of-concept phase, but additional security mechanisms may be added in the future, probably resorting to standard distributed authentication protocols as Kerberos [172], LDAP [173] or RADIUS [174].

According to the current trends on IoT and CPS, it is forecasted for the near future the availability of open-access infrastructure that may be used to build distributed CPS applications. For example, cities may provide services to publish traffic information on real-time, electric grid information, temperature and wind information, river water flow, water reservoir capacity, etc. However, this information does not need to flow on a single direction. For example, traffic control systems might benefit from information about the routes planned for each vehicle, and public services might provide interfaces to schedule appointments with users, where communication flows in both directions.

The external interface of DS-Pnet components offer a simple interface to implement those services. A service consists of a network node running a DS-Pnet model containing a single or multiple components whose input and output signals and events may be accessed remotely. Applications may be built using DS-Pnet models that import components available on these public servers and connecting the different components using DS-Pnet read-arcs. The IOPTflow editor has the ability to connect directly to any server running the C generated code and import the selected components into new models.

All communication details are dealt by the JSON/HTTP layer of the automatically generated C code. Read-only sensor values may be published as component output signals and events, subscribed by each application, that receive notifications whenever those events are triggered or the signals value changes. Bidirectional communication is synchronized using events, as discussed at the begin of this chapter.

Applications employing existing algorithms, developed using traditional programming languages, may resort to foreign components to encapsulate existing code, using input events to invoke functions and object methods. This way, the integration of legacy code into distributed systems is almost transparent: after writing the glue functions «init» and «step» for each class of foreign components to call the legacy code, the resulting components may be inserted into distributed models just by connecting arcs.

User applications, mostly running on mobile computing devices, may connect to these public services and apply the information for multiple purposes, that probably were not even forethought by the original service providers. Under these circumstances, the service administrators do not control the type of applications using the public services nor the number of simultaneous users concurrently accessing the service. In order to be ready for public infrastructure applications, the user authentication subsystem of the new communication protocol supports multiple privilege levels and priorities, that can be requested by different applications.

5.2.1 User authentication and privilege levels

The user authentication database is used to verify user identities and set limits to the priority requests. Table 6 presents the list of existing privilege levels:

Level	Name	Description
0	Unauthorized	Connection not currently authenticated (default state before successful authentication)
1	Observer	May only read or subscribe signals and events and read internal system data (place marking, etc.), typically used to subscribe sensor information from public services.
2	Client	In addition to level 1, may also concurrently emit events and define input values Reserved for proposed protocol extension described previously.
3	Master	In addition to level 2, may exclusively grab input signals and events and perform trace and debug operations: pause, run single steps and force input values. Used in typical distributed applications that establish a static network topology where a main model obtains exclusive rights over remote component inputs.
4	Administrator	In addition to level 3, may reconfigure the network, disconnecting components from existing network nodes and reconnecting to alternative nodes. May be used to create load balancing and fault tolerance solutions. (reserved for future implementations)

Table 6: Privilege levels

Multiple connections from the same user may request different privilege levels according to role of each application. In the same way, each connection may request different priority levels, and may even dynamically change privilege and priority levels during execution, according to the tasks currently being performed. For example, when the C code generated automatically is opening connection to remote nodes, it checks if the application is driving any inputs and automatically chooses the «observer» or «master» privilege levels.

User authentication is stored on a text file, «user_db.txt», that is used both by the server code to authenticate incoming connections and also by the client code to establish connections to remote nodes. Each user is characterized by a username, password and maximum privilege and priority levels.

Passwords may be stored in clear text or using SHA1 cryptography hash strings¹⁰. Passwords used for public accounts, such as guest, will typically be stored as clear text, while private user accounts should be encrypted. As the user database is shared by both the client and server parts of the networking code, it is possible to associate different passwords to the same username on different nodes. While clear text passwords may be manually edited in the user_db.txt file, the same does not apply to encrypted passwords. This way, two utility programs are also packaged with the C generated code, to allow the creation of new user entries and change encrypted passwords.

¹⁰ The SHA1 algorithm has been recently made obsolete, but may be easily replaced by newer versions, as SHA256. It was chosen due to the public availability of C and JavaScript implementations, that are used by the generated code.

Remote components are addressed using a resource location string, composed of a username, a node name and a component identifier, in the format:

user@address:port/component-id

Where the port number may be omitted (default 9000) and the address may be an Internet address (ex: gres.uninova.pt) or a numeric IP address (ex:192.168.0.1). In alternative, the resource location may be specified using just a symbolic node name, that will be resolved at execution time from the information contained in the file «node_db.txt». This file contains a list of symbolic node names, that are associated with usernames, network addresses and ports. The symbolic node database file is used to permit the configuration of usernames, passwords and network addresses during the deployment of distributed applications, without the need to change models or recompile the C generated code. In contrast, the components imported by the editor from models that are already running on physical hardware will be immediately configured with a definitive resource location (that may be edited).

The fact that user authentication is stored into a text file poses an inherent security risk. This risk is acknowledged and is regarded as acceptable during the proof-of-concept phase: Any person obtaining access to a node of a distributed network may read the authentication file and connect to all nodes sharing the same user authentication data.

At this point, the main security concern was avoid transmitting authentication information over the Internet in clear text. This way, a challenge-response approach was employed, creating encrypted unique session identifiers for each client connection.

5.2.2 Request types

The new communication protocol, inspired on a previous work [18][22], was designed with three goals: automate the insertion of remote components into new models to simplify the design of distributed systems, establish the communication between nodes of the resulting CPS and permit remote debug and monitoring. The list of procedure-call requests, presented on table 7, grouped according to the respective usage type, was designed to minimize network bandwidth and resource consumption on the embedded devices. Some requests are not yet implemented and were reserved for future protocol versions, to support dynamic reconfiguration and concurrent usage of the same components by multiple applications.

As the protocol is based on HTTP, Web browsers might connect to embedded device servers and try to get a root document, like the front page of any Web site. When this happens, the browser is automatically redirected to the IOPT-Flow remote debugger application, that will subsequently open a connection to the server, creating the illusion that the remote debugger application is running on the embedded device.

Every communication session must start with a «login» request, to initialize the user authentication process. The server answers with a random challenge key, used to create a session identifier based on the cryptography hash of the challenge key concatenated with the user password. The session identifier is calculated and stored by both sides, and must be transmitted on all subsequent requests. Future versions of the network protocol, using the HTTP keep-alive option, will use a single persistent connection. This way, the session identifier only needs to be transmitted once, eliminating the risk of address spoofing attacks.

After calculating the session identifier, the client must request the desired privilege level, using the «requestPriv». The remote debugger application employs the «master» privilege level to be able to invoke the trace and debug requests. The client side of generated code automatically selects the correct level for each remote node: when the model only reads data it uses the «observer» level, when it also needs to trigger events or drive input signals, it selects the «master» level. Future management applications might use the «administrator» level to perform load balancing or fault tolerance tasks.

Most requests contain parameters that are encoded as part of the URL «get» query string, including the session identifier. As the size of the parameter data is usually small, it fits in the HTTP request strings. In contrast, the payload of the request answers may reach many kilobytes and is encoded in JSON format. Subscribed events and value change notifications are also transmitted as a stream of JSON server side events.

Group	Priv.	Request	Description
User Authentication	Unauth	login (user)	Start a new session with a user – receive a challenge key
	Observ	logout	Terminate an existing session
	Unauth	requestPriv (priv, prio)	Request new privilege and priority levels (restricted by user's maximum on authentication database)
Model metadata	Observ	getModelName	Get original DS-Pnet model name
	Observ	getModelURL	Get model URL, used to fetch the DS-Pnet model document (used by the remote debugger and editor)
	Observ	enumerateComponents	Get list of available top-level components (sub-components inside other components are hidden)
	Observ	getComponentInterface(c)	List the interface input and output signals and events of a specific component
	Observ	listModelMetadata (components/operations=1/0)	Get model meta-data, including signals and events, marking, transitions and dataflow operation results. Metadata about component interfaces and dataflow operations may be optionally included in the list.
Read data	Observ	getAttrValues(list)	Read current attribute values, including signals, events, marking, fired transitions, etc.
	Observ	subscribeChanges(list)	Subscribe changes of a list of attribute values. The connection will remain open, transmitting a stream of server-side-events (JSON objects) to notify changed values, events and debug/breakpoint/trace status.
Write data	Client	setAttrValues(list, values)	Change the value of a list of model attributes or component inputs (not grabbed by other connection).
	Client	triggerEvents(list)	Trigger a list of events (not grabbed by other connection)
Synchronization	Master	grabInputs(list)	Grab a list of model inputs (or component inputs) for exclusive use (or shared use in the future)
	Master	grabComponent(c)	Grab all input signals and events of a component for exclusive use (or shared use in the future)
	Master	releaseInputs(list)	Release list of grabbed inputs
	Master	releaseComponent(c)	Release all grabbed component inputs
Trace & Debug	Master	resetExecution	Reset model execution, restoring status to initial values
	Master	startExecution	Start/continue execution after a pause
	Master	stopExecution	Pause execution
	Master	execStep(n)	Execute 1 or multiple (n) steps
	Observ	getTraceMode	Obtain the current trace mode: Paused, Running, StepByStep or N steps.
	Master	defineBreakpoints(list)	Set breakpoints on a list of transitions, events, signals or dataflow operations. Signals and operations will trigger breakpoints when the current value changes. An empty list clears all breakpoints.
	Observ	getBreakpoints	Get the list of existing breakpoints
	Observ	getActiveBreakpoint	Get the identifier that caused the last breakpoint
Write / Debug	Master	forceValues(list)	Force new values on a list of input signals/events, even if they are associated with hardware pins.
	Master	thawForcedValues(list)	Release forced values that start reading hardware values
Dynamic reconfig. (Not yet implemented)	Admin	relocateComponents	Disconnect components from current server and reconnect to new server
	Admin	subscribeChangesFrom	Force subscribe-changes from a new server node (at runtime)
	Admin	cancelSubscChangesFrom	Cancel subscribe-changes from a server node (at runtime)
	Admin	pushChangesTo	Force push-changes to a new server-node (at runtime)
	Admin	cancelPushChangesTo	Cancel push-changes to a server-node (at runtime)
	Observ	getQueueSize(comp)	Get the number of clients waiting on a specific component, used to measure client load (for future use)
General	Unauth	/	Default: redirect browsers to the IOPT-Flow remote debugger application

Table 7: Communication protocol request/procedure list

JSON was chosen over XML, due to the reduce bandwidth consumption and simplified parsing on Web based applications. Listing 5 presents an example of a communication session excerpt. HTTP headers have been stripped from the example to improve readability¹¹.

```
get http://host/login?user=guest HTTP/1.0
{"sess_key":"bd59c671049c7bb7fb4e109674cadfe2775b68fb"}

get http://host/requestPriv?
priv=2&priority=5&sessid=65aeldca6b7400b992b8b454c4b8952568f1bb88 HTTP/1.0
{"priv":2}

get http://host/getModelName?sess-id=65aeldca6b7400b992b8b454c4b8952568f1bb88 HTTP/1.0
{"model_name":"ui_test","version":"V1.0"}

get http://host/getModelURL?sess-id=65aeldca6b7400b992b8b454c4b8952568f1bb88 HTTP/1.0
{"model_url":"http://gres.uninova.pt/iotp-flow/files/ui_test.xml"}

get http://host/listModelMetadata&sess-id=65aeldca6b7400b992b8b454c4b8952568f1bb88
{"attributes":[
  {"name":"NV","node":"input","type":"int-range","min":0,"max":65535,"def-value":0},
  {"name":"Page","node":"input","type":"int-range","min":0,"max":15,"def-value":0},
  {"name":"Sens","node":"input","type":"boolean","def-value":0},
  {"name":"Vis","node":"input","type":"boolean","on-error":1,"def-value":0},
  {"name":"Checked","node":"output","type":"boolean","driven":1,"def-value":0},
  {"name":"Disp","node":"output","type":"boolean","driven":1,"def-value":0},
  {"name":"Pct","node":"output","type":"int-range","min":0,"max":65535,"driven":1,"def-
value":0},
  {"name":"Status","node":"output","type":"boolean","driven":1,"def-value":0},
  {"name":"Val","node":"output","type":"int-range","min":0,"max":65535,"driven":1,"def-
value":0},
  {"name":"R","node":"input","type":"event","on_error":0,"def-value":0},
  {"name":"S","node":"input","type":"event","on_error":0,"def-value":0},
  {"name":"ChgEvt","node":"output","type":"event","driven":1,"on_error":0,"def-
value":0},
  {"name":"DEvt","node":"output","type":"event","driven":1,"on_error":0,"def-value":0},
  {"name":"PG","node":"output","type":"event","driven":1,"on_error":0,"def-value":0}]
}
```

Listing 5: JSON/HTTP communication session excerpt

The metadata presented in this example contains only input and output signals, as the model employed does not contain any Petri net nodes. In case Petri net nodes were present, the listing would also include «place» and «transition» nodes. The word «attribute» was chosen as a general designation for the variables contained in the generated code, associated with nodes on the original DS-Pnet model, as place marking, fired transitions, event and signal values, component input and outputs and dataflow operation results.

¹¹ HTTP headers are enabled by default, required to communicate with Web browsers. However, headers are not necessary for the communication between nodes that were built using the automatically generated code. To reduce the overhead, the generated client code immediately disables headers when a communication session starts.

5.2.3 Server

Observing the network topology on figure 19 with multiple nodes, some nodes just provide components designed to build distributed applications, and other nodes run the main “maestro” models that implement the applications. The application models manage information received from some nodes and send instructions to other nodes, respectively by reading component outputs and driving other component inputs. Under this topology, each node runs a copy of a minimalist HTTP server that implements the requests listed on the previous table. Even the application nodes run the server code, in order to allow remote debug and monitoring, or the creation of dedicated graphical user interface to operate and monitor these applications.

Although the communication server code might be disabled at compilation time (makefile options), reducing resource consumption on small hardware devices, it is usually enabled on all nodes. However, the automatic code generator only adds client code to the applications that employ remote components.

In order to support real-time systems, the execution semantics code must run in a predictable way, without interruptions. As a consequence, the communication layer may not block, even during network failure situations. When these situations occur, the communication between different nodes may be temporarily interrupted, but the local code running on each node to perform critical functions should not be affected. An «on-error» field, present in the previous meta-data listing, is used by the communication layer to notify the nodes reading these attributes, automatically assigning the «on-error» value.

To avoid blocking execution, the low level networking code was developed using non-blocking system calls (timed-out selects and socket non-blocking options), and the server functions are invoked in an interleaved fashion. On each execution step the following actions are executed:

- 1 - Process pending HTTP requests (up to max. requests per step)
- 2 - Read hardware inputs
- 3 - Apply forced values (previously forced by the remote debugger)
- 4 – When not paused, run execution semantics code (single step)
- 5 - Write outputs to hardware
- 6 – Check breakpoints and update trace step counter
- 7 - Process subscriptions (send events, changed values, breakpoints & trace status)

The last step, involve sending messages to multiple clients that subscribed value change notifications about component inputs and outputs and internal state variables. However, subscription data is sent through secondary communication channels, using a persistent TCP connection, that do not require waiting for confirmation if data arrives to the subscriber clients. This way, information is immediately written to the client TCP sockets, without blocking the application. The operating system will subsequently send data packets through the corresponding network interfaces, in parallel with the model execution.

Each client connection has a list of subscribed attributes. Subscription notifications are processed using the model metadata data-structures of C code generated automatically. These data-structures, with information about each model attribute, contain fields to store default values, current values, and the previous value (before the current execution step), used to detect values changes.

To minimize network bandwidth, subscribed values are only transmitted when they suffer changes. Information about events and fired transitions is treated in a different way: it is sent when events are triggered, and omitted otherwise. Changes are send as HTTP server side events, encoded as JSON object “deltas”, containing only the changed attributes. Clients must memorize past values and update the new changed values, but events are cleared at the end of every execution step. In addition to the subscribed data, the secondary channel is also used to transmit trace and debug information and an execution step count, since the previous update.

In order to keep connections alive, the server has an idle counter for each connection, that counts the number of consecutive steps without sending notifications: when a predefined number is reached, the server sends a complete message containing all subscribed values. This avoids reaching operating-system timeouts that would automatically hangup the TCP connections and also refreshes the client status, to ensure data consistency at both ends of the connection.

5.2.4 Client

From the opposite direction, the client part of the communication layer is only added to the generated C code when a model contains references to distributed remote components. The client side performs two main tasks: process notifications arrived from remote servers, and propagate events and changed signals to drive the inputs of remote components. These actions directly mimic the read-arcs connecting a main model to remote components: arcs reading information from remote component outputs will be dealt using subscriptions; arcs starting on the main model (or other remote components)

to drive remote component inputs are dealt with HTTP write commands to push changed values.

In the same way as the server part, the local model execution should not block and is also interleaved with the step execution. At the begin of each execution step, the secondary channels of each connection are scanned for new notifications, that are immediately processed, updating the internal data structures with new values. This way, the new values are immediately used in the current execution step. After the step execution has terminated, changed values are immediately pushed to the remote components.

Regarding execution delays, the reception of value change notifications poses absolutely no problem as the network sockets are inspected using non-blocking instructions. All notifications are immediately processed and discarded from the socket queues. However, pushing changed values to remote components employing the «triggerEvents» and «setAttrValues» HTTP requests, currently wait for the respective answers, to be able to retransmit the commands in case an answer does not arrive under a minimal timeout. In order to minimize this problem, the actions of sending requests and response parsing were split into separate functions, sending all requests to the remote servers before start polling for the answers. This way, the operating system may send the requests in parallel, and the answer reception code can use timed-out select instructions to wait for all answers in parallel.

Using this strategy, the maximum wait time to push changes is limited by the slowest connection to the remote servers. A next version of the protocol, using the HTTP keep-alive feature that employs persistent TCP connections to send multiple requests, might eliminate the need to wait for push-change confirmations, that could be postponed to the next execution step. However, with the current implementation, ignoring push-change confirmations could cause undesired consequences due to missed event propagation, that could not arrive at destination. In contrast, persistent TCP connections ensure a sequential stream of data, that arrives precisely by the same order as it was transmitted and both sides are notified when a connection drops.

The client side of the communication code is divided into several parts:

HTTP protocol:	Encode HTTP URL requests, headers, network transmission and parse JSON answers
Authentication:	Parse the node and user authentication databases and calculate session keys.

Notification processing: Process notifications received from remote servers with subscribed events and value changes, to update internal execution semantics data-structures

Event/Change pushing: Propagate events and changed values to drive the inputs of remote components

CPSnet: Setup of a components network, creating a list of nodes and assembling a list of meta-data from all components running on each node

In turn, the CPSnet initialization is divided by the following steps:

1 – Identify a list of all remote nodes and the respective components according to the component resource locations

2 – For each node, assemble a list of all component outputs being read by arcs

3 – For each node, assemble a list of all component inputs driven by the main model using arcs

4 – Open connections to each node, according to resource location (user and network address) using the «node-db» and authentication databases. Wait and try to reopen failed connections until all nodes are online. Log errors and fail in case of authentication failure.

5 - For each node, subscribe the list component outputs prepared in step 2.

6 - For each node, try to grab the list component inputs prepared in step 3. This operation obtains exclusive control over remote component inputs, that cannot be driven by other applications until this connection is terminated. Log errors and fail in case grabs are refused due to conflicts with other concurrent applications.

7 – Try to reopen connections whenever a connection is dropped (during model execution).

On connection termination, all subscribed values and grabbed inputs are automatically dropped by the server, and must be reconstructed by the CPSnet code. The information about remote components and the respective lists of input and outputs connected using arcs, are obtained from the meta-data information. The next listing presents example meta-data form a distributed application used to test this communication protocol. Components C1 and C2 run on the same node, and the respective input and outputs must be combined in single output-subscription and input-push lists.

Dynamic network reconfiguration functionality, planned for future implementations, imply the reconstruction of CPSnet information, including updated lists of nodes, new subscription and push lists, and terminating and opening different connections, as a result from the remote requests received by the server part of the networking code.

```

ioptf_metadata distributed_comp_c1[] = {
  { "c001.I5", nt_input_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c001.I6", nt_input_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c001.I7", nt_input_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c001.O5", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c001.O6", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c001.O7", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { NULL }
};

ioptf_metadata distributed_comp_c2[] = {
  { "c002.O1", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c002.O3", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c002.O4", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { NULL }
};

ioptf_metadata distributed_comp_c3[] = {
  { "c1.O4", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c1.O5", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c1.O6", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c1.O7", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { "c1.O8", nt_output_sig, dt_boolean, 0, 1, 0, TRUE, 0, 0, -INT_MAX, -INT_MAX, NULL },
  { NULL }
};

ioptf_comp_info distributed_components[] = {
  // name resource-location implementation model io-metadata run master-conn
  { "c1", "guest@172.16.3.101:9000/c1", " //_192_168_1_65_9000/local/IOX8.xml",
distributed_comp_c1, TRUE, NULL },
  { "c2", "guest@172.16.3.100:9000/c2", " //_192_168_1_65_9000/local/IOX8.xml",
distributed_comp_c2, TRUE, NULL },
  { "c3", "guest@172.16.3.100:9000/c3", " //_192_168_1_65_9000/local/IOX8.xml",
distributed_comp_c3, TRUE, NULL },
  { NULL }
};

```

Listing 6: Distributed components meta-data information example

6 The IOPT-Flow Tool Framework

The DS-Pnet modeling formalism was designed to enable the development of embedded system controllers and distributed cyber-physical systems. To reach this objective, a tool-chain of Web based development tools was created, providing an integrated development environment that covers all development stages from model design, simulation, model-checking, node distribution, code generation and remote debug and monitoring of the deployed systems.

The IOPT-Flow framework offers the following tools:

- 1) Model editor: Model design and edition, with the ability to import remote components to create distributed applications
- 2) Simulator/Debugger: Model simulator with debug and trace capabilities, waveform visualization and the ability to compare results with previous simulations
- 3) Node-split: Split centralized models into distributed nodes
- 4) Automatic code generation: Generate C, JavaScript and VHDL code to execute the model semantics
- 5) Remote Debugger: Monitor and debug models deployed on remote embedded devices
- 6) IOPT Import/Export: Import IOPT models and export the Petri net part of a DS-Pnet model to PNML
- 7) IOPT Model checking: Use the IOPT model checking framework to analyze the Petri net part of DS-Pnet models

This framework was inspired on previous work, the IOPT-tools toolchain [13] [16], inheriting many concepts and design solutions. Although the frameworks as based on different formalisms, DS-Pnets and IOPTnets, the interactive tools share many algorithms and design solutions that had previously given good results [24][20][21]. From the opposite direction, many usability features implemented on new tools were back-ported to IOPT-tools, resulting in benefits for both frameworks.

The editor tool works as a front-end for all the other tools. It contains buttons to invoke all other tools, including the simulator, code generators and remote debugger. However, the remote debugger is more frequently used by simply directing the Web browser to the HTTP servers running on the embedded devices.

By default the editor stores model files on the server running the IOPT-flow tools, but the user may also download and upload models from the personal computer. When an empty «save-as» file name is entered, the model XML files are transferred to the user PC and the browser will ask for a local file name. Model files may be stored in a public server folder, shared by all users or on private folders. The login dialog contains options to manage personal user accounts.

The tools were implemented using a combination of Web technologies, with the interactive applications running directly on the user Web browser, except the data storage and computational intensive tasks that are executed in the server. As the computational intensive tasks are executed on the server, the tools may be used on low-end computational devices as smart-phones and tablet computers. This way, a technician

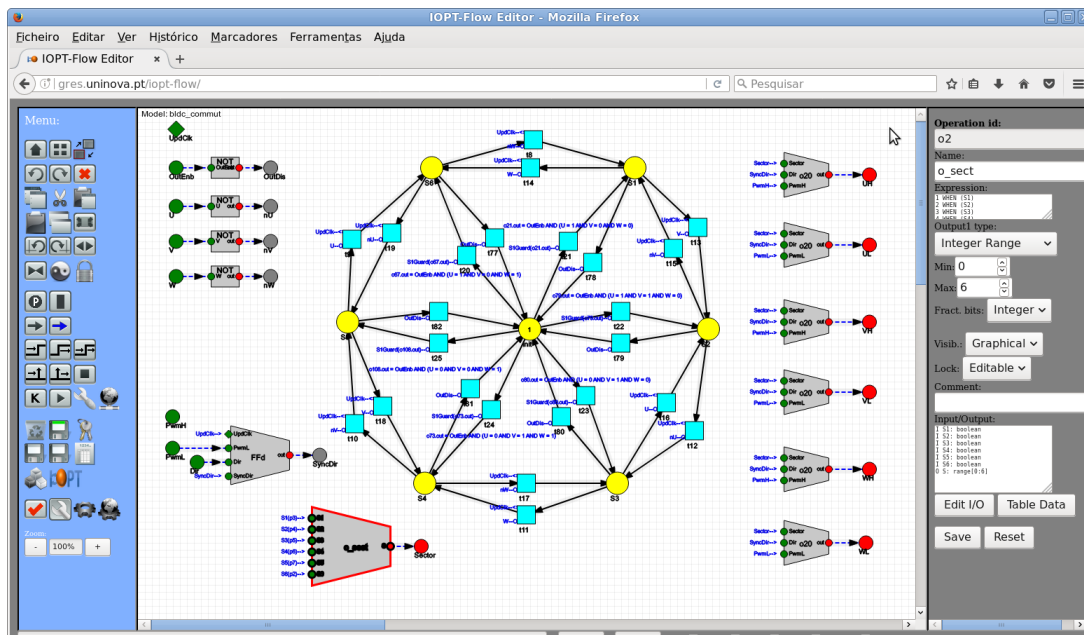


Fig. 20: The IOPT-Flow Editor

on the field may run the debugger application from a mobile device and immediately edit a model to correct any problems detected.

As JavaScript and HTML are the standard languages available on Web browsers, they were used to write the interactive applications. The server side code used to interface with the interactive applications was developed using PHP and performs tasks as user management, file management and security checking. The code generation tools and IOPT state-space computation, also running on the server, are based on XSL transformations and other XML processing tools.

The tools, available at <http://gres.uninova.pt/iopt-flow>, are installed on an Apache HTTP server running over Linux. The interactive tools have been developed using the Mozilla Firefox and Google Chrome Web browsers, but other W3C standards compliant browsers should work, including Opera and Safari. Curiously, when the development of the preliminary work on the IOPT-tools framework started, the Internet Explorer browser did not support most of the technologies chosen to create the tools, including direct support for SVG rendering and the XSLT engine did not employ the standard interface. Meanwhile these technologies have been progressively added to Internet Explorer and some of tools have started to work, but other tools continue to malfunction due to divergences in the interpretation of XSL transformations. Maybe a future version of the Edge browser will someday be able to run the complete tool-chain...

6.1 Editor

Figure 20 presents the IOPT-Flow editor. The editor interface offers a toolbox on the left, a form with the selected object properties on the right and a drawing area at the center. The form entries correspond to the properties of each type of DS-Pnet node, as presented in chapter 3, except those that are manipulated interactively in the drawing area, as the XY coordinates.

As DS-Pnet models are stored in XML documents, the editor works directly on the XML document, operating changes to the model DOM (Document Object Model) tree. Visualization is performed using SVG (Scalable Vector Graphics), a XML based language for graphics representation supported by modern Web browsers.

Figure 21 displays the user feedback loop, including the XML document and the XSL transformation used to generate SVG graphics. Each time the model is changed or new nodes are added, the changes must be immediately reflected in the corresponding SVG document, using a XSL transformation (Extensible Style-sheet Transformation). In addition to the graphical representation of the DS-Pnet elements, the resulting SVG

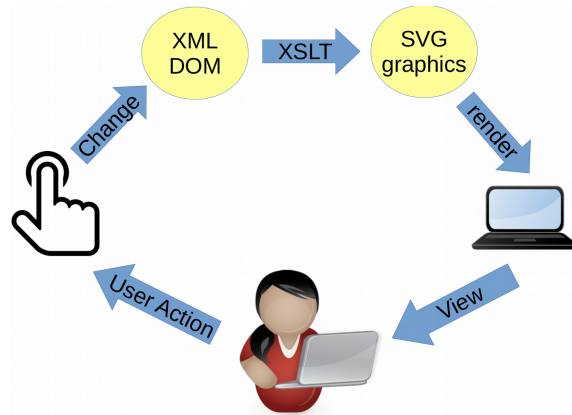


Fig. 21: Editor interaction/feedback loop

document also contain references to callback editor functions, invoked by the browser when the user interacts with the graphical elements.

As the editor employs XSL transformations to display the graphic representation of the XML documents in real-time, the editor performance is highly dependent on the efficiency of the browsers JavaScript interpreter and the respective XSLT processing engine. Recent versions of both Firefox and Chrome employ just-in-time compilation techniques to execute JavaScript code, contributing to increase performance levels. A comparison between both browsers has shown that Chrome offers superior interactive response, being able to display more than 20 frames per second while moving multiple nodes and performing arc rubber-banding, offering a performance level comparable to native applications while editing non-complex models. However, as model complexity increases, the feedback response starts lagging. To solve this problem, the user may select a «fast» edition mode that does not perform arc rubber-banding but offers faster feedback.

Although the main purpose of editor is the design and edition of DS-Pnet models, it also performs other tasks. A very important task is the creation flat models that merge the internal elements of all components, subsequently used to schedule the evaluation sequence of dataflow operations and transition firing, assigning micro-step and nano-step numbers to each node. This task was implemented in the editor to be able to interactively warn the users about cyclic loops in the evaluation sequence. The resulting micro-step and nano-step numbers may also be visualized directly in the model, helping to manage conflicts between transitions.

Group	Function	Description
Selection	Select Nodes	Select nodes with the pointer. Single or rectangular selection. Shift/Control add/remove selection elements.
	Select All	Select all document nodes
	Invert Selection	Select previously unselected nodes
	Undo	Undo last performed operation from undo stack

Undo	Redo	Redo previous undo operation
Copy & Paste	Delete	Delete selection or delete the next picked node if selection is empty
	Copy	Copy selected nodes to clipboard
	Cut	Copy selected nodes to clipboard and delete from document
	Paste	Paste a copy of the clipboard contents to the document
	View Clipboard	Open an auxiliary window to view the clipboard contents. May be used to exchange the clipboard contents with other users, enabling collaborative use.
Duplicate	Duplicate	Duplicate selected nodes
View Mode	Collapse	Switch the selected element viewing mode between graphical or symbolic mode (applicable to arcs and operations)
Geometric transform.	Rotate 90° CW	Rotate selection 90° in the clockwise direction
	Rotate 90°CCW	Rotate selection 90° in the clockwise direction
	Mirror	Mirror selection horizontally
«Smart» tools	Node fusion	Join multiple places or multiple transitions, automatically rearranging the connected arcs
	Complem. place	Create the complementary place of an existing Petri net place, adding complementary arcs
	Semaphore	Create a semaphore place to lock a critical section
Node creation	Place	Add new places
	Transition	Add new transitions
	Petri net Arc	Add Petri net (normal) arcs
	Read Arc	Add dataflow read arcs
	Input Signal	Add input signal
	Output Signal	Add output signal
	Internal Signal	Add internal signal
	Input Event	Add input event
	Output Event	Add output event
	New component	Create a new component on-the-fly (defining number of input/output signals/events)
	Constant	Insert a new operation with a constant value
	New Operation	Insert a new dataflow operation with N input anchors
Component Reuse	Insert library element	Open the library dialog and insert existing components/operations
	Remote Component	Open a connection to a remote embedded server and import components running on that server.
Open & Save	New Model	Start a new model
	Open Model	Open an existing model on the server (or upload a model from a local file)
	Login	Authenticate with a user ID to access a private folder, or manage users/passwords
	Save & Save As	Save model on the server with same name or new name (or download model to local PC)
	Show XML	Exhibit the model as a XML document on a separate window
Verification	IOPT Model Checking	Extract an IOPT model from the Petri net part of the model and invoke que IOPT model-checking subsystem
	Check Syntax	Attribute micro-step & nano-step sequencing numbers to each node and detect cyclic loops
Invoke Other Tools	Split Dist. Nodes	Invoke the node splitting tool, to automatically create sub-models to run on each distributed node
	Code generation	Invoke the automatic code generation tools (C, JavaScript, modular VHDL, monolithic VHDL and XML)
	Simulator	Open the simulator / debugger tool
	Remote Debugger	Open the remote debugger tool

Table 8: Editor toolbox functions

Table 8 presents a list of the functions available on the toolbox. It contains icons to create and select nodes and perform multiple level undo and redo, copy and paste, among many other functions. In addition to the traditional functions usually offered by other Petri net and dataflow editors, the editor offers «smart» functions to automate some error-prone tasks that would otherwise require full user attention, including Petri net node fusion, the creation of complementary places and the creation of semaphores that prevent multiple tokens from entering a critical section.

The concept of semaphore is employed when a system is composed of several concurrent sub-systems that share a common resource. Usually, each of the concurrent sub-systems is modeled by different parts of a Petri net model. However, only one of these sub-systems can simultaneously perform tasks involving the shared resource. A critical section is defined by the set of all places where this resource is being used, that may include places from multiple sub-system nets. After selecting the places that form a critical section, the automatic semaphore function creates a new place and adds input arcs to all transitions entering the critical section and output arcs to all transitions leaving it, preventing more than one sub-system from simultaneously accessing the constrained resource.

Figure 22 displays the expression editor dialog, used to enter the mathematical expressions that define the outputs of dataflow operations. Expressions may be inserted using a keyboard or using menus to select operators, input anchors and literal numeric values. The second option is useful for users of tablets or other mobile computing devices.

An expression can consist on a single mathematical formula, or may be composed of multiple conditional sub-expressions, forming a case construct.

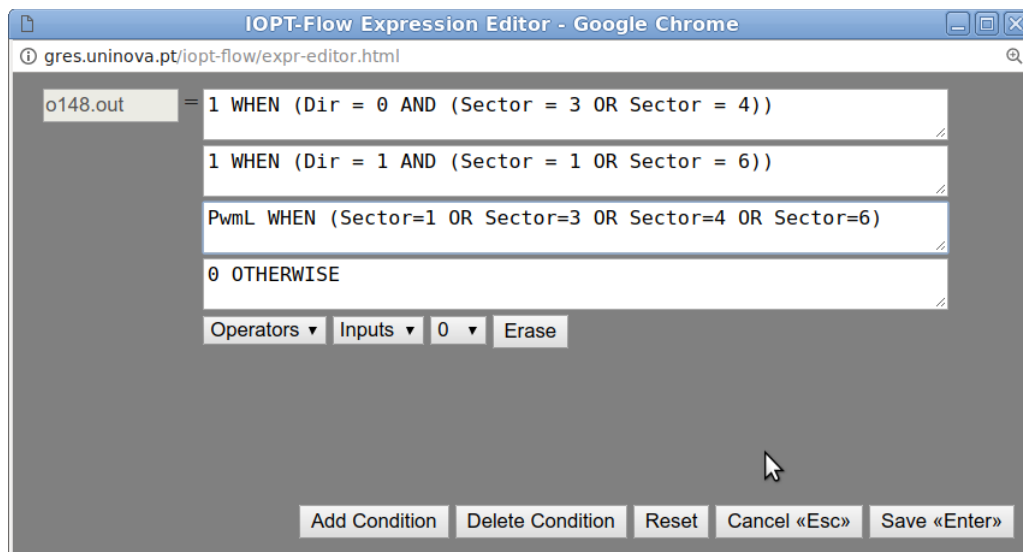


Fig. 22: The Expression Editor

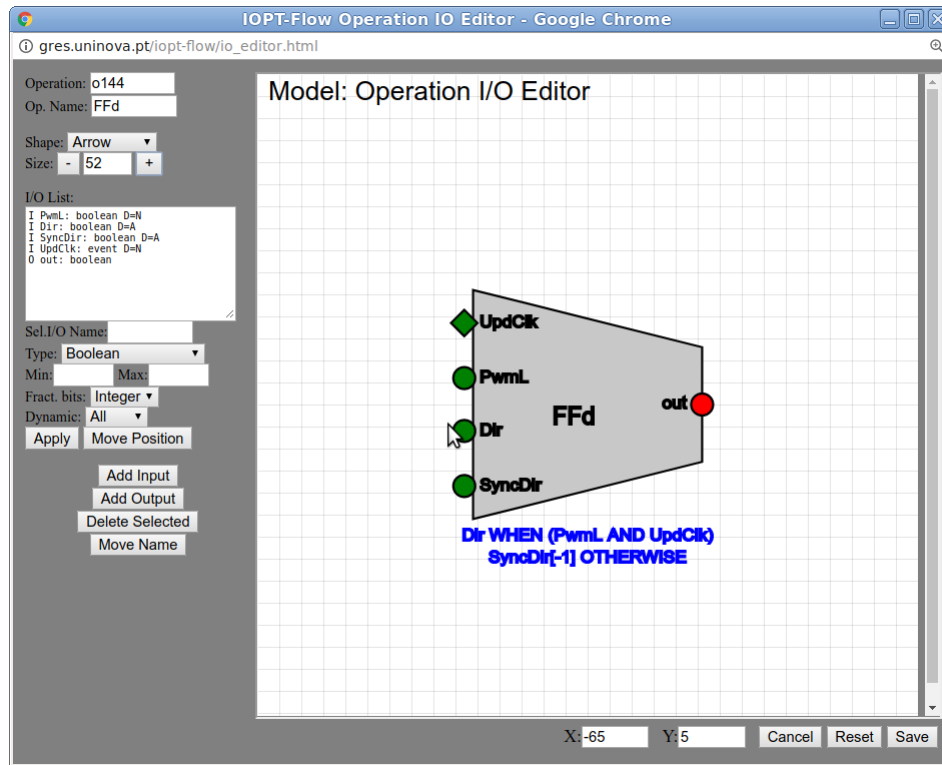


Fig. 23: The operation input/output editor

When an expression is saved, the expression editor is responsible for syntax checking and converting the results to the hierarchical XML format used on DS-Pnet model files.

Figure 23 presents the dataflow operation input and output editor used to define both the graphical and semantic properties of operations, including the graphical size and shape (arrow/trapezoid, rectangular and circular), the number of inputs and output anchors and the respective position. However, the main purpose of this editor is the attribution of names to each input/output anchor and define the respective data-types.

Both names and data-types of input anchors may be attributed in a static or dynamic way. Static names and data-types remain unchanged unless edited again using the input/output editor. In contrast, dynamic names and data-types of input anchors are changed automatically whenever a driver arc is connected to it, inheriting the respective attributes from the arc source.

Dynamic data-types of output anchors are determined using a set of heuristics that take in account the data-types of the inputs and the mathematical expressions. For example, the outputs of expressions containing comparative and logical operators are dynamically assigned the Boolean data type. These heuristics produce the desired results in most cases, but may be manually overridden using the input/output editor.

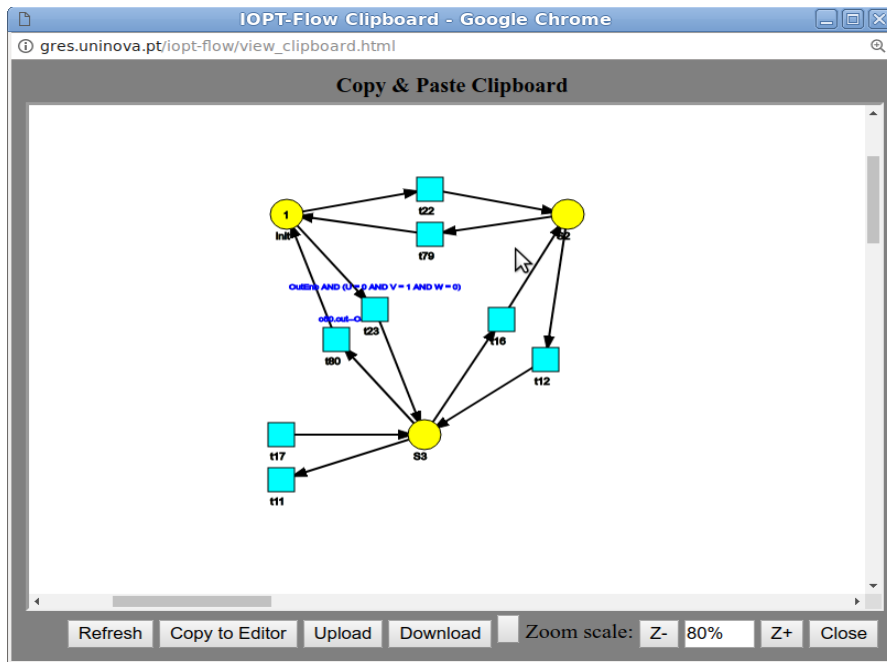


Fig. 24: Clipboard View

Figure 24 shows the clipboard view dialog, used to store the contents of the copy and paste buffer, that may be used to transfer information between different models. Two buttons, «upload» and «download» are used to save a copy of the clipboard buffer on the IOPT-Flow server or download the saved copy back again. This feature was designed to assist the collaborative work between multiple users logged in the same user account: A user may copy parts of a model to the clipboard buffer using the copy&paste functions and upload the clipboard contents to the server. Other users may open the clipboard view window and download the contents saved on the server, observe the downloaded elements and paste it to other models.

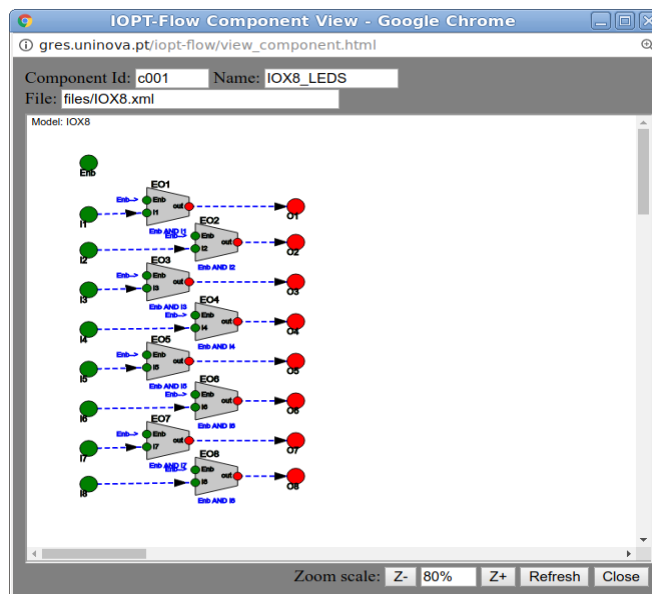


Fig. 25: View component implementation model

Complex models are usually designed using multiple components and the interdependencies between signals often cross the component boundaries. As a result, in order to understand the relationships between these signals, it is often necessary to inspect the implementation models of these components. The IOPF-Flow editor provides two ways of inspecting component implementations: opening a secondary editor window with the corresponding model, or using the component view dialog shown in figure 25.

When a secondary editor window is used to edit component models, the component interface may suffer changes. This way, when a user saves models on the secondary editor window, the secondary window is automatically closed and the main window will scan for interface changes. When the interface suffered the removal (or renaming) of signals or events, any potential connected arcs are erased.

Components may be added to new models in three ways: creating a new component on-the-fly, inserting a component from the library or importing remote components.

On-the-fly components are typically used when a model is designed with a top-down approach. In this case the designer creates multiple components to implement different subsystems, starting with the definition of the component interfaces and developing the component implementation models afterwards. The editor asks for the desired number of component input and output signals and events and creates a new component model that is immediately opened in a secondary editor window.

Finally, the editor has the ability to import remote components from systems that are running the C code generated automatically, as presented in figure 26.

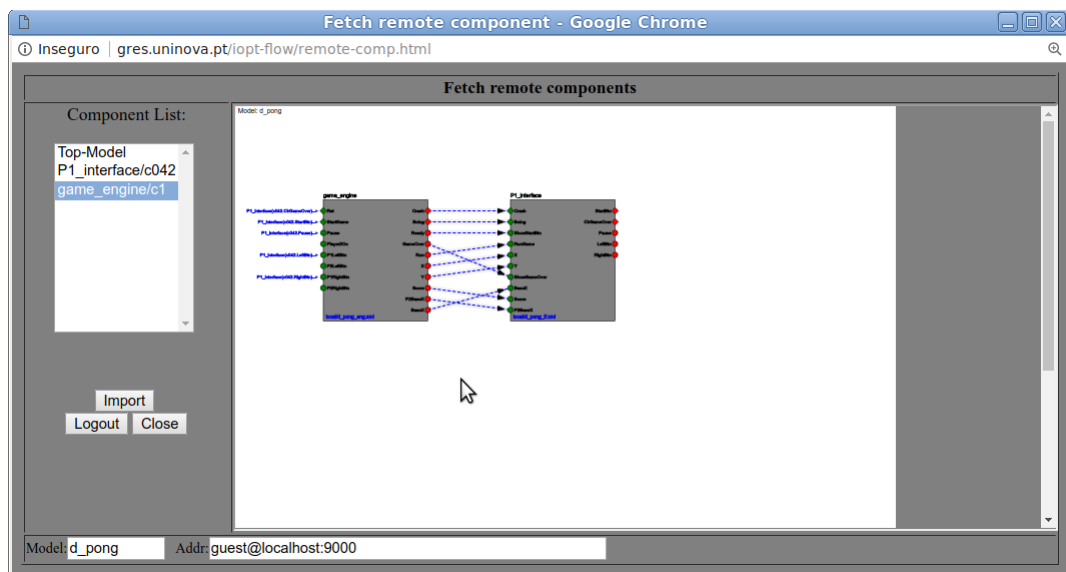


Fig. 26: Import DS-Pnet components from remote embedded nodes

This window displays a model running on a remote node (guest@localhost:9000), and the list of available components on the left side. These components may be selected and imported to the main editor window, in the same way as a library element. However, these components will be marked as remote and preserve the resource location properties, creating distributed systems. Next, the designer just needs to connect arcs to the imported component and the automatic code generator will deal with all communication details.

6.2 The Simulator tool

The IOPT-Flow simulator is used to test and debug DS-Pnet models. It plays an important role in rapid application development: after changing a model in the editor, the changes can be immediately tested using a single mouse click, without any compilation delays to run the code on embedded devices and the risk to damage hardware due to modeling mistakes. In contrast, projects running on reconfigurable hardware usually take many minutes to generate bit-stream files to program FPGAs, resulting in very slow test cycles.

Simulation runs DS-Pnet directly on the Web browser, using the JavaScript code produced by the automatic code generator. As the modern JavaScript engines employ just-in-time compilation techniques, the code generated automatically can reach very fast simulation speeds, only limited by the screen update of the forms and graphics.

Figure 27 displays the simulator window, presenting a toolbox on the left, a form with current values on the right and the model on the center. Input, output and internal values are displayed in real-time on both the graphical model and the form.

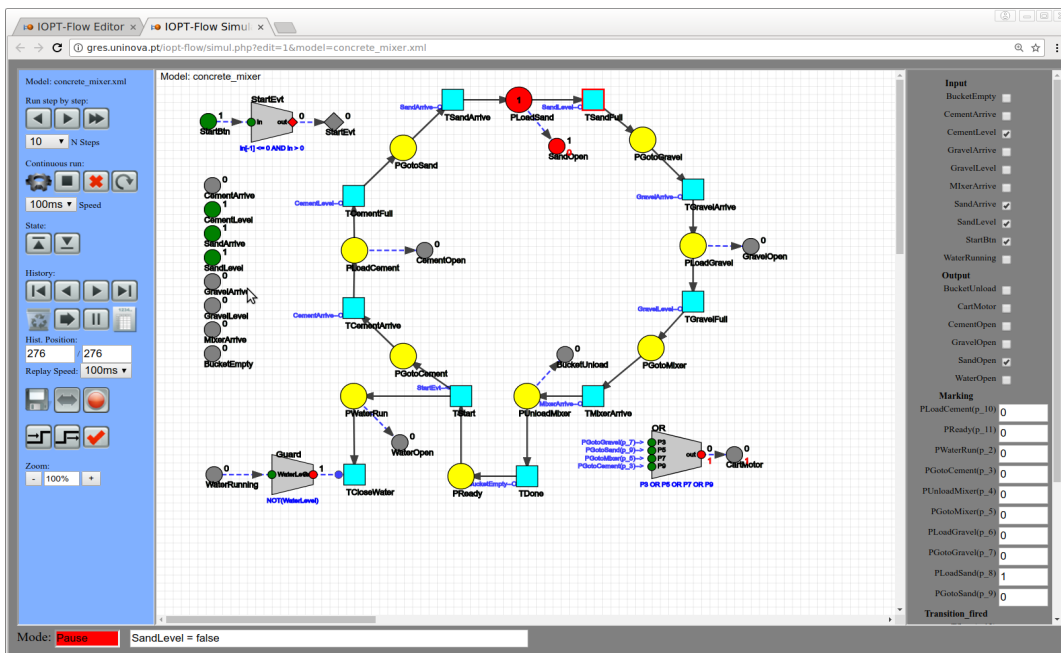


Fig. 27: The IOPT-Flow simulator (Chrome Web browser)

The simulator displays two values next to each output and dataflow operation: the current value and the future value, after the next execution step. In addition, it also highlights the transitions about to fire. This way, when execution is paused the user can change input values and check the consequences before executing the next step.

In order to accelerate debug sessions, the simulator stores information about the execution history, including all input, output and internal values on each step. This way, the user may run simulations at high speed and in case of mistaken input changes, can undo the last execution steps, replay and navigate through the history, and restart simulating from any recorded step.

The user may also define breakpoints associated with transition firing or dataflow operation result changes, stopping the simulation when any of these conditions is reached. This way, simulations can run at high speed, avoiding the need to pause and inspect the results after each execution step.

Simulation history may be viewed as graphical waveforms, as shown in figure 28, or exported to CSV files for further processing using a spreadsheet application. For example, a spreadsheet may be used to define conditions and search for undesired states on long history files, with millions of steps.

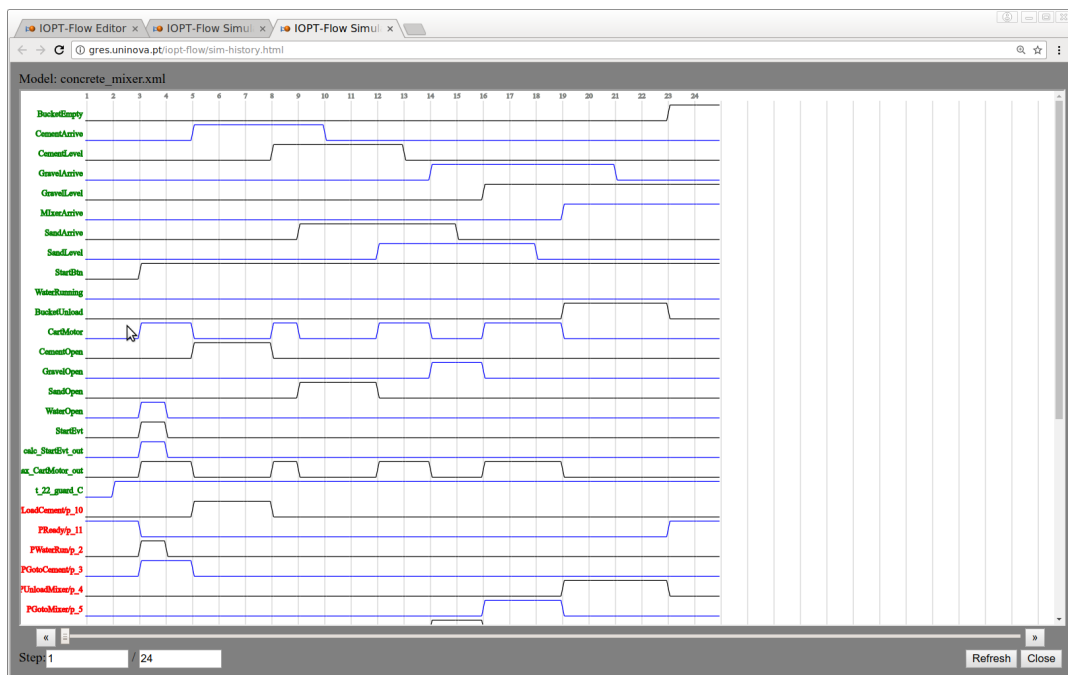


Fig. 28: Waveform view window (simulation history)

The simulator supports the following features:

- Step-by-step execution and continuous run with predefined speed
- Undo step-by-step
- Breakpoints associated with transition firing of value changes in signals and dataflow operations
- Reset to initial state and force new state (force new place marking)
- Simulation history recording, navigation and replay
- Waveform view
- Export/download history to CSV spreadsheet files
- Save simulation history sessions on server
- Replay history sessions, using input values from data saved on server
- Compare history waveforms with previous simulations
- State-space exploration (for closed models)

The last features in the previous list were designed to automate model debugging and further contribute to reduce development cost and time. The possibility to save the history of simulation sessions and later reproduce these sessions with other versions of the same model, with the automatic detection of changes in the resulting waveforms, can be used to automate unit testing and regression tests. This is done by extracting the values of input signals and events from a previous simulation session and replaying it with these values. After replay, the user is informed if the resulting waveforms changed, the first and last step where changes were detected and the graphical waveform window highlights the changed values.

A state-space exploration function is useful to verify closed models, without any input signals or events, or models where inputs are only employed to define constant parameters that do not change during execution. This function was designed to test autonomous systems composed of two sub-models, a controller and a plant, that do not employ input signals except from working parameters or a start command. The state-space exploration continuously runs the model, until it finds a repeated state. As this exploration may reach millions of states, performance plays a critical role and the graphical feedback is disabled, except for a step counter and an interrupt button.

The output of the state-space exploration is stored on the simulation history and may be exported in spreadsheet CSV format for further analyzes. However, several properties may be immediately checked: deadlocks, live-locks and the reachability of

the initial state. If the system repeats the last two steps, a deadlock was found. A live-lock is detected when the system jumps to an unexpected intermediary state, after the initialization phase.

Future work on the simulator include:

- Open parallel windows to display the contents of components (although values appear in the form)
- Run foreign components from the «standard» component library, for example to run graphical user interfaces from the simulator
- Connect models running on the simulator to remote components, located on real hardware devices
- Add a query-system [15] to the waveform dialog to search/filter states that verify certain conditions without having to resort to external spreadsheet applications

6.3 Remote Debugger

Figure 29 shows the IOPT-Flow remote debugger application. With an interface similar to the simulator application, it is used to connect to embedded boards running the “C” code generated automatically, using the JSON/HTTP protocol discussed in chapter 5, to remotely monitor and troubleshoot systems deployed in the field.

The example on figure 29, implementing a distributed game, presents a top level model containing two components, the game engine and the user interface. As the top model does not contain any dataflow or Petri net nodes, they are not visible, but other models may display all types of nodes. In the same way as the simulator, the remote

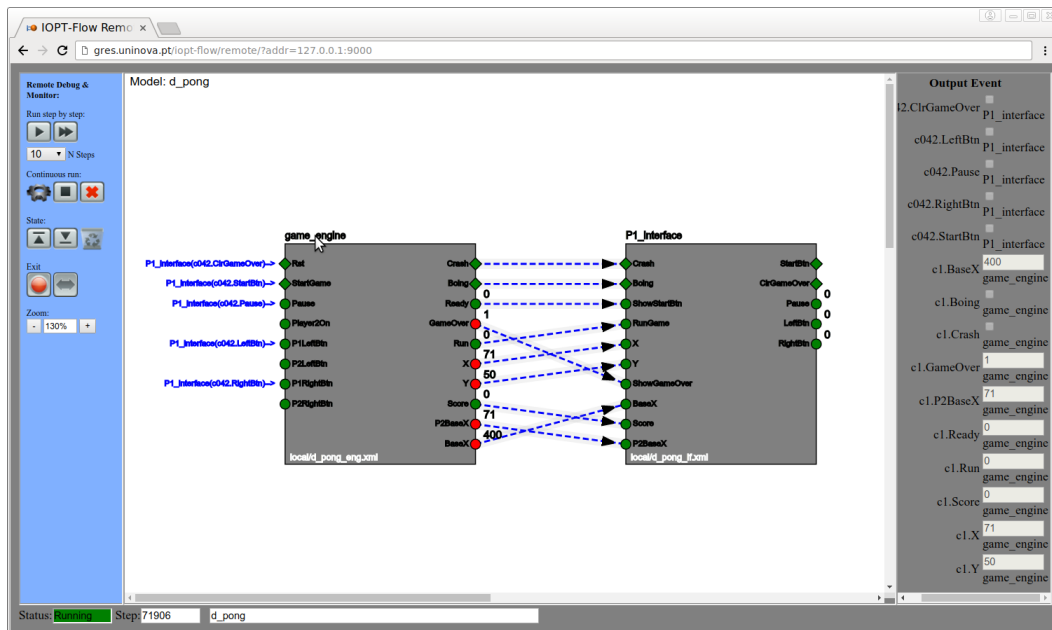


Fig. 29 The IOPT-Flow remote debugger application (Chrome browser)

debugger does not have the capability to enter into components and display the internal elements. However, in the case of the remote debugger, this is a desired feature: developers wishing to hide implementation details can encapsulate the protected models inside components, exposing only the external interface and keeping the interior models private. To ensure privacy, the internal component information is not even exported by the server running on the generated code.

Users may invoke the remote debugger from the IOPT-Flow editor, or can simply point any Web browser to the URL of the embedded devices running the automatically generated code, that redirects the browser to the remote debugger application and sets all parameters. By default the server binds to TCP port 9000, but other ports may be selected in the source code or using environment variables.

After successful login, the remote debugger fetches the required model meta-data and the URL of the original DS-Pnet model, that is presented graphically, offering an user interface similar to the simulator. The user may simultaneously run multiple copies of the remote debugger on different browser window tabs, useful to monitor and debug distributed applications spread across many network nodes.

As the communication between the embedded systems and the remote debugger uses a stream of HTTP server side events, it does not impose any noticeable performance penalty on the remote devices and does not contribute to slow the execution speed. The remote debugger receives a stream of subscribed values, optimized to transmit only changed values, that may suffer from network lag, but does not suffer from information loss. This way, the performance of the remote debugger application is usually limited by the graphical user interface refresh speed and not by the communication bandwidth, even for long distance connections.

The remote debugger offers the capability to pause execution on the remote devices, execute step by step, define breakpoints and restart model execution. Input signals and events may also be forced, overriding the values read from hardware, useful to test the model behavior on unexpected situations or to bypass malfunctioning sensor devices and maintain operation until a replacement sensor is installed. However, in order to use these debug commands, the user must login with a username associated with the «master» privilege in the remote node authentication database.

6.4 Node-Split

The node-split operation is used to split centralized models into several distributed nodes. It is typically used when a system designer starts with a main model that is divided into several local components. The initial model can be tested and debugged using the simulator tool, until it obeys all design requirements. After successful testing, the designer may spread the original model across multiple nodes and create a distributed system.

To perform this task, the designer starts with the creation of a list of virtual node names, according to the physical devices planned for the distributed implementation. After the node list has been decided, the designer should assign each component to the destination nodes, by setting the virtual node name on the respective resource-location attribute.

After assigning all components, the node-split tool is ready to be used. This tool will create a set of new sub-models, to run on each of the virtual nodes, plus an application top model, called the maestro, that runs on another node. Finally, the complete distributed system may be deployed to hardware by applying the automatic code generation tools to each of the sub-models, producing executable applications to run on each node.

The node-splitting procedure has been presented in the distributed execution chapter, but it obeys a basic principle: any signal, event, dataflow operation and Petri net nodes connected to components from a single node, will be implemented on that node sub-model; elements connected to components from more than one node, including arcs, are implemented in the main “maestro” model. The maestro orchestrates the communication between all nodes, according to the arcs that establish inter-node connections. The node-split tool was employed to implement the fourth validation application.

As previously stated, this solution has performance limitations, as an arc starting on «NodeA» and ending in «NodeB» implies two communication messages: the maestro subscribes a source value from «NodeA», that is received using server side events, but whenever it receives a change notification, it has to push the changed values to «NodeB». A more efficient solution could be achieved if «NodeB» directly subscribed the source data from «NodeA», requiring a single message. However, this solution could create increased debugging difficulties, as pausing the main model would not prevent communication between component nodes whose state continued to evolve.

With the current version of the tools, direct communication between component nodes can be implemented by manually creating the sub-models and defining the communication interconnections. However, a future version of the node-split tool might offer options to select centralized or distributed communications.

6.5 Automatic code generation

The automatic code generation tools create code that executes the model semantics, translating the model behavior to several programming or hardware description languages, to deploy on real hardware devices. Figure 30 presents the code generation options provided by the editor:

- C code for micro-controllers, embedded PCs, IoT devices and other computing devices
- Monolithic VHDL for reconfigurable hardware platforms (a single VHDL entity)
- Modular VHDL for reconfigurable hardware platforms (a VHDL entity for each component)
- JavaScript code to run on Web browsers (and on the simulator)
- XML language independent code that may be transformed to different programming languages

The automatic code generation algorithm and available options have been presented on chapter 4, including information about the communication layer added to the generated C code.

Currently the editor contains options to select hardware or software code generation for each component, reserved to support automatic co-design code generation solutions in the future. In the current version, the user must manually call the modular VHDL code generator to obtain hardware descriptions of all components, apply the C code generator on the main model and then manually code the glue logic that connects the hardware and software components, dependent on each hardware platform and operating system employed.



Fig. 30: Code generation options

6.6 Import and export IOPT models

The DS-Pnet formalism inherits many concepts from the parent IOPT Petri net class [29]. The external model interface, composed of input and output signals and events, data types and the Petri net part are common to both formalisms. This way, an IOPT model may be used as a component of a DS-Pnet model, and may be transformed into an equivalent DS-Pnet model. However, as IOPTnet formalism currently does not support components, the opposite is not valid.

In the opposite direction, a DS-Pnet model is not usually convertible into an IOPT net because the mathematical expressions associated with IOPT places and transitions are not able to express the chains of dataflow operations that often appear in DS-Pnet models. In addition, any signal or variable used in IOPT expressions always refers to the results obtained in the previous execution step. This effect is easily translated into DS-Pnet expressions by adding a delay operator suffix «[-1]» to each identifier. On the contrary, IOPTnet expressions do not offer any way to access immediate results from the current execution step. Finally, DS-Pnet models may be constructed without any Petri net element, using just dataflow operations, where the delay operator is used to inherently define state variables, holding values from previous executions steps. Although it would be possible to add a dummy Petri net place to an IOPT model (always marked), just to associate output expressions, it still could not express all types of DS-Pnet constructs.

As a result of this asymmetric relationship, the IOPT-Flow editor has the ability to open IOPT models. When it detects a XML document formatted using the IOPT PNML syntax, it automatically invokes a XSL transformation that converts the IOPT model into an equivalent DS-Pnet.

From the other side, as discussed above, it is not always possible to convert DS-Pnet models into IOPT nets. However, the interface and the Petri net part of a DS-Pnet model is directly convertible on an IOPT net. This way, another XSL transformation was created that extracts all Petri net places and transitions plus input and output signals and events, to form a PNML document. Petri net arcs are automatically translated and read-arcs from places to transitions are converted into test arcs. Figure 31 displays an IOPT Petri net model extracted from a DS-Pnet.

At this point, any dataflow operations are ignored, but future versions might try to extract a subset of these operations to create transition guard conditions and output expressions, but only when these operations may be converted to IOPT mathematical expressions.

As a result, although the conversion from DS-Pnet models to IOPT nets is only partial, it may still be useful, as it permits the application of the state-space calculation and model-checking tools of the IOPT-Tools framework. These tools were developed as a preliminary work and the algorithms and techniques employed for the respective development were described in various papers [7][9][15].

6.7 IOPT Model Checking

The IOPT models extracted from DS-Pnet models retain the Petri net part of the original models, responsible for the system state evolution. This way, it is important to study the properties of the resulting IOPTnet models, as these properties might also apply to the original DS-Pnet model.

One exception to this rule happens when the dataflow operations working as transition guard conditions and as input events, prevent the firing of certain transitions that prevent the reachability of undesired states. However, this problem might be mitigated in future versions with improved guard translation. In this case, the state-space graph of the original DS-Pnet model is a sub-set of the state-space graph produced by the IOPT-tools model-checking sub-system.

From another side, even the state-space graphs built from native IOPT net models often include states that are impossible to reach due to physical constraints imposed by the controlled systems. For example, a model for a water dispenser will never reach a state where the water recipient is full, if the output that opens the valve was never opened. As a result, this kind of problems must be studied using hybrid models of the controller and the controlled system (plant), that will be discussed next.

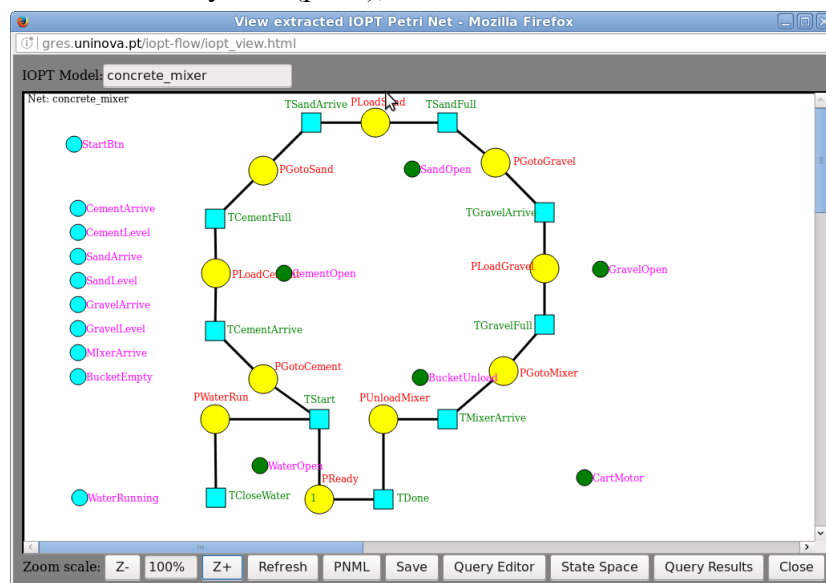


Fig. 31: IOPT Petri net view

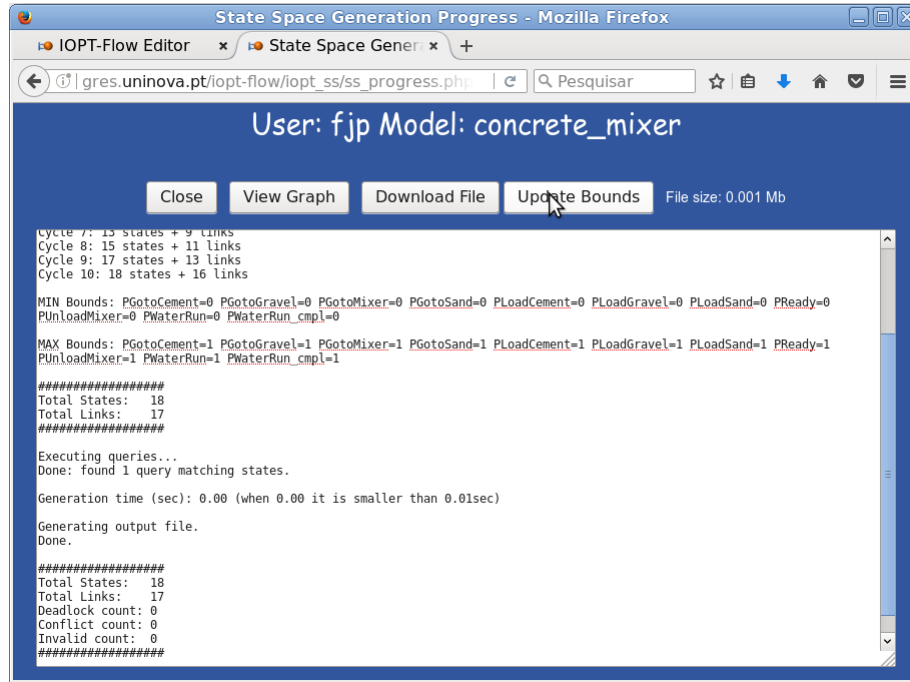


Fig. 32: State space generation progress window

Figure 31 presents the IOPT import window, displaying an IOPT model imported from the DS-Pnet editor. It contains buttons to invoke the IOPT state-space generator and query-system, presented in figures 32 and 33.

As state-space graphs frequently reach millions of states, the IOPT state-space computation algorithm uses “C” code generated automatically, running on the IOPT-Tools server, taking advantage of multi-core processors, using the OpenMP extension of

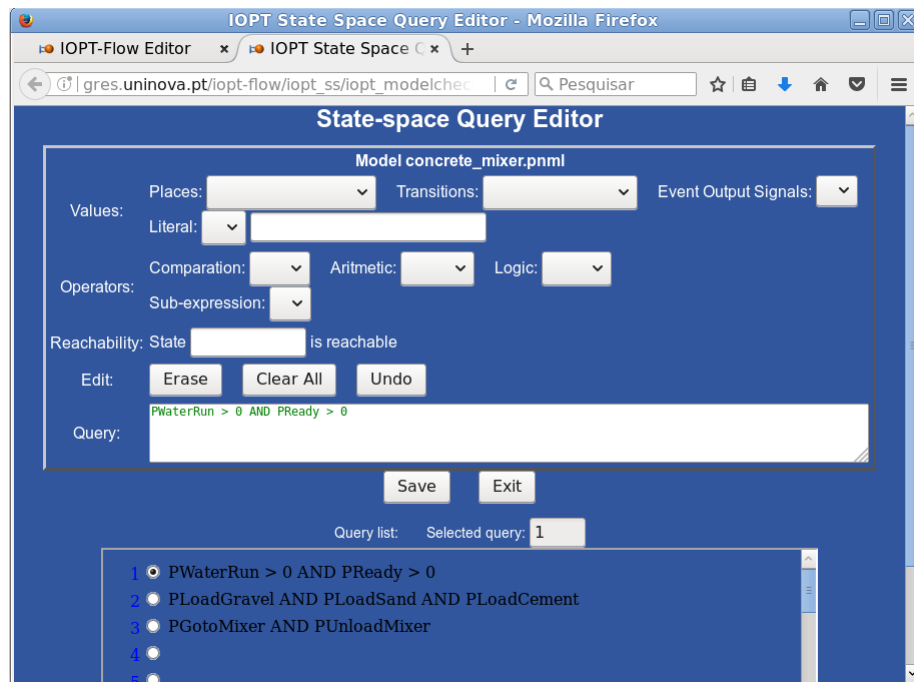


Fig. 33: The query editor (IOPT model checking)

the C language for parallel processing. An example of a very small state-space graph can be viewed in figure 34.

When the state-space graphs exceed hundreds of states, human inspection to detect the reachability of undesired states becomes a time consuming task. To automate this task, it is possible to specify a list of query conditions that are checked during state-space calculation, as presented in figure 33.

Net concrete_mixer

18 (from 18) Nodes, 17 Loops, 0 Deadlocks, 0 Conflicts, Max. Depth = 11, 0 Invalid

Min Bound = [PGotoCement=0 PGotoGravel=0 PGotoMixer=0 PGotoSand=0 PLoadCement=0 PLoadGravel=0 PLoadSand=0 PReady=0 PUnloadMixer=0 PWaterRun=0 PWaterRun_cmpl=0]

Max Bound = [PGotoCement=1 PGotoGravel=1 PGotoMixer=1 PGotoSand=1 PLoadCement=1 PLoadGravel=1 PLoadSand=1 PReady=1 PUnloadMixer=1 PWaterRun=1 PWaterRun_cmpl=1]

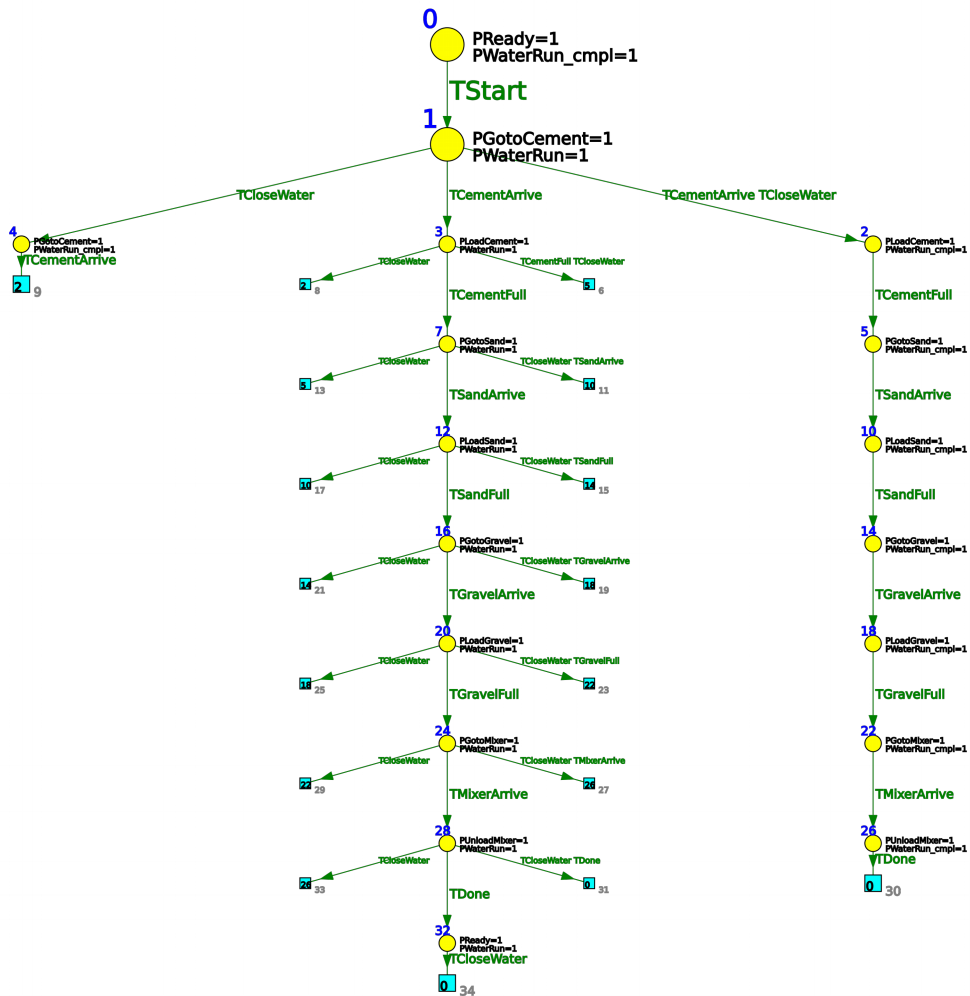


Fig. 34: A state-space graph of an IOPT model extracted from a DS-Pnet

6.8 Component Library

The ability to reuse previously designed and well debugged component models plays a fundamental role in the rapid design of new applications, contributing to reduce the development effort, with gains in terms of development cost and time to market. Applications involving multidisciplinary fields of expertise, as Cyber-physical systems, greatly benefit from the availability of hierarchic libraries containing the most frequently used building blocks of each discipline, alleviating the need for large development teams with experts from every area. For example, if the peripheral devices present on hardware boards are well supported by components that hide the low level details, then it may not be necessary to hire hardware engineers.

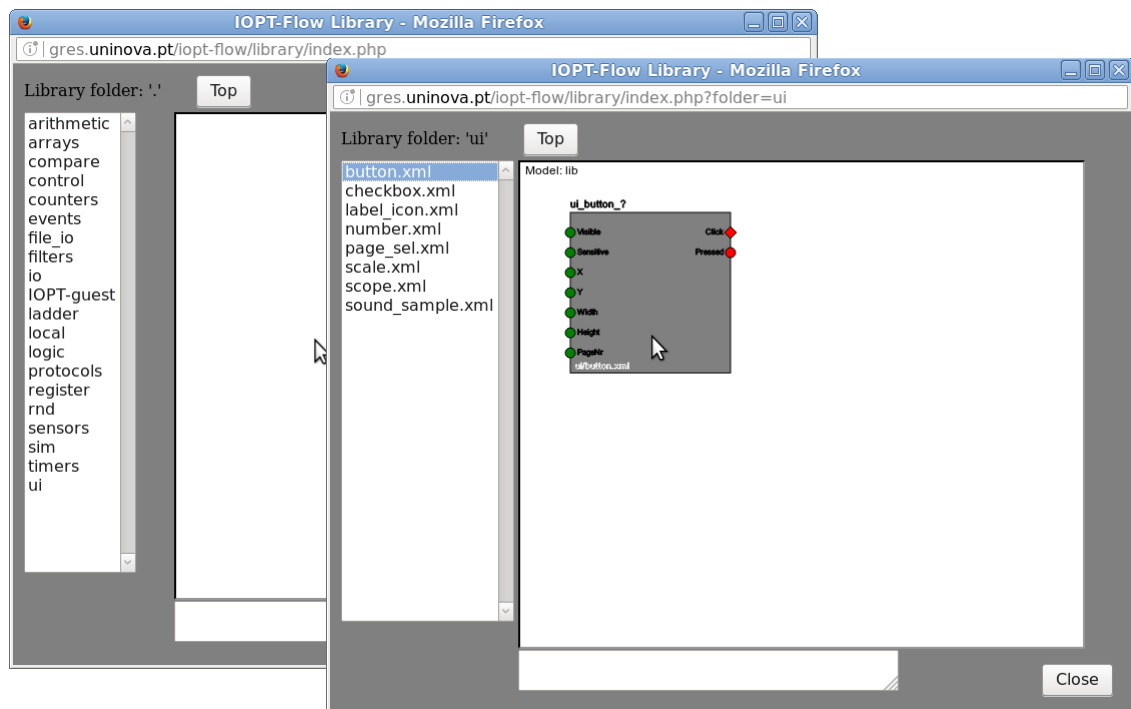


Fig. 35: IOPT-Flow Editor tool - Library dialog (user interface widgets folder)

A large component library, covering many fields of application, is often an important factor in the choice of development formalisms and tools to start a new project. When a library already contains almost all of the required building blocks, then the application design is greatly simplified. In this case, the library coverage may be the deciding factor in the choice of development tools, above the merits of the underlying formalism.

In order to simplify component reuse, the IOPTflow editor provides an hierarchical library, whose interface is displayed on figure 35. The library is divided in folders, according to the field of applications, including arithmetic and logic operators,

ladder diagram blocks and user interface widgets, among others. The current version of the library is in very incipient state, containing almost only the components that were needed to develop the validation examples. However, as soon as other developers start using the IOPTflow tools to develop applications on other fields, the library should naturally grow. In fact, for every model saved in the server, a component interface is automatically added to the «local» library folder, that may be immediately used in other models.

A library element may contain only one of the following items:

- 1 - A dataflow operation
- 2 - A DS-Pnet component
- 3 - A foreign component, designed with external development tools

Library elements containing dataflow operations are used to automate the specification of frequently used mathematical expressions, contributing to avoid mistakes. These expressions may be as simple as constants, basic arithmetic and logic operators, or contain long mathematical formulas. The graphical notation and the respective title texts also contribute to increase model readability. Long mathematical expressions may be specified in two ways: As a dataflow branch containing multiple operation nodes connected through arcs, or just by creating a single operation with the entire expression. Expert users typically prefer the second, but users operating from tablet computers often choose the former, to avoid using a virtual keyboard.

For example, figure 36 presents several components used to emulate ladder logic, as the normally-open contact, normally-closed contact and rung junctions. Although these components were defined using simple AND / OR operations, the graphical notation and anchor placement was designed to allow the creation of horizontal graphs, connecting the contacts in series or parallel, to mimic the Ladder logic diagrams popular in industrial automation.

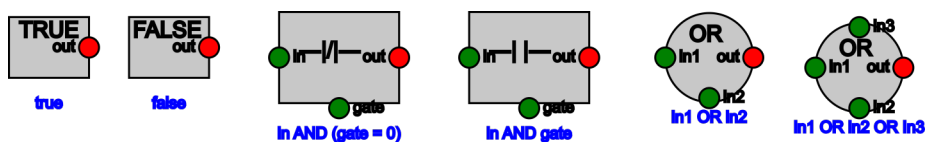


Fig. 36: Ladder-logic library components specified as dataflow operations.

The mathematical expressions used in the dataflow operations refer to the names of the input anchors. These anchors work as internal local variables, that simplify copy&paste operations and the insertion of library elements, as the expressions remain unchanged after inserting an operation clone.

The left side of figure 35 presents the main library menu, that corresponds to a list of folders used to group components according to the respective field. For example, the

«arithmetic», «compare» and «logic» folders contain constants and basic mathematical operations. The «events» folders contain operation that detect simple threshold crossing events and the «sim» folder contains components that detect complex sequences of events [175][177]. The «ladder» folder contains some of the most frequently used Ladder-logic blocks as the normally open and normally closed contacts. Other folders contains counters, clock dividers, timers, registers and flip-flops, etc. The «IOPT-guest» folder contains a long list of components whose interfaces were extracted automatically from the IOPT-tools server (guest account). These models may be directly opened by the IOPT-flow editor, that automatically converts the PNML files to the DS-Pnet format.

The local folder contains a list of component interfaces corresponding to all models stored in the IOPT-flow server. Every time a model is changed or saved, the respective component interface is updated.

Library components may be implemented as native DS-Pnet models (or IOPT models) or using external development tools, with implications to the automatic code generation tools: native components are independent of target programming language, and the tools apply the code generation algorithms to the component implementation models. In contrast, foreign components require external code that must be re-implemented in every language. This way, every component that can be specified as DS-Pnet model, should be designed in that way. Figure 37 presents two native components and their respective implementation models.

However, components that use external resources, as time information, disk file access, database access, use communication ports, interface with hardware devices and graphical user interfaces, must be implemented as foreign components.

An exception to this rule, the tables of variable data were not added to the core language and were implemented using foreign components, as the hardware implementation (Block RAM) differs from the software implementation (arrays). As a conclusion, native DS-Pnet models are independent of the target hardware and programming languages. In contrast, foreign components require the writing of glue logic code to connect the output of automatic code generators to the foreign component functionally, that must be re-implemented on each language.

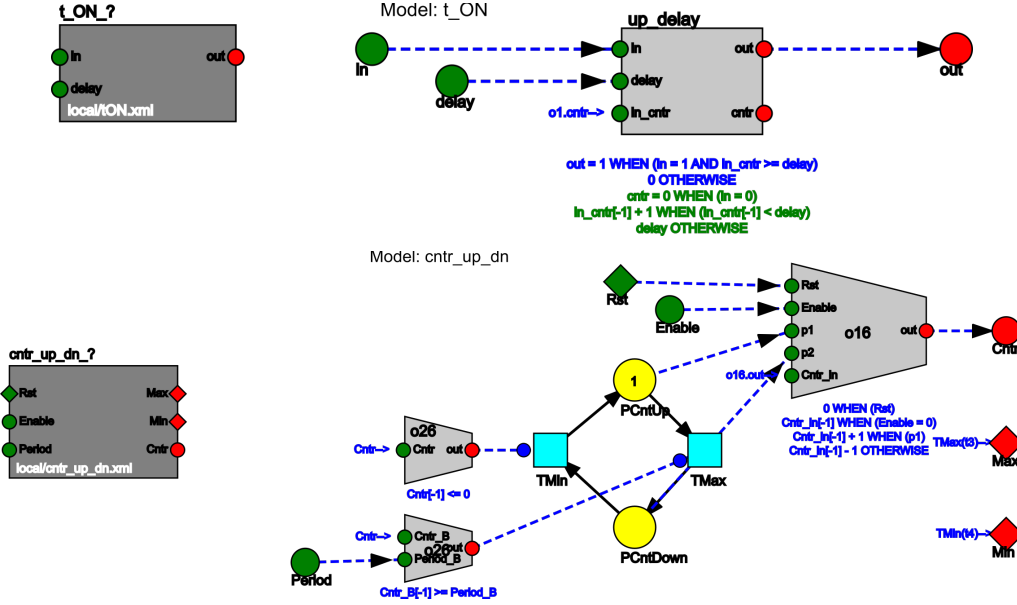


Fig. 37: The t_ON timer and Up/Down counter native components. Component interfaces (left) and implementation models (right).

6.9 Standard foreign component library

Foreign components add enhanced functionality to DS-Pnet models, providing an interface to access any capabilities offered by the operating system and hardware. However, the implementation of these components is not usually portable across different hardware platforms, operating systems or programming languages. As a consequence, in order to obtain portability, it is necessary to define a «standard» library containing a minimal set of foreign components that must be supported by all automatic code generators. Any application models using just components from the standard library (and native DS-Pnet components) will be portable across all supported languages and hardware platforms. In addition, the implementation of these components should also aspire portability goals, employing broadly disseminated APIs and libraries, to simplify the porting for different platforms.

The current version of the «standard library» contains only a reduced set of components, and is only supported by the C code generator, but a future JavaScript implementation is planned to allow the simulation of models containing «standard» components and also to permit the design of Web based remote user interface applications. Those components may even be implemented as reconfigurable hardware in association with the VHDL code generator. For example, arrays may be implemented in hardware as RAM blocks and the graphical user interface widgets may inherit work previously developed [10][11].

At moment, the set of “standard” foreign components is restricted to the following list:

- Variable arrays (vector and matrix)
- Data file input/output and data log
- System time information
- Random number generator
- User interface widgets
- Audio sample player
- Industrial ModBUS Gateway

6.9.1 Arrays

All development formalisms aiming to solve non-trivial problems provide some type of data structures to handle large quantities of data, including arrays, linked lists or structured databases, among others. In the case of DS-Pnets, dataflow operations can use single dimensional vectors or bidimensional matrices of constant values, used to store tables of data and implement general functions of one or two integer arguments. However, arrays of variable data were excluded from the core formalism in order to uniformly cover both software and reconfigurable hardware platforms.

Arrays may be implemented in hardware using the block RAM modules included by most FPGA devices. However, access to the data stored in block-RAM is restricted to a single element at a time (or double, for dual port block RAM devices), while the arrays offered by software programming languages can be accessed multiple times on a single mathematical expression, something that would be difficult to replicate on hardware platforms that execute one step per clock cycle.

Using components, the external interface of the arrays automatically expose the same limitations as block RAM devices, restricting access to a single array element per execution step that can be supported by both software and hardware implementations. As a consequence, possible concurrent accesses to array data must be explicitly dealt by the model designers, that must create the appropriate state-machines. As concurrent array access is a typical use-case, other components may be developed to automate this design pattern.

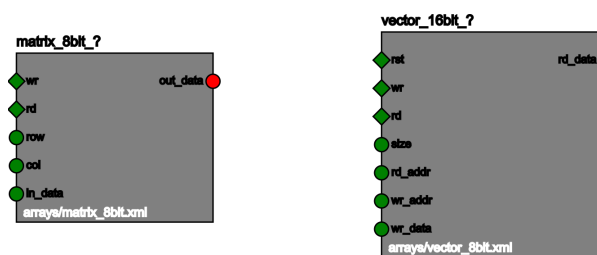


Fig. 38: Foreign array components: vector and matrix

Figure 38 presents the two types of available array components: single dimension vectors and two-dimensional matrices. There are two versions of each component, storing 8 or 16 bit values. The matrix component has fixed dimensions, limited to a maximum of 256x256 elements, but the vector may be resized at run-time by changing the value of the size input.

The external interface of these components is similar to real-world memory devices, that employ address buses, data buses and control signals to perform read and write operations. In this case, the buses correspond to input signals and the control signals are the «wr» and «rd» events. The last «rd» value is copied to «out_data».

6.9.2 Data file input and output

The file input and output components bring the ability to store information in a permanent form and retrieve data from mass storage devices. As the core formalism does not presently support textual data-types, the information stored and retrieved using these components is restricted to numeric values, that are stored in a spreadsheet-friendly CSV (comma separated values) format.

Figure 39 displays the interface of the «data_source» and «file_log» components, respectively used to read and write CSV files. The path/filename of the files is specified using the component resource-location property. In both cases the component interface has four data channels (A-D), meaning that it can read or write up to four simultaneous values, stored in the CSV files as different columns.

The «data_source» component is controlled by two events, «RstFile» and «ReadData» that respectively rewind the reading position to the begin of the file and read a single line, publishing the read values to the «DataA-C» output signals. Two additional signals, «OpenOK» and «EOF» notify successful file opening and reaching the end of a file. In an equivalent way, the «file_log» component uses a «RstFile» event to erase the file content and a «WriteData» event to add a new line to the CSV file with the current «DataA-C» values.

Possible uses for the file I/O components include the data logging to store relevant information about execution history, run simulations from previously stored input data,



Fig. 39: The file input / output foreign components

harvest sensed data for future data processing and graphical presentation, but different applications may find different uses for data storage. By connecting the data-channels to other component inputs and outputs, it is possible to automate debug and simulation from vectors of test data previously stored in CSV files. Data files may be edited using spreadsheet applications, and processed to create reports and graphics.

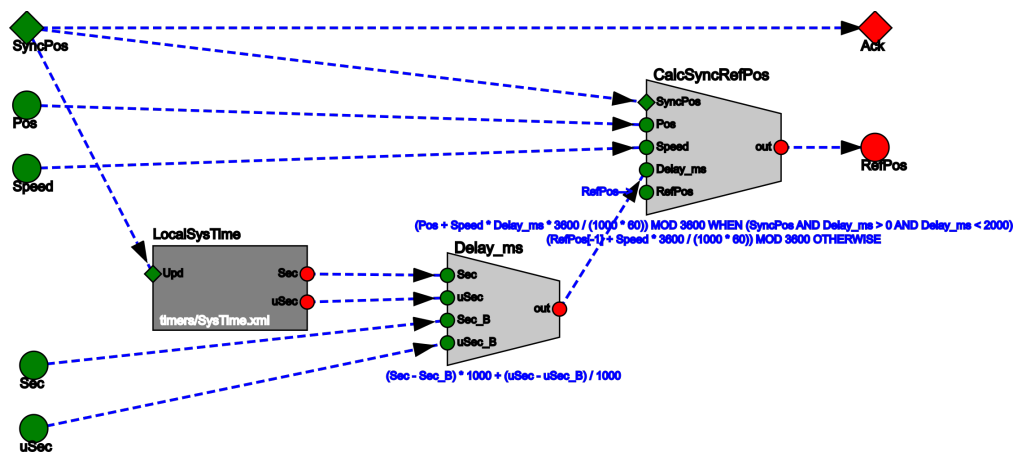
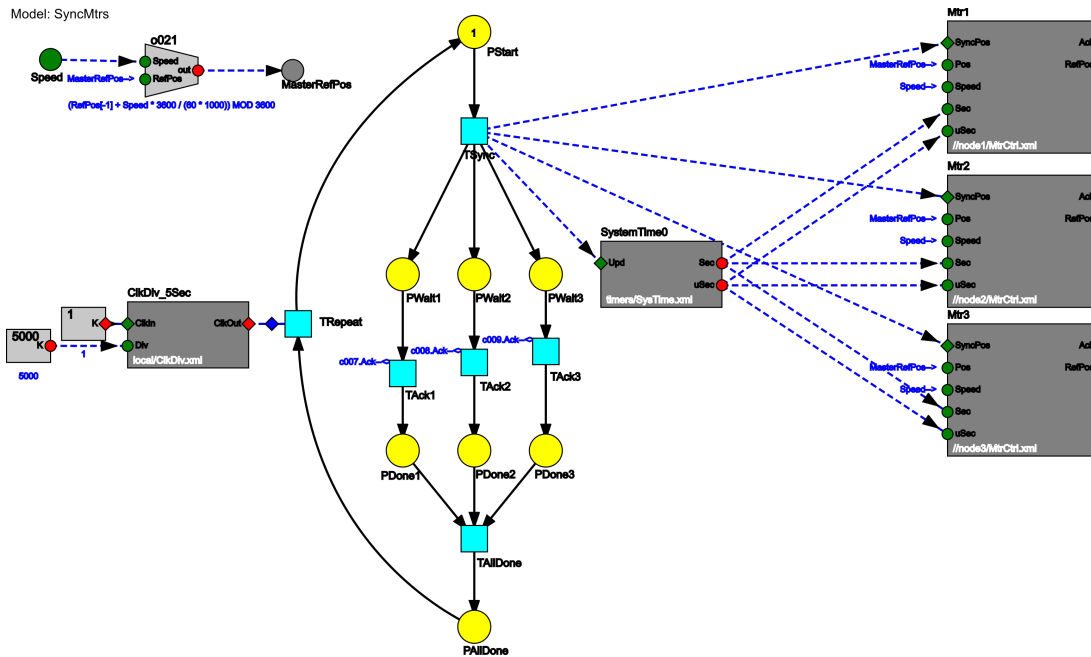
Finally, the interface of the file I/O components may also be used to implement other foreign component classes, used to access data from other back-ends that can provide streams of simultaneous data channels, such as database tables, network connections or serial lines. Although the components were designed with only 4 channels, multiple files can be simultaneously opened, and this number may be increased in the future.

6.9.3 System time information

The system time component is used to obtain the current time from the operating system clock. It has an input event and produces two output signal values: the number of seconds since 1970-Jan-1 in universal system time and a number of microseconds. When the component receives an «Upd» event it updates the output signals with the current time stamp, that may be used by other components in the same execution step.

This component is important to synchronize the actions of sub-systems running on different distributed nodes. As the network latency of Internet communications is unpredictable, components may pass time stamps associated with request/answer events, interpreted by the receiving components to take the appropriate actions to compensate for the transmission delays. In order to achieve this, all clocks in a distributed system must be synchronized using an external protocol, as NTP [178].

Figure 40 presents an example model that uses time-stamps to synchronize the position of three motors running on distributed controllers. The event triggered by transition «TSync» is used by component «SystemTime0» to generate a time-stamp (sec,usec), that is forwarded to each motor controller together with the instantaneous reference-position and speed. Using this information, each of the remote components «Mtr1», «Mtr2» and «Mtr3», can calculate the time elapsed since the event was triggered until it was received and use the speed to compensate for the delay, as presented in figure 41.



6.9.4 Random number generator

The random generator component provides a very simple interface to generate sequences of random numbers. When execution starts, the «Rnd» output is assigned with an initial random number, that is refreshed with a new number whenever the «Gen» input event is triggered. The component implementation automatically seeds the random number generator sequence according to the local time and process identifier.

In addition to the traditional applications of random numbers, for instance in games, random numbers may also be used to automate the testing of other components, by feeding sequences of random numbers to the respective inputs and storing the results, or modeling additional logic to detect the reaching of undesired states.

6.9.5 Graphical user interface

Graphical user interfaces are used on almost every application, to operate and monitor the controlled systems. The user interface library folder contains a list of components that was selected to allow the creation of both physical interfaces running on embedded hardware and remote Web user interfaces, although only the first is currently implemented.

The “C” version of user interface sub-system was implemented using the GTK library, available on the Linux, Windows and MacOS operating systems, being used by default on many embedded Linux distributions. For example, the user interface of the Raspbian operating system for the Raspberry PI family of devices is based on GTK. The Web version will be implemented using HTML and JavaScript. Previous work on hardware based user interfaces [10][11] may also be used to create a VHDL version of this library.

An user interface built using this component library is divided into “pages”. A page is the equivalent of a computer window or screen, that contains a set of graphical objects, called “widgets”. An application model may have up to 16 pages. At any time only a single page is visible. A special component, called the page selector, is used to select a visible page, permitting the creation of user interfaces where the user navigates through the pages.

Widgets are objects that appear inside the pages to display information and receive user input. It is possible to display text messages, icons and background images. User input is received using buttons, check-boxes, scroll-bar scales and numeric inputs. A special «scope» widget is used to present graphical waveforms of two input signals in real-time, like an oscilloscope. Figure 42 presents the external interface of the widget components.

Each widget has inputs to define a page, the XY coordinates where it appears, and corresponding size (width x height). An application may decide when each widget is visible or hidden, sensitive or disabled and dynamically move a widget by changing the coordinates. For increased usability, the width and height inputs may be left unconnected and the system will automatically calculate the correct size according to the used text fonts and icon sizes.

A widget may also be dynamically moved between pages. For example, if the «PageNr» input of a component is driven by the selected page, then the widget will be present on all pages. An application can only have a single instance of the page selector component and the initialization code will abort if more than one copy is found.

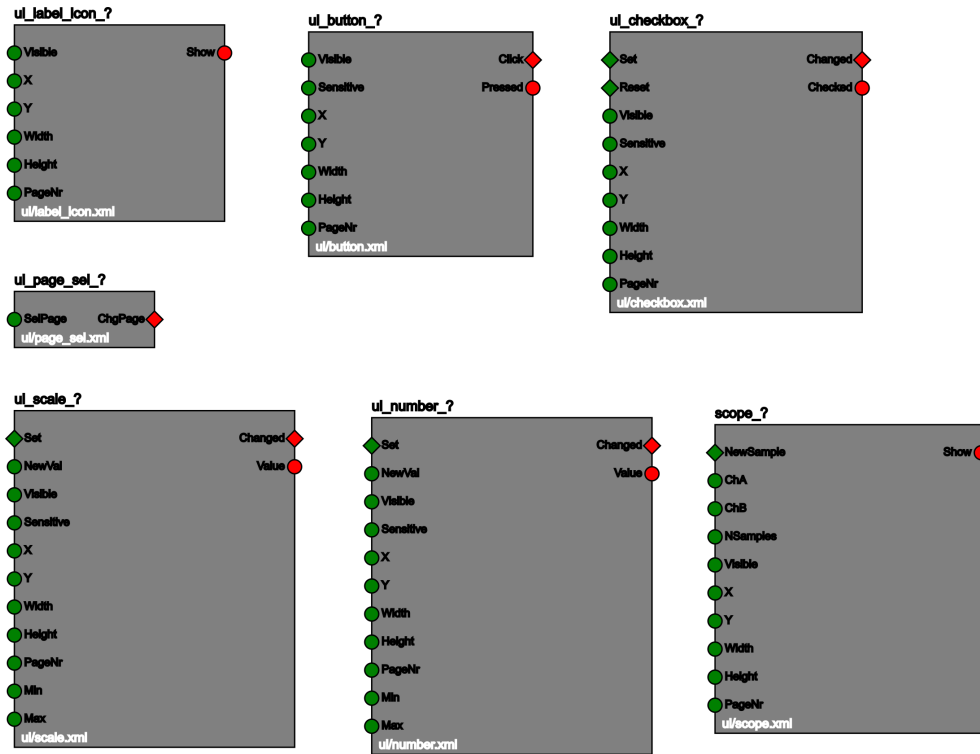


Fig. 42: Graphical user interface components

The «label_icon» is used to display text messages or graphical icons, that may be hidden or visible according to the application logic. Components implementing sensitive widgets, that receive user feedback, have output signals to hold the values read from the user and output events to notify button clicks and changed values. In the opposite direction, components that hold state, as check boxes, number inputs and scales, have input signals and events to let the application force new values.

The output signals and events of the widget components may be used by the applications as if they were produced by physical buttons or sensors. These values may be used to perform dataflow calculations, control the firing of Petri net transitions or be forwarded to remote components. In a cyber-physical system, a user interface node may read data from multiple remote components and present it graphically, just by connecting arcs. In the same way, the output of user interface buttons, scales and numeric inputs may be connected to the inputs of remote components, providing remote control capabilities.

Figure 43 presents a screenshot of the test application, used during development to test the user interface widgets. The scope widget at the bottom presents two waveforms. This component employs a «NewSample» input event to memorize a fresh sample and scroll the image horizontally. Application logic may control the «NewSample» to emulate the trigger level functionality of real oscilloscopes and start capturing consecutive samples.

This screenshot also displays icons and text messages. Again, as DS-Pnets do not have textual data-types, these messages were defined using the «comment» parameter of the respective widget components and the path of icon files was defined using the resource location parameter. Button keyboard accelerators may be defined using a «param_string» component parameters.

Using component properties to specify text messages help separate the user interface text from the application logic. User interface and application logic can be further separated by encapsulating both parts into different components (application and user-interface), connected through arcs. Next, the local user interface can be transformed into a remote user interface just by changing the resource location parameter of the application component are generating the C code again.

The user interface toolkits of the modern operating systems offer many widgets and hundreds of configuration options. In contrast, the set of proposed user interface components is very small, but was selected to support a wide range of applications, covering many industrial automation use-cases. Although this set may grow in future versions, it was considered appropriate for proof-of-concept applications.

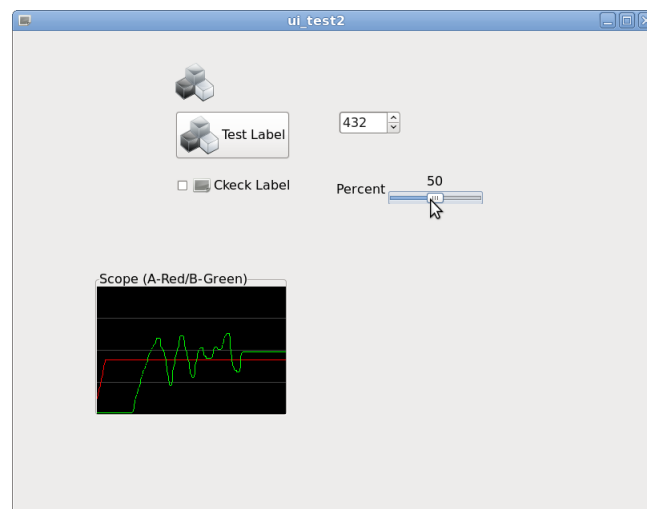


Fig. 43: User interface test application

6.9.6 Audio samples

An additional user interface component is used to play audio samples. It loads a Wave file containing a sound sample and plays it whenever a «Play» input event is fired. An output signal notifies if the sample is currently playing or has terminated.

Audio samples may be used during error and warning situations that require immediate user attention, but may also be used to create rich multimedia applications combining animations and sound, as games and other entertainment systems.

6.9.7 Industrial ModBUS Gateway

A final component was added to the standard library, implementing an interface to communicate with industrial devices using the ModBUS field-bus protocol [67]. This protocol was chosen due to the implementation simplicity and level of dissemination. Supported by virtually every automation manufacturer, it permits the communication between DS-Pnet models and programmable logic controllers, CNC controllers and servo motor controller drives, among others. This component contributes to allow the immediate application of DS-Pnet models in real-world situations, by providing a bridge between the new IOPT-flow cyber-physical systems and legacy industrial applications.

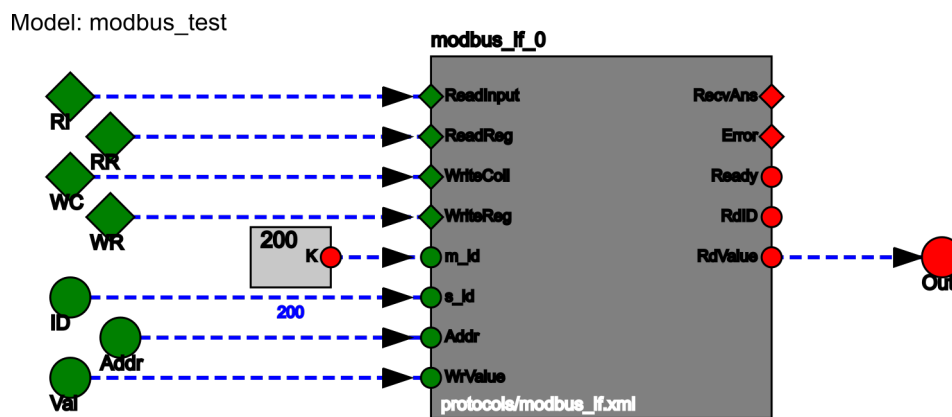


Fig. 44: ModBUS component test model

The ModBUS protocol is usually deployed over a RS485 bus, with a single master controller and up to 252 slave devices, all using the same serial line. As the line length is frequently long, using relatively low speeds (ex: 9600, 19200 up to 38400), the communication latency usually reaches many hundreds of milliseconds. As a result, many control applications cannot send commands and wait for the answers in an idle state. To avoid this problem, the «modbus_if» component was implemented in a non-blocking asynchronous way, to permit sending ModBUS commands and continue execution without blocking.

A test application used to debug the ModBUS component can be viewed on figure 44. It presents the component external interface and a set of input events and signals, used to force values using the IOPT-flow remote debugger tool, including commands, slave device identifier, addresses and values to set on the slave.

The component interface uses four input events to trigger 4 ModBUS commands: ReadInput, ReadRegister, WriteCoild and WriteRegister. When one of these events is triggered, the corresponding ModBUS request is immediately transmitted through the

serial line. Later, when an answer or an error is received, the component assigns the results to the «RdValue» output and triggers an output «RecvAns» or «Error» event.

To test the ModBUS interface component, it was connected to a DELTA programmable logic controller. The test application was able to read the PLC inputs, command the PLC relay outputs and read and write to the PLC internal memory. In another application, presented in figure 45, it was used to control an industrial variable speed drive, to start, stop, set the velocity and read electric current values of an induction motor.



Fig. 45: ModBUS + UI motor control application running on a Raspberry-Pi 2 card, with an LCD+touchscreen hat and an USB-RS485 serial converter

6.10 Debug And Model-Checking

The debug and model-checking tools play an important role in the rapid development of embedded applications, permitting the discovery of modeling mistakes during the early design stages, resulting in faster development time that contributes to reduce costs. In addition, as cyber-physical systems involve both computing and physical devices, error detection before reaching the prototype implementation phase avoids potential catastrophic hardware malfunctions, that could cause permanent damages to physical devices.

From another side, both consumer devices and industrial systems are subject to an ever increasing set of regulations and certification requisites that must be verified before a product reaches the market. This way, in addition to the general properties that must be checked, as the reachability of potential deadlocks and live-locks, the model-checking tools may also be used to automate the detection of undesired states that could pose safety violations or break certification rules.

In the preliminary work, before creating the DS-Pnet formalism, a model-checking sub-system for parent IOPT Petri net class was created, including an IOPT state-space generation algorithm [6][7] and a query-system [9], with the goal to satisfy the problems presented above. This sub-system may be used from the IOPT-Flow framework, to apply these tools to the analysis of the Petri net part of a DS-Pnet system, responsible for the state-machines that implement the reactive part of the controllers.

Unfortunately, this solution loses information about the dataflow part of DS-Pnet models, that impose restrictions to the evolution of the reactive part of the models, resulting in much larger state-space graphs. For example, the guard conditions and input events that inhibit the firing of transitions are defined using dataflow operations, that are lost in the Petri net extraction.

As a result, the state-space graphs built without considering the effects of the dataflow part of the models contains entire branches that would never be reached on a real system. This way, the resulting state-space graph does not correctly represent the behavior of the DS-Pnet system. However, it contains the state-space of the DS-Pnet system and thus, if the undesired states are never reached on the resulting graph, then the real system will also never reach these states.

A native model-checking system for DS-Pnet systems has been planned but has not yet been implemented. The dataflow part of the models usually employs integer input signals with large ranges (analog signals, etc.), and their implications on the

dataflow results would produce huge state-space graphs that consume enormous computational resources and calculation time, not practical for real world applications.

Several strategies to mitigate this problem were considered:

- 1) Hardware accelerated state-space calculation, taking advantage of the VHDL code generator, but performance would still be limited by memory bandwidth limitations, to store a reached-state database and detect repeated states
- 2) GPU accelerated state-space calculation, taking advantage of the massively parallel architectures to execute thousands of simultaneous threads
- 3) Apply state-space reduction techniques, using approaches presented in the literature [37][38][46][78]
- 4) Employ other model-checking techniques not based on state-space graphs, as symbolic model-checking [179]

However, even when the state-space graph of a DS-Pnet controller model is accurately generated, it may still result in huge graphs containing millions of states that are never reached in real-life implementations, as it does not account with the interactions between the controller and the controlled system (plant).

Recalling a previous example, the level of a water tank will not raise if the controller output that enables a water pump that fills the tank is disabled. However, this information is not available when we study just the controller model, and the resulting state-space graph may contain ramifications containing states where the tank reaches the maximum level but the water was not flowing. In the same way, studying the plant model alone will also generally result in very large state-space graphs, including many states that are not reachable under normal operating conditions.

In contradiction, the state-space of a complete system, combining both the controller model and the plant, will usually result in much smaller graphs, that consume considerable less computing resources to calculate and are obtained faster, although the combined model is larger than each of the parts.

The reason behind this reduction is related to the main function of a controller: by definition a controller imposes a set of rules to the plant that does not let it escape from a set of valid desirable states. In addition, the plant obeys a set of physical world restrictions that limit the set of output values produced at any moment. Using the previous example, once water starts pumping in an empty tank, the water does not instantly reach the maximum level, but will slowly grow with time. As plant values are

sensed by the controller models, these physical restrictions will contribute to reduce the size of the combined state-space graph.

Combining Petri nets and dataflows, DS-Pnets are well adapted to model both controllers and the controlled systems (plants). From one side, the controllers are usually reactive systems, that respond to changes in sensed values and events coming from the plant. From another side, plants usually employ mechanical devices and other dynamic physical systems that are better modeled using dataflows. Finally, model composition based on components lets the designer model both the controller and the plant as separate components, that may be simulated independently to perform a first stage of debugging and testing, ensuring that both models behave correctly under typical use-cases.

The complete models, combining the controller and plant components, form an autonomous system, that may be simulated without requiring user input, except eventual «start» commands to initiate the system operation. The top level model exposes only the two components and the arcs that establish the communication between controller and plant.

This solution implies the creation of additional models to simulate the plant, that may be subject to design mistakes and simplifications to reduce modeling complexity. This way, the plant simulation models require individual testing to verify if the behavior under typical use-cases corresponds to the expectations. However, the plant models offer an additional advantage, as they permit debugging the controller models using only software tools, not suffering from possible hazardous situations that happen when physical and hardware components are in the loop, including damaging physical components and causing personal injuries due to controller design mistakes.

After testing terminates, the component used to simulate the plant may be replaced with another component, sharing the same plant external interface, that communicates with the physical plant devices, reading sensed values and driving actuators, maintaining the same communication arcs.

The simulator tool offers a function to explore the state-space of the autonomous controller-plant systems, that continuously calculates and records the system execution, until it finds a loop to a previously reached state. To improve performance, this function works in background mode and does not show any graphical feedback. The resulting history waveforms may be searched to detect undesired states and confirm the existence of desired end-states. In addition, the data may be exported to a spreadsheet application, where information may be processed using automatic filters.

6.10.1 Application example

Figure 46 presents a diagram of a concrete mixer plant, extracted from an example previously published on [27], where a detailed explanation of the models can be found. The plant mixes four ingredients, cement, sand, gravel and water to produce concrete. A cart moves over the rail in the clockwise direction, stopping at four positions, to load cement, sand, gravel and finally to unload the bucket contents into the mixer. In parallel a pipe dumps water directly to the mixer.

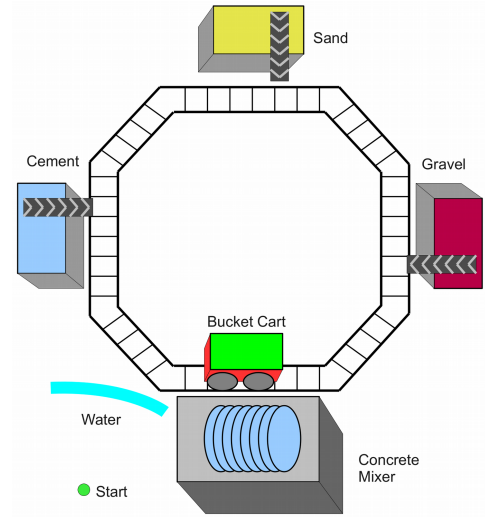


Fig. 46: Concrete mixer plant

The plant is equipped with a set of sensors to check if the cart has arrived at each stop position, to measure the volume of the materials loaded into the cart bucket, to read the water level and check if the bucket has been completely unloaded to the mixer. In the opposite direction, it offers actuators to enable the cart motor, control the conveyor belts that load materials on the cart bucket, unload the bucket and control a water valve.

The controller model, shown in figure 47, starts with place «PReady» marked, assuming that the cart is parked near the concrete mixer with the bucket unloaded. The controller is waiting for a «StartBtn» button press, that produces a «StartEvt» event required to fire the transition «Tstart», before start executing a concrete mixing cycle.

After «TStart» fires, place «PGotoCement» will be marked, enabling the cart motor according to the dataflow operation at the bottom right corner. Next, the transition «TCementArrive» will be waiting for the sensor input «CementArrive». Upon arriving, place «LoadCement» will be marked, enabling the «CementOpen» output that controls a conveyor belt used to dump cement on the cart bucket. When the cement reaches the desired level, transition «TCementFull» will fire and the cart starts moving to the next stop position. This pattern repeats for the sand and gravel loading.

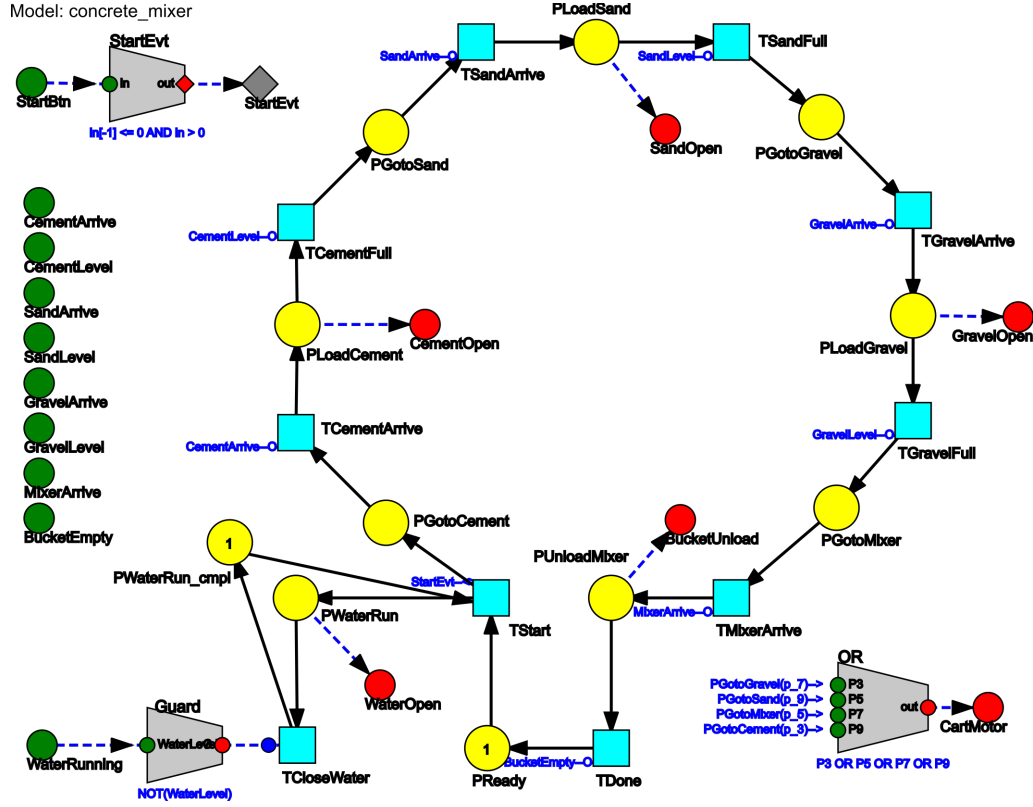


Fig. 47: Cement mixer controller model

When the concrete mixing cycle starts and transition «Tstart» fires, the place «PWaterRun» is also marked and opens a valve to start dumping water on the mixer. This operation runs in parallel with the cart travel, and stops when the water reaches the desired level, firing «TCloseWater».

When the cart reaches the fourth stop position, with place «PunloadMixer» marked, it enables the «BucketUnload» output and dumps the bucket contents into the mixer. The cycle ends when «BucketEmpty» input holds true and transition «TDone» fires, marking «PReady» again. Although, «TDone» does not wait until the water level has been reached, the cart cannot start a new cycle until «PwaterRun_cmpl» is marked. This place is complementary to «PWaterRun», meaning that transition «Tstart» is inhibited while the water is still running.

Although the controller model could be immediately submitted through the code generation tools to run an embedded board, testing the controller behavior directly on the physical plant is not recommended, as any modeling mistake could damage the equipment. For example, running the conveyor belts when the cart bucket is not on the correct position, or unloading the bucket outside of the mixer area, would dump raw materials over the cart rails.

Running the controller model on the simulator provides a faster and cleaner way to test the model and find potential design mistakes. However, the controller reacts to a

series of signals and events coming from the plant, that are not available when the controller is simulated alone. As a result, to simulate the controller, the user has to manipulate the model inputs according to the expected plant behavior.

Manually simulating the plant behavior is an error prone task that requires full user attention and consumes time. To mitigate this problem, the simulator has the ability to display waveforms and save the simulation history to files stored on the server. The saved files might be used to quickly repeat previous simulations without the risk of entering wrong input sequences. When a model suffers changes, the simulation repeat function automatically detects changes in the output signal waveforms.

However, even the automatic repetition of debug sessions has limitations. The debug sessions usually correspond to well defined use-cases and do not account with unexpected behaviors, that were not forethought by the developers.

As the controller is based on a Petri net, the Petri net part of the model may be converted into a IOPT model. Figure 34, from the tools chapter, contains the state-space graph of controller model, calculated using the IOPT-Tools model-checking subsystem. This graph may be used to detect the reachability of states that were not forethought.

In the case of the current model version, no undesired states were found. However, a previous version of the controller contained a mistake that was found with the help of the state-space graph, that contained thousands of states. The mistake was corrected with the addition of the complementary place «PwaterRun_cmpl» to prevent restarting a new kart cycle before the water has reached the desired level.

Unfortunately, the IOPT model checking tools do not automatically detect all mistakes: As this option discards the dataflow part of the original DS-Pnet models, when undesired states are found the user must manually confirm if the conditions that lead to these states are affected by possible dataflow guards or transition input events.

As discussed earlier, a more flexible approach involves the design of a plant simulation model, to study complete systems composed of the controller, plant and the respective signal and event interconnections.

Figure 48 presents the top model combining a controller and a plant. The arcs at the center transmit the control signals used to drive the plant actuators and the arcs at the left transmit sensor signals from the plant to the controller. These arcs are displayed using a symbolic mode to avoid drawing multiple curves crossing the entire model. The model is autonomous, except for a single start-button input that is managed by the user. However, after pressing this button the entire simulation runs autonomously.

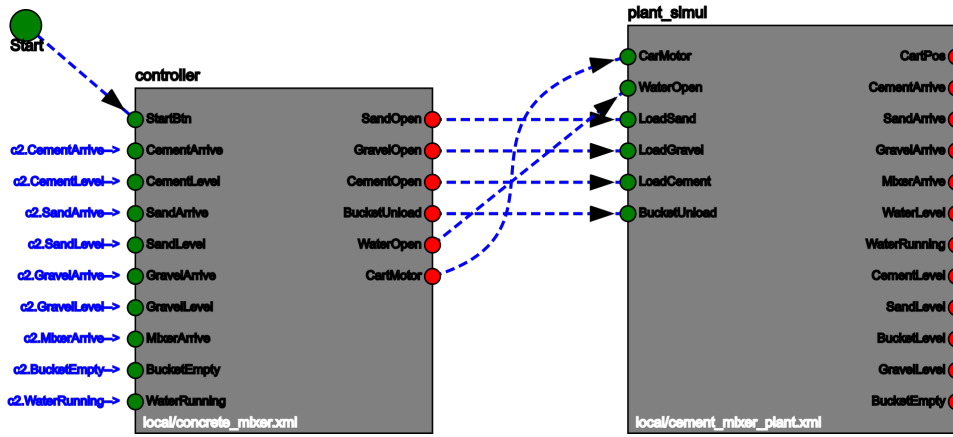


Fig. 48: Cement mixer main model: controller + plant

The plant model, found in figure 49, contains only dataflow elements and has an external interface complementary to the controller. This model is centered around three signals «CartPos», «WaterLevel» and «BucketLevel», that define the plant state.

The data flow on the center, «ModeCart», «Water» and «LoadBucket» are used to calculate the plant state signals, increasing or resetting the respective values according to the inputs coming from the controller. The dataflow operations on the right compare the value of the state signals with several threshold values: to detect if the cart position had reached each of the four stop locations, to check if the water level is full and verify if the bucket level has reached the correct thresholds for cement, sand and gravel, or is empty.

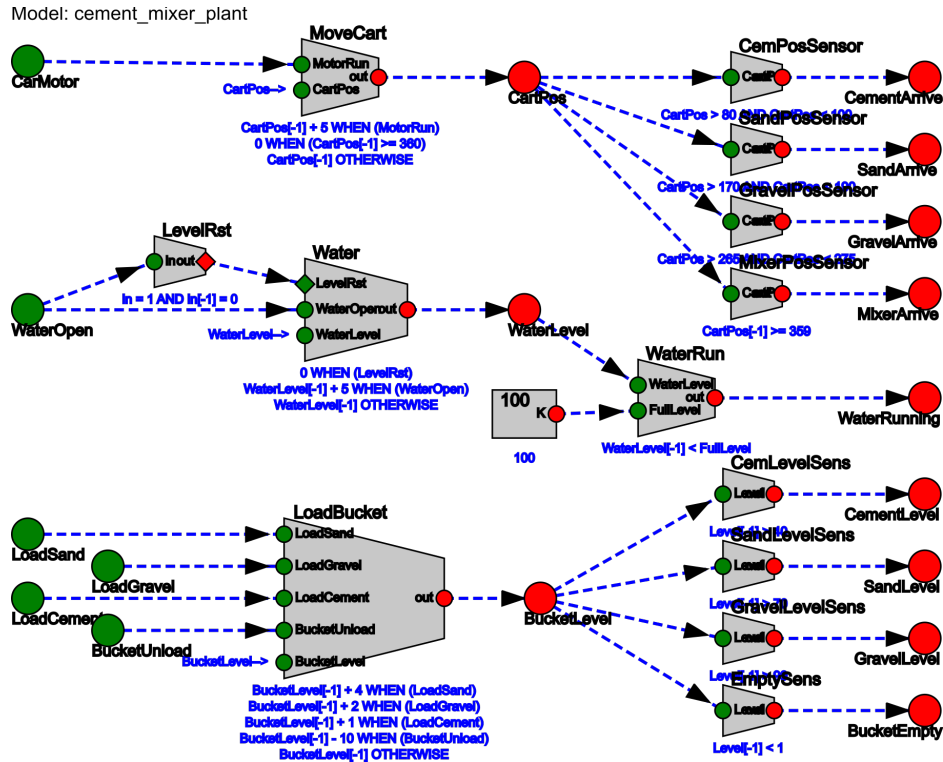


Fig. 49: Cement mixer plant model

After setting the «Start» input, the simulator state-space exploration function may be applied to the top model, generating 147 steps until reaching a deadlock (the last two steps were repeated). However, replaying the saved history or inspecting the resulting waveforms, it is possible to verify that the simulation run an entire cement mixer cycle and is ready to start again, waiting for a positive edge on the start button: the user must release the start button and press it again. In this case, the deadlock does indicate an error condition and corresponds to the intended model behavior.

Finally, it is important to note that in this case, the simulation of the complete system produced a total of 147 different states, while the state-space of the Petri net part of the controller contains only 18 states. This difference happens because the IOPT state-space graph does not account with the internal plant state signals that hold integer range types to represent analog variables, whose value changes continuously with time, resulting in many incremental state changes. Applying a state-space generation algorithm to the plant model without considering the controller, would produce millions of states, representing all possible combinations of controller inputs.



7 Validation Applications

To validate the proposed development formalism and the respective support tools, a set of example applications was prepared, leading to creation of four prototypes. All applications were fully developed using the proposed tools, from model edition, simulation to automatic code generation, without manually writing any line of code. The first of these validation applications was the subject of a publication on a scientific journal [28].

The validation examples consist on the following applications:

- 1 – Controller for a brushless servo motor, implemented on reconfigurable hardware, using the modular VHDL code generator
- 2 - Distributed multi-user game with graphical user interface
- 3 - Graphical console to control an industrial variable speed drive, using the ModBUS field-bus protocol
- 4 – Distributed cyber-physical system simple application with 3 nodes (1 processing node and 2 «physical» nodes)

In addition to the validation applications, another set of applications was created to assist the development of the new tools. These were small applications, used to test the editor, simulator, automatic code generation, inter-component communication and the various library components, that also resulted in prototypes. From these, the following applications should be mentioned:

- UART Serial port model, used to debug the VHDL code generator, implemented on a Xilinx Spartan 3AN board, tested communicating to a PC serial port, presented in figure 10.

- Graphical user interface test application, permitting the visualization of all types of Widget components. The behavior of the widget components can be configured and manipulated with the remote debugger tool, presented in figure 28.

- File I/O test models: used to test other components by automatically feeding sequences of data to the component input signals.

- ModBUS test application, in association with the remote debugger, was used to communicate with a ModBUS industrial programmable logic controller, being able to read and write the PLC memory, read PLC inputs and drive the output relays.

In addition to the listed examples, a MsC student used the IOPT-Flow tools to implement a library of structured events [177], based on the work of a previous PhD work [175][176], producing several components that were grouped in the «sim» library folder. Publication of the results obtained is currently being prepared.

7.1 Bushless servo motor controller

The first validation application is a closed-loop brushless servo motor controller, implemented on re-configurable hardware. The application was built using a diverse set of sub-system components, some purely data-driven and others exhibiting an event driven behavior, ideal to demonstrate the DS-Pnet modeling capabilities. As most of these components are frequently used in control and power electronics applications, they can be immediately added to the library for reuse in future models, contributing the abbreviate the development of future applications. In fact, these components are equivalent to the hardware modules that are currently packaged with specialized micro-controller devices that target the areas of control and power-electronics.

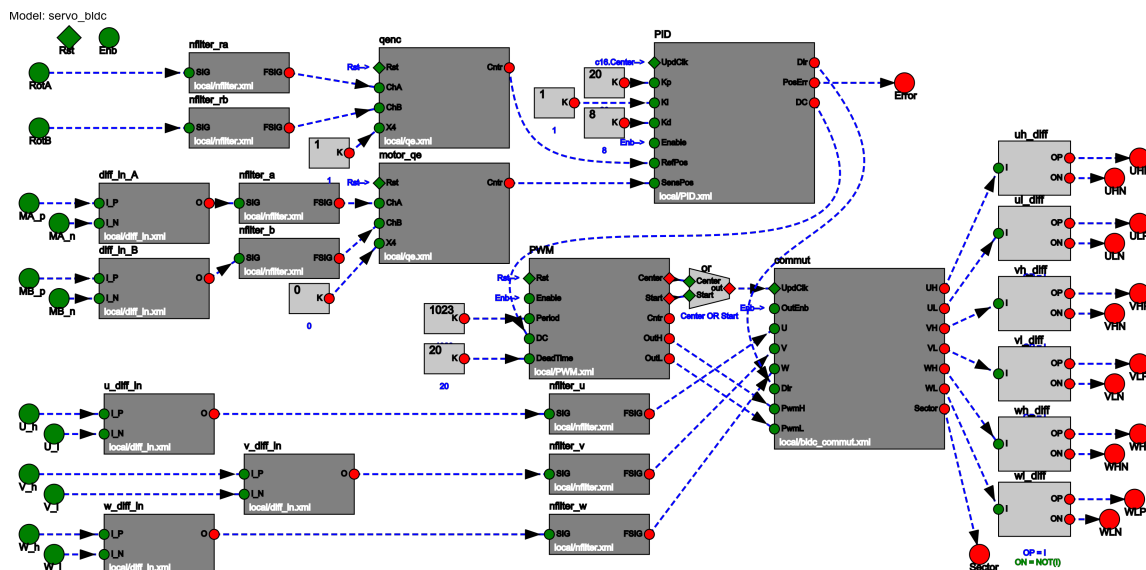


Fig. 50: Closed-loop servo motor controller top-model

7.1.1 Model development

Figure 50 presents the application top model, composed of components, arcs to perform the respective signal interconnections, constants used for tuning purposes and the external interface input and output signals. The model employs eight different component classes, some of them instantiated multiple times, reaching approximately 20 components, performing the tasks listed on table 9.

QE	Quadrature encoder pulse counter (decoder)
PID	Digital PID controller
Commut	BLDC commutation table based on commutation sensor inputs
PWM	PWM Generator (half-bridge high/low outputs with dead-time insertion)
SpdPosCtrl	Generate the reference motor position according to the Speed/Position mode
NFilter	Simple digital noise filter
DiffIn	Digital differential receiver
DiffOut	Digital differential output

Table 9: Component classes used in the application

Figure 51 presents the quadrature encoder implementation model used to track the motor rotor position, reacting to changes in the «ChA» and «ChB» input signals to update the «Cntr» output.

Under normal operation, one of the four places «PA0B0», «PA0B1», «PA1B1» and «PA1B0» is always marked, reflecting the status of encoder A and B input channels. Place «PInit» and the four transitions connected to it, are only used to determine the initial encoder state before start counting pulses.

Whenever one of the input channels changes state, one of the eight transitions near the corners of the Petri net (TAUp1/2, TADn1/2, TBUp1/2, TBDn1/2) will fire, updating the place marking according to the new channel configuration. Observing all arcs attached to the corner transitions, it is possible to notice that they form two circular

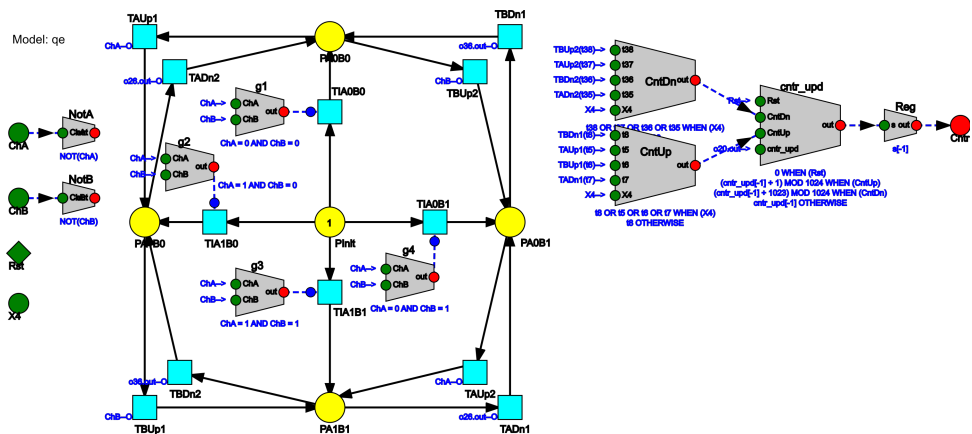


Fig. 51: Quadrature encoder model

rings. The arcs of the outer ring have arrows pointing in the CCW direction and the arcs in the inner ring in the CW direction, directly reflecting the motor rotation.

The “corner” transitions trigger events that are caught by the «CntDn» and «CntUp» dataflow operations. Depending on the mode of operation ($X4=1$ or $X4=0$), these operations perform the Boolean «or» of all events, or consider only the events coming from the transitions at the top right corner, counting every pulse, or just one pulse per cycle:

```
t8 OR t5 OR t6 OR t7 WHEN (X4)
t8 OTHERWISE
```

Next, the «cntr_upd» dataflow operation uses the outputs of the previous operations to update the rotor position counter output:

```
0 WHEN (Rst)
(cntr_upd[-1] + 1) MOD 1024 WHEN (CntUp)
(cntr_upd[-1] - 1) MOD 1024 WHEN (CntDn)
cntr_upd[-1] OTHERWISE
```

Finally, the «Reg» operation applies the delay operator «s[-1]» to the counter result, creating an implicit shift-register that will output the value calculated on the previous execution step. This operation was added to avoid the propagation of potential glitches produced by the combinatory logic used to implement the previous dataflow operations in VHDL. This operation inserts a negligible delay of just one execution step (20 ns in this prototype), but produces a clean output signal without transient glitches.

General purpose components, employed in multiple applications, may benefit from registered outputs (using the delay operator), as the output signals might be used by external dataflow operations, leading to long chains of operations implemented as combinatory logic that can impose restrictions on the maximum clock frequency.

The PID controller displayed on figure 52 is a purely data-driven model that does not include any Petri net elements. This prototype employs a single instance of the PID controller to directly control the motor position. More efficient control strategies

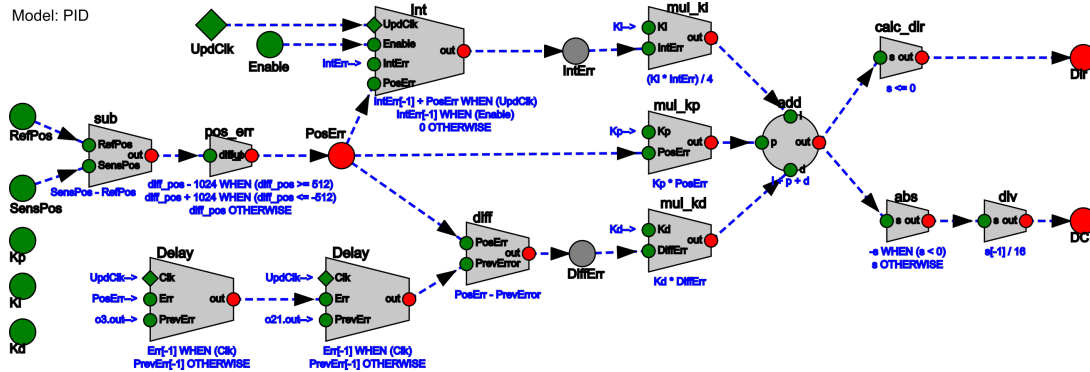


Fig. 52: Digital PID Controller model

employing cascaded controllers to control the position, speed and motor current could be implemented using three instances of this component.

Observing the model, inputs «Kp», «Ki» and «Kd» correspond the traditional proportional, integral and derivative gains, that are multiplied by the respective error signals. On the left, the «sub» operation subtracts the reference and sensed positions and the «pos-err» operation normalizes the result to the range 0 to 1024 ($0 \leftrightarrow 360^\circ$).

The two «Delay» operations at the bottom are used to sample past «PosErr» values whenever the «UpdClk» event fires. As the output of these operations is connected in sequence, the second «Delay» operation always stores the value of «PosErr» sampled two «UpdClk» events ago. As the execution step clock frequency used to run the prototype (50Mhz) is several orders of magnitude faster that the encoder feedback frequency, the «UpdClk» is used to provide a slower clock to update the derivative and integral errors without immediately saturating the respective variables.

The «Int» and «diff» operations respectively accumulate the integral error and calculate the derivative errors. Next, the error values are multiplied by the respective «Kp», «Ki» and «Kd» gains and the results are added. Two operations, «calc_dir» and «abs» extract the sign and absolute value.

As fixed-point arithmetic has not yet been added to the code generators, all operations previously described employ integer numbers, including the gains. To improve the tuning sensitivity, the absolute value result is divided by 16, producing an effect equivalent to 4bit fixed-point gains.

Finally, the values of the «DC» and «Dir» outputs are used to provide a duty-cycle for PWM component and the rotation/torque direction for the BLDC commutation model.

The component models displayed in figure 53 provide three basic functions. The «diff_in» model converts a differential signal into a single-ended signal. The differential input signal is only considered valid when the positive and negative inputs hold opposite logic values, maintaining the previous value otherwise.

The «diff_out» model, employs a double output dataflow operation to generate two signals where the OP output copies the input and ON produces a negated output. It is important to notice that these two components are translated to VHDL as pure logic

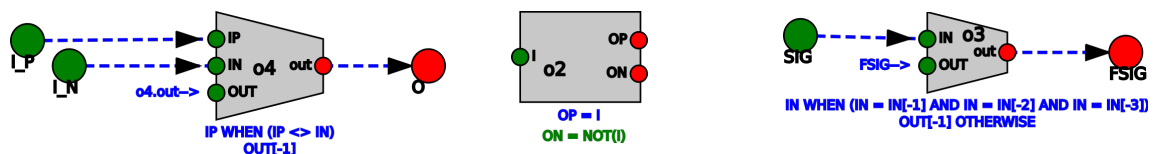


Fig. 53: The diff_in(left), diff_out(center) and nfilter(right) models

and do not produce any differential line received and line driver hardware. For improved noise immunity, these components should be replaced with real differential drivers and receiver modules provided by the FPGA manufacturers, as the IBUFDS and OBUFDS modules available on Xilinx hardware. However, this substitution should be done by manually editing the VHDL code.

The third model on figure 53 is a digital noise filter used to remove high frequency transients from digital input signals. The input signal is only considered stable after it maintains the same value for more than three consecutive execution steps. The output always contains the last stable value.

Figure 54 displays the PWM generator model. It outputs two center aligned complementary PWM signals, with dead-time insertion, used to control two power semiconductors in an half-bridge configuration. The model behavior is defined by three input parameters: period, duty-cycle and a dead-time, expressed in execution steps (clock-cycles).

The PWM generator model employs a component, presented on the right side of the figure that implements an up/down counter. This counter has two modes of operation, according to the «PCntUp» and «PCntDn» places, cyclically counting up from 0 to «Period-1» and returning back to 0. In addition to the «Cntr» output, it also produces two events to notify the instant when the minimum and maximum values of the counter are reached. These events mark the start and the center of each PWM cycle.

The 3 phase brushless DC servo motor used in this application is equipped with an encoder producing two quadrature AB signals and three commutation UVW signals. The commutation signals provide information about the absolute rotor position, used to define the correct motor phase combination required to drive the motor in each rotation/torque direction. The commutation signals may be configured in one of six valid combinations, corresponding to size 60 degree sectors.

For this validation application, a very simple trapezoidal commutation strategy was employed. Although more efficient strategies exist, it was chosen to reduce the modeling complexity. Figure 55 contains the «bldc-commut» model that implements a

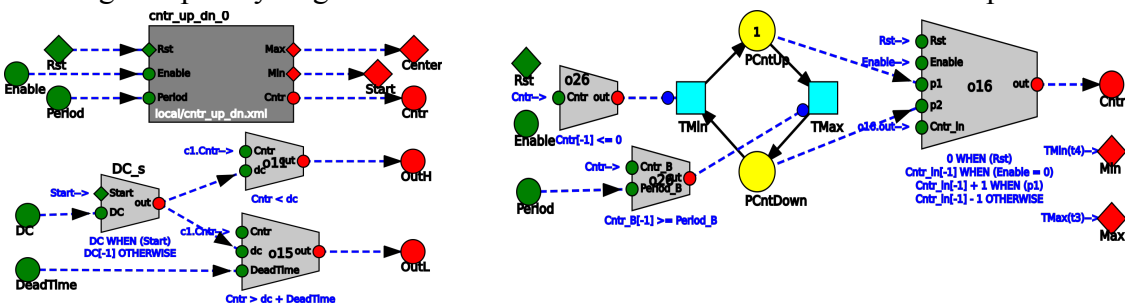


Fig. 54: The PWM generator model (left) and the cntr_up_dn component model (right)

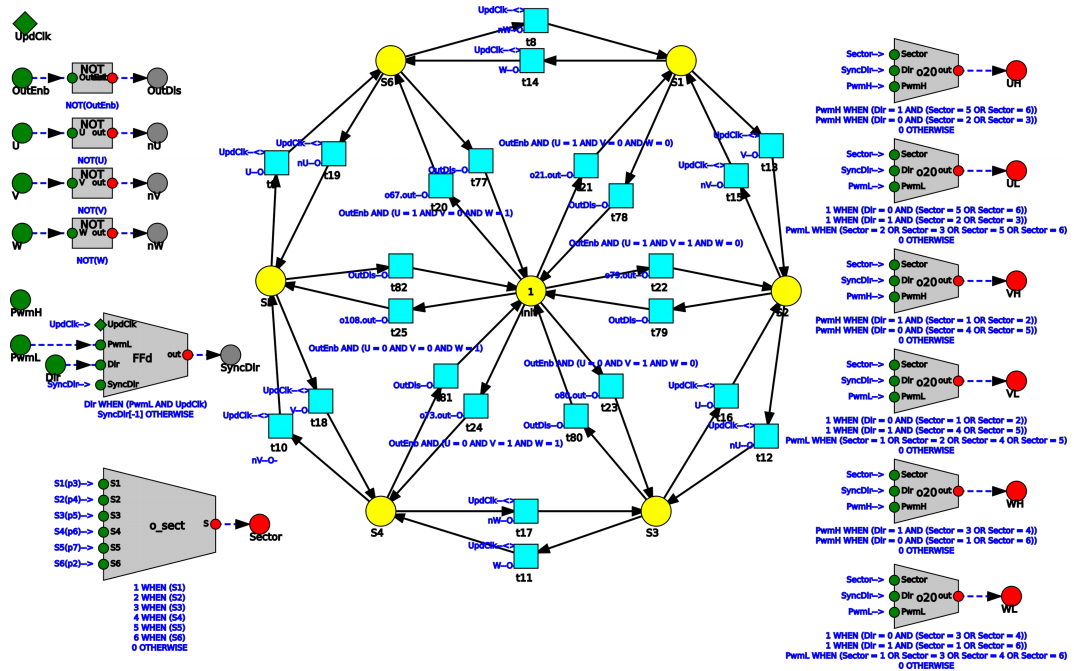


Fig. 55: The BLDC Commutation table model

trapezoidal commutation state table. The model operation is based on the three UVW commutation sensors, the rotation direction «Dir» coming from the PID component and the two complementary PWM signals. It outputs six gate signals to drive the six transistors of a three phase inverter.

Like the quadrature encoder model, the Petri net state-machine used to track changes in the commutation signals graphically resembles the physical motor commutation sectors where places «S1» to «S6» represent the six sectors. The dataflow operations on the right are used to route the PWM signals to one of the motor phases while another phase is driven low and a third phase is kept at high impedance, according the sector currently selected and the desired rotation direction.

The gate drive output signals should not exhibit any transient glitches, that could damage the power semiconductors. This way, the Petri net transitions were synchronized with the «UpdClk» event, ensuring that the selected sector only changes at the begin of every PWM cycle. In the same way, the «SyncDir» is only sampled at the begin of each PWM cycle.

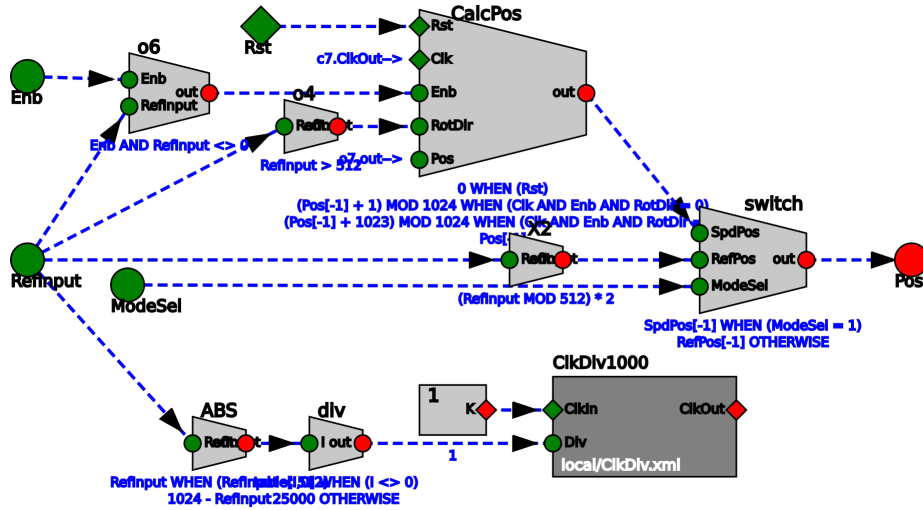


Fig. 56: The Speed/Position selector model

Figure 56 shows the last component model. The «SpdPosCtrl» component calculates a reference position for the PID controller model and according to the selected mode of operation: speed or position. In position mode, the reference position is calculated directly from the «RefInput» signal. In speed mode, the «RefInput» is used to define the rotation speed and direction. In both case, the reference input value is defined by a rotary button on the FPGA development board that contains other encoder.

In speed mode, the «CalcPos» dataflow operation periodically increments or decrements the reference position. The update frequency is calculated according to the inverse of the «RefInput» value, using a variable frequency clock divider component. Observing the clock divider component, the «ClkIn» input is driven by a constant event value 1, representing an omnipresent event that happens on every execution step. The number of clock cycles corresponding to the selected speed is calculated using a division. However, the VHDL synthesis tools only perform divisions by numbers that are powers of 2 (bit shifting). To overcome this limitation, the «div» operation employs an internal table of inverse values.

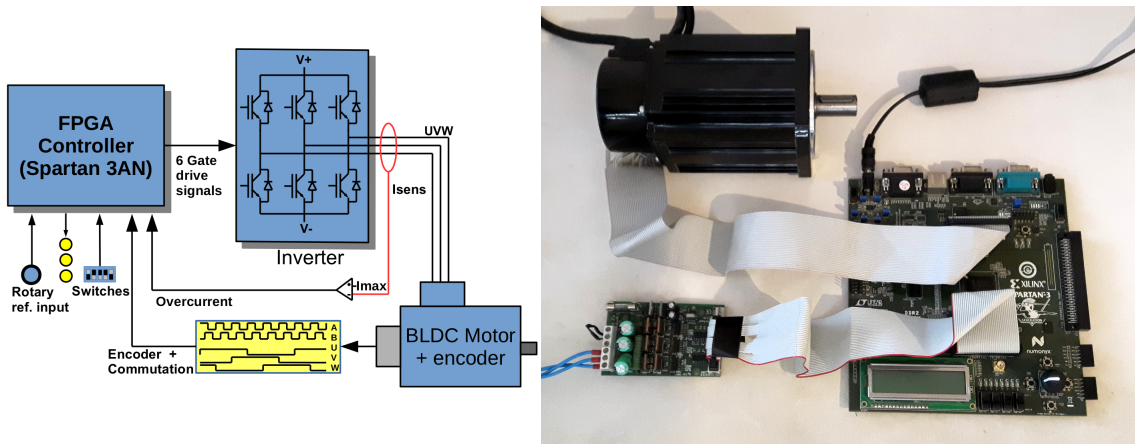


Fig. 57: Prototype diagram on the left and photo on right: BLDC Motor, FPGA and Inverter boards

7.1.2 Prototype implementation:

The application prototype was implemented using a Xilinx Spartan 3AN FPGA development board, a three phase inverter card, a BLDC servo motor and a DC power supply. Figure 57 presents a prototype diagram and a photo of the hardware employed.

The motor encoder and the inverter board are connected to the FPGA board using differential signals. The inverter provides a current limit digital output used to quickly disable the gate drive signals and protect the electronic devices. The reset and enable inputs were assigned to push buttons and DIP switches on the FPGA board. LEDs were used to monitor the PID position error.

The automatic code generation tools were applied to the top model, generating a compressed ZIP archive containing VHDL files for each component class and a main file. These files were used to create a Xilinx ISE project, with the addition of an UCF file, created manually, to associate physical FPGA pins to each input and output.

This application was created to validate the new formalism and associated tools and demonstrate the applicability to the design of power electronics controllers. This way, details about motor control and linear control theory are out of the scope of this work.

After correcting the range of internal integer signals to avoid arithmetic overflows, the prototype was successfully tested. The project run from a 50MHz clock signal, producing 25KHz PWM signals. The applications consumed just a small fraction of the available FPGA resources: 334 flip-flops and 593 latches (7% of the available slice registers), 1812 LUTs (15%), 3 hardware multipliers (15%), occupying 1300 slices (22%), including a data table containing sampled values of the $1/x$ function.

7.1.3 Results

The prototype was entirely developed using the new tools, the component models were designed and debugged using the simulator, and later the automatically generated VHDL code was deployed on a FPGA board, without the need to manually write a single line of VHDL code except for a pin assignment UCF file.

Graphical simulation contributed to accelerate the development time, permitting the early correction of design errors. Debug sessions take just a few seconds to start, leading to very fast development cycles. In contrast, the FPGA vendor synthesis tools take several minutes to optimize and create bit-stream files to run on physical hardware.

The DS-Pnet modeling formalism provides an higher level of abstraction, that lets the developers focus on modeling the desired system behavior and the underlying

control algorithms, and the automatic code generation tools hide the low level details, contributing to avoid manual coding mistakes. However, system designers must still be aware of hardware idiosyncrasies, including clock frequency limitations and carefully check the range of integer values used to store the results of mathematical operations, to avoid arithmetic overflows or underflows, a problem that may be automatically detected in future versions.

The design of power electronics controllers requires knowledge on multiple fields, including control theory, power converter topologies, software development, electromagnetic interference management and printed circuit design. Combining Petri nets and dataflows, the DS-Pnet formalism is well adapted to the design of mixed systems combining digital controllers that interface with sensors and analog readings. Comparing to traditional hardware description languages, the new formalism has a much shorter learning curve and users may start creating useful models in just a few days, releasing the designers from the time consuming software and hardware coding tasks.

The graphical dataflow notation contributes to simplify the implementation of control algorithms, that may be translated from control block diagrams. To assist this task, in the future the library may be extended with a folder containing component implementations of frequently used control blocks.

Although the generated code is not optimal, the FPGA synthesis tools optimization algorithms are able to suppress most inefficiencies. This prototype consumed just a small fraction of the available FPGA resources, leaving much space to add extended functionality. However, there is a trade-off between faster development and reduced time-to-market versus a slightly increased resource consumption. Overall, except for mass production projects, the economical effect of faster development cycles and higher flexibility surpasses the cost of using larger FPGAs.

Comparing with a previous prototype, an open-loop motor controller implemented using the IOPT-tools framework [12], the new tools brought two fundamental advantages: dataflows and model composition. Compared to pure Petri nets, the new formalism offers advantages in the design of the data-driven parts of the models, more easily expressed using dataflows than text expressions attached to Petri net nodes. The relationships between signals that were previously hidden in the place expressions, are now graphically expressed as arcs.

Although the parent IOPT-tools did not support model composition, the previous prototype was designed using a series of sub-system models that were separately processed by the IOPT VHDL code generator, producing a set of independent VHDL

entities. As a result, the main VHDL application had to be manually coded, instantiating the components and defining port maps to connect all components, without the ability to simulate the entire model and loosing part of the advantages of the automatic code generations tools.

Comparing to micro-controller based solutions, employing devices with dedicated peripherals for the control of electronic power converters, as PWM generator modules, timers, quadrature encoder interfaces and ADCs, at first sight the new application appears to be in disadvantage, as all components were developed from scratch. However, this was the first power electronics application using DS-Pnets and future applications may reuse the new components, leveling the playing field.

The new formalism offers an additional advantage: the same component model may be implemented as hardware or software, without requiring modeling changes. This flexibility permits reusing the same components on different hardware architectures, and debug algorithms on the simulator and on software targets and later implement the same component on hardware. For example, the PID controller model, typically implemented in micro-controllers as software, may be reused, both in hardware and software implementations. In fact, if the hardware peripherals offered by these micro-controllers are modeled as foreign DS-Pnet components, it would be possible to design DS-Pnet models to create software solutions that run on these micro-controllers, using the C code generator and some of the components created for this example.

7.2 Distributed multi-user game with graphical interface

This validation application was chosen for several reasons. A distributed game could not contrast more with the previous example, targeting two completely different fields of application, demonstrating that the DS-Pnet modeling formalism can be applied to a wide range of applications. Although the entertainment sector is not frequently taken seriously by the academic community, the gaming and entertainment industry corresponds to a multi-billion Euro business.

Contrary to the first application, the game was implemented as software and uses the automatic “C” code generator, including the networking layer and the «standard» library’s user interface and audio components. Due to the interactive nature of games, any software glitches or networking delays are immediately noticed by the users, providing an ideal test-bed to validate the entire tool framework. However, as the main goal of the application was the validation of the new concepts and not the game quality, a very simple «pong» game was selected, presented in figure 58.

The game was designed in several steps. In the first step, a single-user centralized model was created, to experiment with the game dynamics and user interface. After the single user version was successfully tested, the centralized model was divided in two components: the game engine and the user interface. Next, the game engine component was extended to support a second player, including internal data and new input and output signals and events to communicate with a second user.



Fig. 58: Distributed dual-user «pong» game (graphical user interface)

The double-user game runs in two distributed nodes. The first node runs the game engine and the user interface of the first player. When the second player is not connected, it can be used in standalone mode with a single player. The second node runs the user interface of the second player and connects to the first node to communicate with the game engine component. The model running on the second node looks almost identical to the first, except the game engine component now is a remote component and the arcs that connect the user interface to the game engine were connected to the input and output anchors of the second player.

The first version of the game model, before separating the game engine and the user interface, can be viewed in figure 59. Most of the model space is occupied by dark-gray components, corresponding to the graphical widgets that appear on the game window, including the ball, the “bases”, a score number, several buttons (pause, left and right) and two additional buttons used to display the «game-over» and «start-game» messages when the game is not running. On the bottom right corner, two sound-sample components are used to provide audio feedback: a “crash” sample when the game finishes and a “boing” sample when the ball bounces on the base.

Figures 60 and 61 present the two main models used to implement the dual player game. The second player model was designed starting with a copy of the first model and a few additional changes. On target property of game engine component was defined as

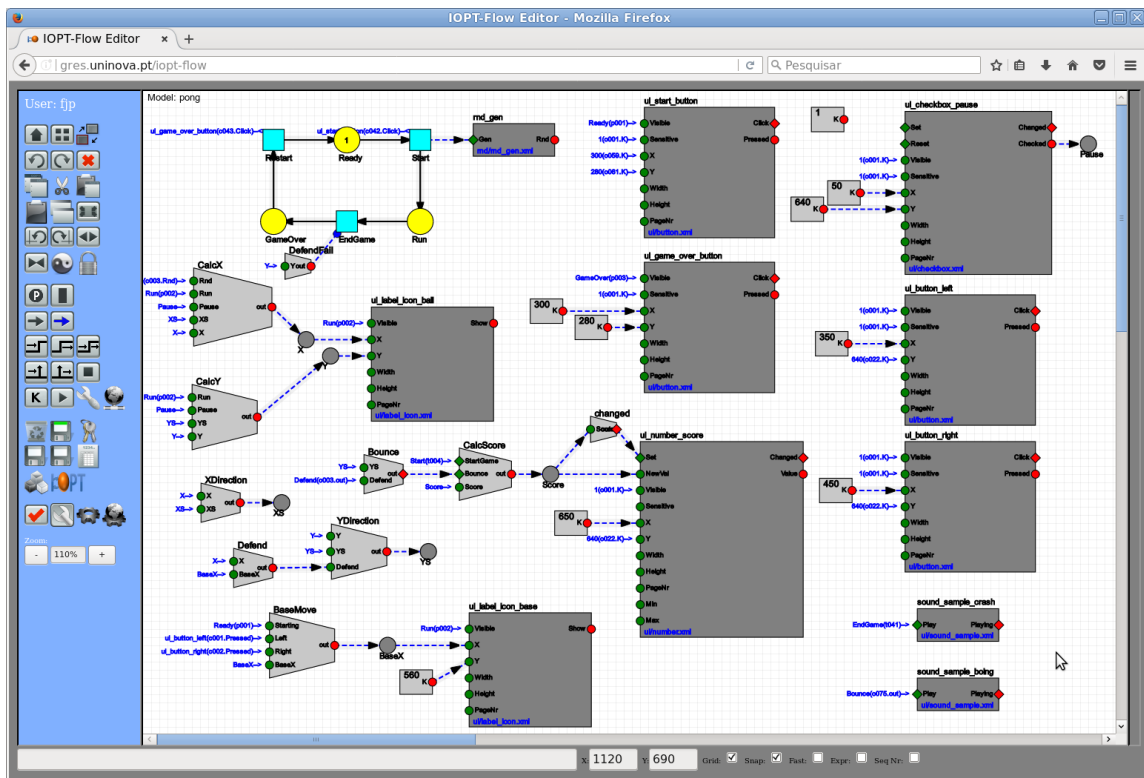


Fig. 59: The entire single-user game model fits in a single editor page

«remote» and the resource location defined with the «player1» virtual node address. Both models use exactly the same types of components and the game engine is shared by both models. This component is implemented in the first model and the second model uses it remotely.

It is important to observe the differences between both models. Each of the inputs of the shared game engine component is only driven in one of the models and never in both. This happens because the game engine component has specific input anchors for the second player and only the first player is allowed to pause and start a new game.

A special input «Player2On» is used to inform the game engine when the second player is connected. This Boolean input has both the default value and the «on-error» values defined as zero. As a consequence, «Player2On» only holds true when the second player application is connected and falls back to false when the connection drops. When the second player is disconnected, the game engine automatically copies the ball X position to the second player base position, behaving as a single-user game playing against a computer that never loses...

Finally, the second player model uses a «MirrorY» dataflow operation to mirror the ball Y position ($680 - Y$). This is required in order to reflect the game window viewed by the second player in the Y direction. This way, each player sees the corresponding

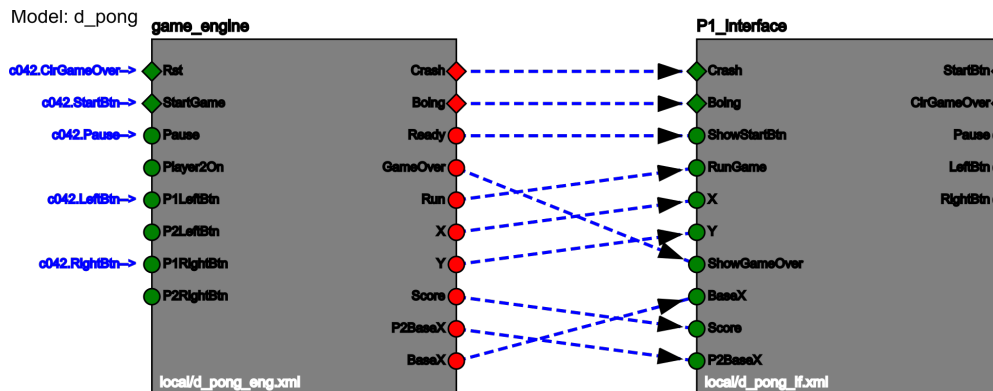


Fig. 60: The main game model: game engine + player1 interface

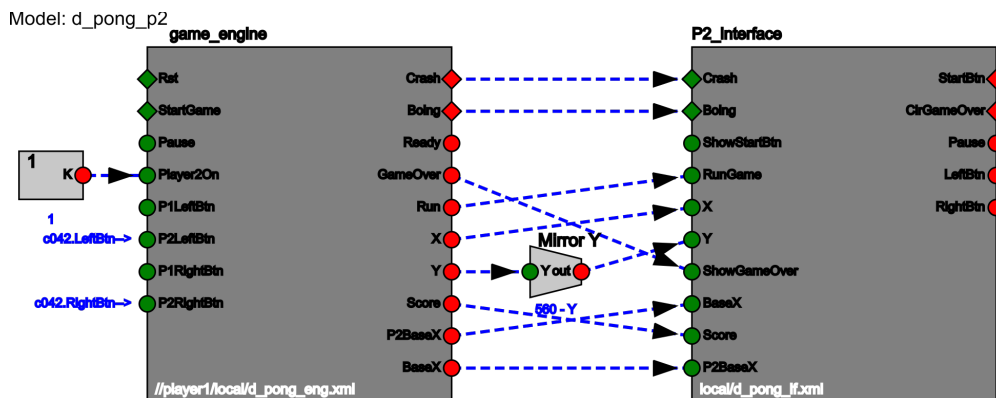


Fig. 61: The player2 model

base at the bottom of the window and the adversary at the top. To achieve this effect, the arcs connecting the «BaseX» and «P2BaseX» signals to the user interface components where also switched in each player model.

Figure 62 displays the game engine model. This model receives input from the user interface of each player and outputs information about the ball coordinates and each player base X position in real-time. It also keeps record about the game score, corresponding to the number of consecutive bounced balls.

A simple Petri net state machine is used to manage the game state: waiting to start, playing the game or displaying the game over message. The rest of the model is purely data-driven and is controlled by dataflow operations. The main variables are the ball «X» and «Y» positions and the respective «XS» and «YS» speeds, plus the «BaseX» and «P2BaseX» base positions. The ball speed variables change signal whenever the ball bounces on a player base or on a lateral wall, and the ball coordinates continuously accumulate the respective speed variable.

The score is incremented when the ball bounces on a base and the game ends as soon as the ball escapes the vertical interval between the two bases.

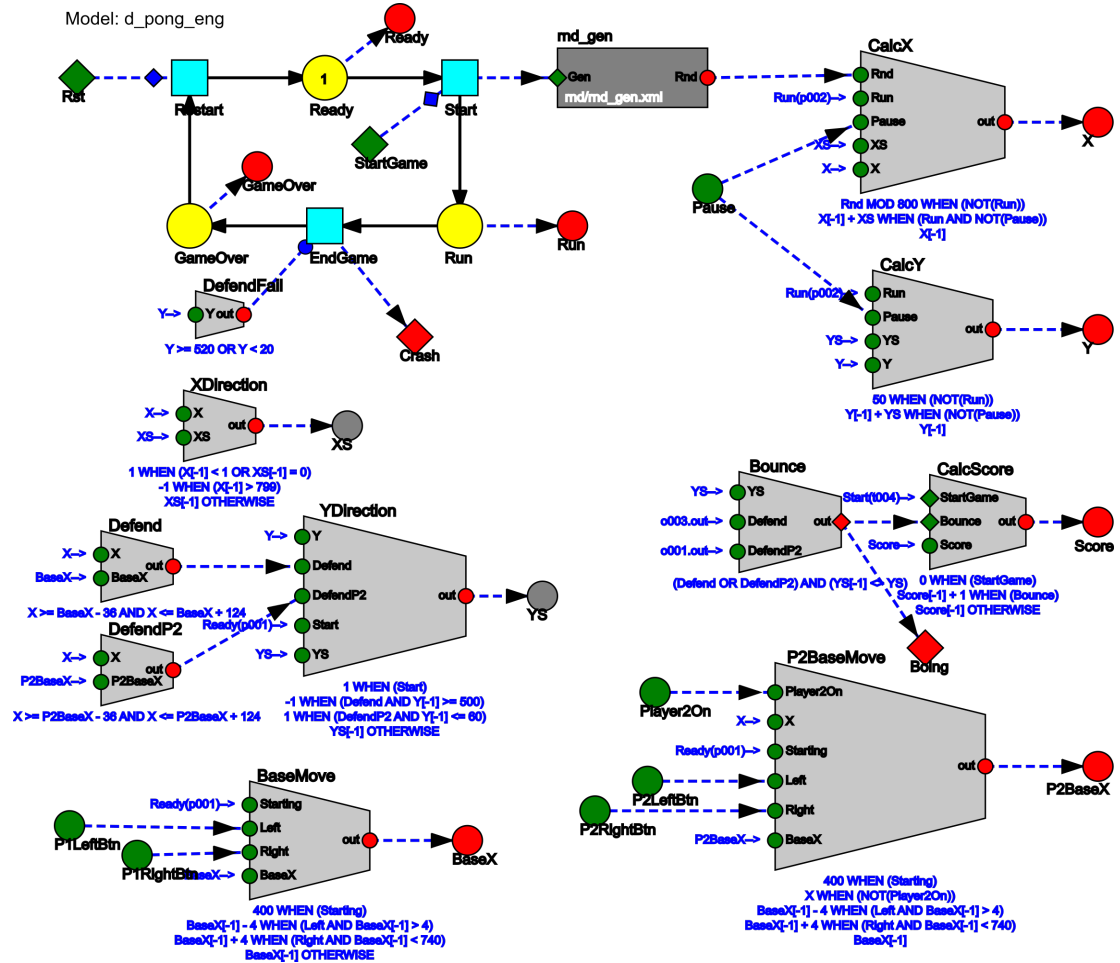


Fig. 62: The game engine component model

Figure 63 shows the interface model containing the widgets present in the game window, with the respective screen coordinates set using constants. Input signals are used to hide or show the widgets, to define the ball XY coordinates and the horizontal position of each base. The buttons produce signals used to command the base positions and pause or restart the game.

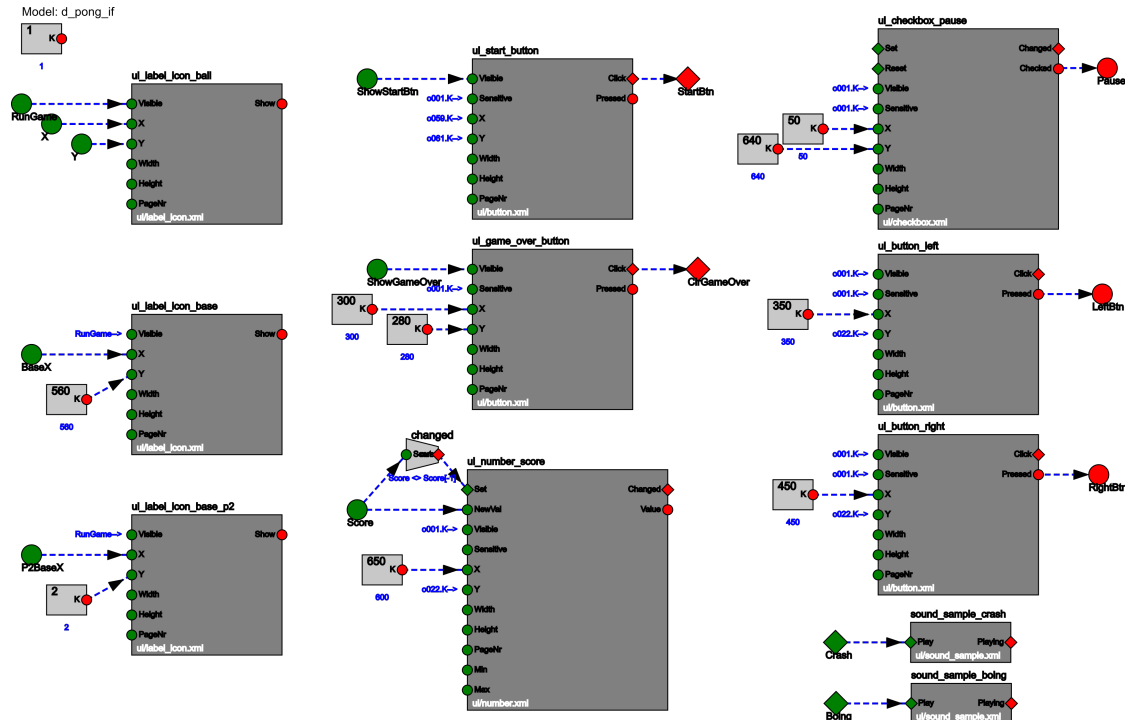


Fig. 63: Game user interface

7.2.1 Results

The two game application models, for the first and second players, were submitted through the automatic “C” code generation tools, producing two executable applications to run the distributed game. The entire build process did not involve manually writing any line of “C” code, except uncommenting a line in the project Makefile to enable linking with the GTK user interface libraries. However, the game speed is defined in the delayPause() function from file «model_io.c». This function uses the usleep() function to insert a delay between execution steps. As this delay is disabled by default, a 2 millisecond delay was inserted by uncommenting the usleep() function call.

The game was tested using the Linux operating system, but as the Gtk libraries are available on Windows and MacOS, it should be possible to compile the code on these operating systems without requiring major porting changes. However, the sound effects are based on the «pulse-audio» library that may not be available outside Linux, but the game may still run without audio. The code runs directly on embedded linux boards as the RaspberryPi V2 and V3 boards, without any changes except for processor

architecture parameters in the project Makefile, as the Raspbian operating system user interface is based on GTK.

The distributed games runs in two nodes where the first node executes the game engine and the first player user interface and the other node executes the second player user interface. This way, the first player executable should be launched before the second.

In order to establish the communication between the two nodes, the «user_db.txt» file should contain the same username/password pair on both nodes. In addition, the «node_db.txt» file of the second node should contain a line with the «player1» virtual node pointing to the first node network address. It may be also necessary to define new firewall rules on the first node to open the TCP port 9000 to the second node.

The distributed game was tested both on local networks and over a long distance connection, with the first player located at the south-west Alentejo coast and the second player in Oporto, using a proxy in Lisbon to bypass the firewall of the first player router. Even with both players located more than 500Km away, the game was still perfectly playable. The second player observed just some sporadic discontinuities in the ball movement, without game consequences.

The first single user model fits in a unique editor page, and was developed in less than four hours, where half of that time was consumed searching for appropriate icons on «iconfinder.com» and experimenting the game with different sets of images and sound samples. Next, the distributed version of the game was developed in just one hour, as the model had already been divided into the user-interface and game-engine components.

An implementation of this game using traditional development tools would require much more than a single page of code. For example, an implementation using the C programming language would require multiple pages of code just to initialize all libraries employed, create the user interface and setup the TCP/IP communication. Except if a dedicated commercial game engine were employed, the game implementation would require many hours of work and some proficiency in the usage of many programming libraries to implement the networking, audio and graphic user interface parts of the game. In contrast, the distributed games could be implemented using the new tools by a novice user with just a few days of training.

Finally, game design is a multi-disciplinary area, involving graphic designers, sound engineers, mathematicians and computer programmers. The availability of graphical formalisms may contribute to reduce the range of expertise requirements, as

the graphical nature of Petri net and dataflow languages may appeal to mathematicians and designers without deep computer science knowledge.

Quantitative data about communication performance over local networks and long distance connections using different embedded computational devices has been presented in [22]. This publication covered an early version of the JSON/HTTP communication protocol, used by the IOPT-Tools remote debugger application. However, the new protocol version offers enhanced performance as it employs subscriptions to filter the subset of remote model meta-data that should be transmitted, while the previous protocol version always transmits information about the entire model.

7.3 Graphical console for an industrial variable speed drive

The third validation application implements a graphical user interface to control and monitor an industrial variable speed drive that controls an induction motor, communicating using the ModBUS field-bus protocol. This example was selected to demonstrate the ability to integrate DS-Pnet models in industrial environments and cooperate with existing automation systems created using legacy languages and development frameworks. As today there are millions of automation applications running on the industry, the adoption of the next generation automation systems, based on IoT and CPS paradigms, depends on the capability to communicate and cooperate with the legacy systems.

Contrary to the previous application, the core of this model is a Petri net. This application continuously monitors a variable speed drive, reading instantaneous motor speed and current values, to display the respective waveforms in a scope widget. In addition, the user has the ability to start, stop, reverse and define the motor velocity. All of these variables are accessible through the ModBUS interface, to read or set new values. However, as these values are not placed into consecutive ModBUS registers, only one value may be read at a time, requiring a Petri net state-machine to continuously scan the monitored values and transmit user commands.

Figure 64 presents the console user interface, running on a laptop PC connected to a «Delta» (www.deltaww.com) variable speed drive using an USB/RS485 serial adapter.

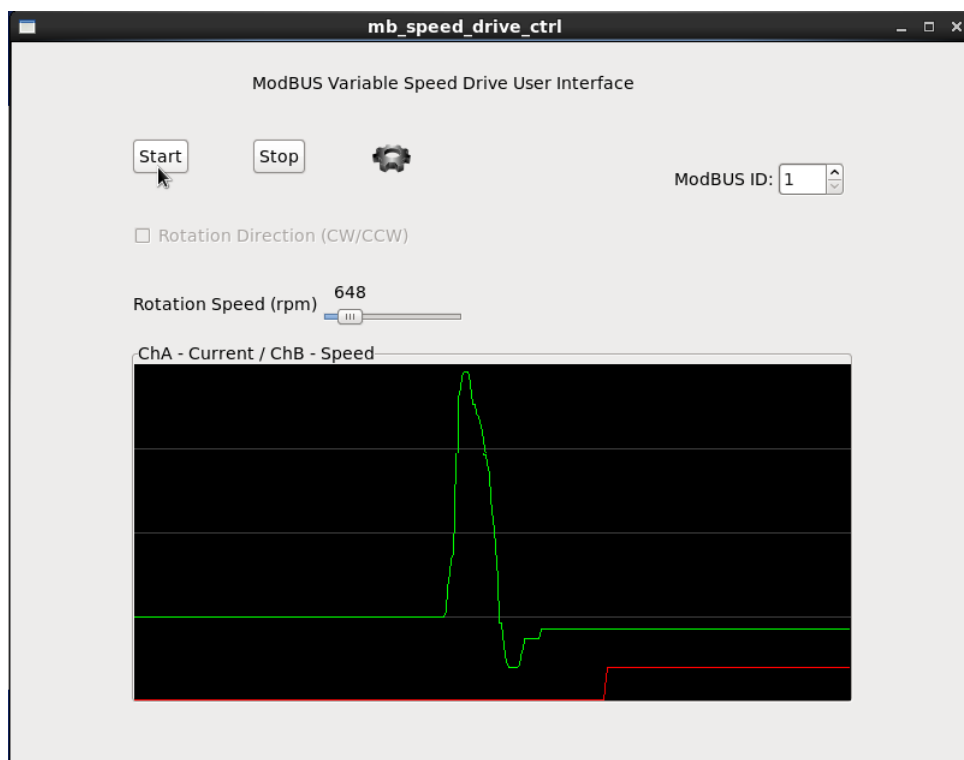


Fig. 64: Variable speed drive console application (running on a laptop PC)

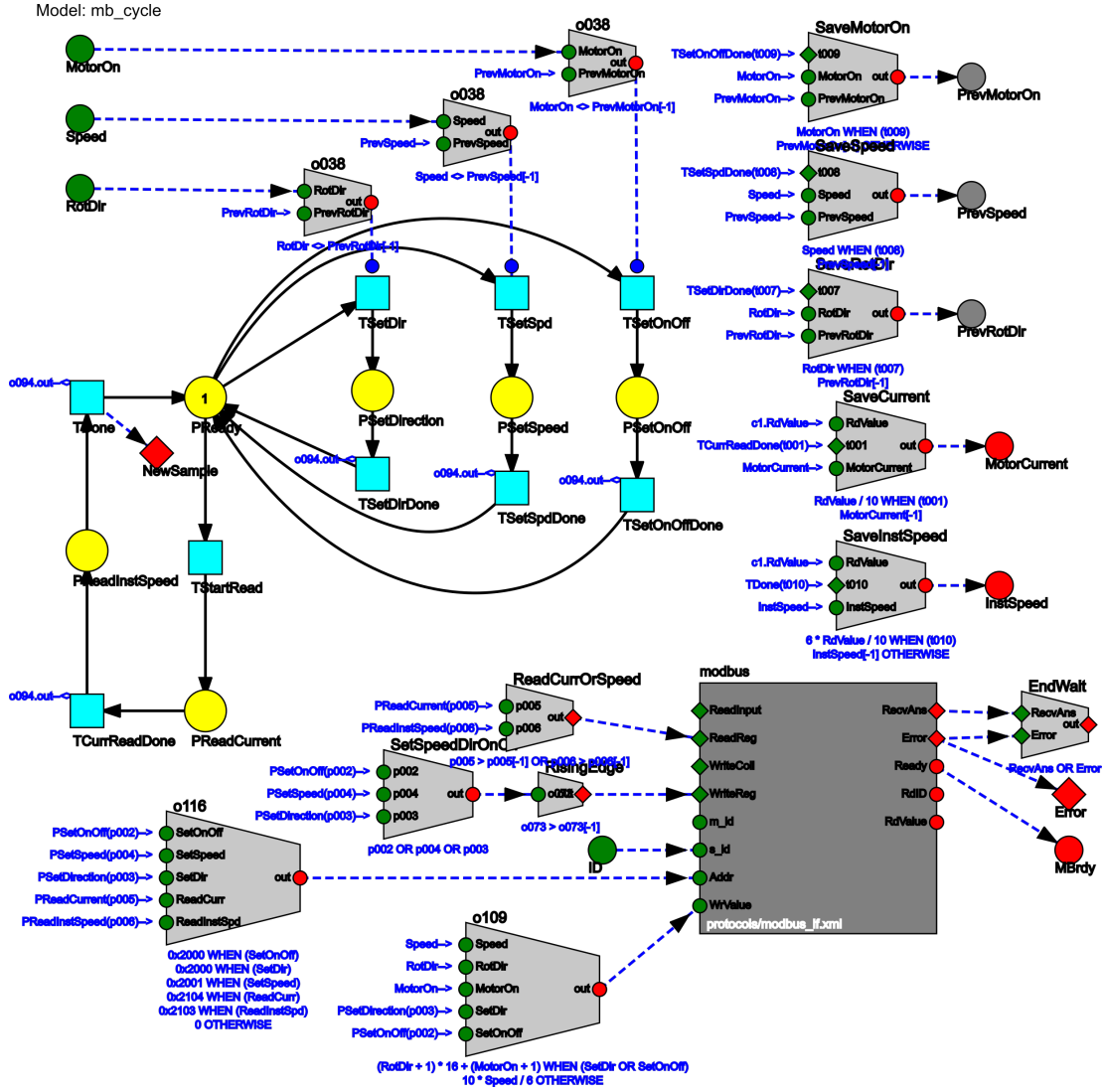


Fig. 66: ModBUS Scan-cycle model

Unfortunately, ModBUS communication can be relatively slow as older devices employ low baud rates (9600, 19200 baud) and response times may reach more than 0.1 sec. To handle this, the ModBUS interface component works in an asynchronous non-blocking way, sending ModBUS commands when it receives input events and later producing output events when answers arrive from the bus.

Communication with the selected ModBUS slave device is orchestrated by the Petri net part of the model presented on figure 66. Under normal operation, the loop on the left side (PReady, TStartRead, PReadCurrent, TCurrReadDone, PReadInstSpeed, TDone) is continuously «running». However, when any of the «MotorOn», «Speed» or «RotDir» inputs change value, this loop is interrupted to transmit the new changed value to the respective ModBUS register on the slave device. When this happens, one of the transitions «TSetDir», «TSetSpd» or «TSetOnOff» enters in conflict with transition «TStartRead», that is solved using priorities: the former transitions have priority over

«TstartRead», meaning that one of them will immediately fire as soon as «PReady» is marked.

The three guard operations above these transitions (top left) compare the current input values with saved copies. The three operations on the top right corner update the saved copies as soon as a changed value has been successfully transmitted. The two additional operations on the right side save the last motor current and speed values read from the drive, applying a scale transformation to the desired units (speed from 0.01HZ to RPM and current using a 0.1 Amp scale).

Each ModBUS command is performed in two steps, starting with the emission of a request event and waiting for an answer event, corresponding to sequences of two transitions. All transitions whose name ends with «Done» mark the reception of a command answer. The application uses the «ReadReg» and «WriteReg» ModBUS commands to read and modify memory registers on the slave device. The remaining commands «ReadInput» and «WriteCoil», to access possible physical inputs and outputs of the device, were not used in this application. The read register commands are used to read current and speed values and the write register to set the speed, direction and start or stop motion. The operations «ReadCurrOrSpeed» and «SetSpeedDirOnOff» trigger events immediately after places whose name start with «PRead» or «PSet» are marked.

The operation «o116» selects a ModBUS register address according to the place marked, corresponding to the type of value being read or written. These addresses are dependent on the manufacturer and often change on different model types. In the same way, the operation «o109» defines the value to be written, applying the correct scale transformation. For example, speed is converted from RPM to 0.01HZ.

The ModBUS scanner produces two output events, «Error» to notify possible communication errors and the «NewSample» to shift a new sample into the scope data and update the waveforms with the values of the «MotorCurrent» and «InstSpeed» output signals.

Figure 67 displays the user interface component. It contains all widgets that appear in the application window, plus a set of constants to define the screen coordinates and size. Two operations transform the scope data sampled inputs to fit the entire scope range. A simple Petri net is used to manage the motor status: idle or running. This information is used to hide or display an icon according to the motor status and also to inhibit the sensitivity of the rotation direction check-box, preventing the user from switching directions while the motor is running: changing the rotation direction while the motor is moving at high speed may cause mechanical hazards.

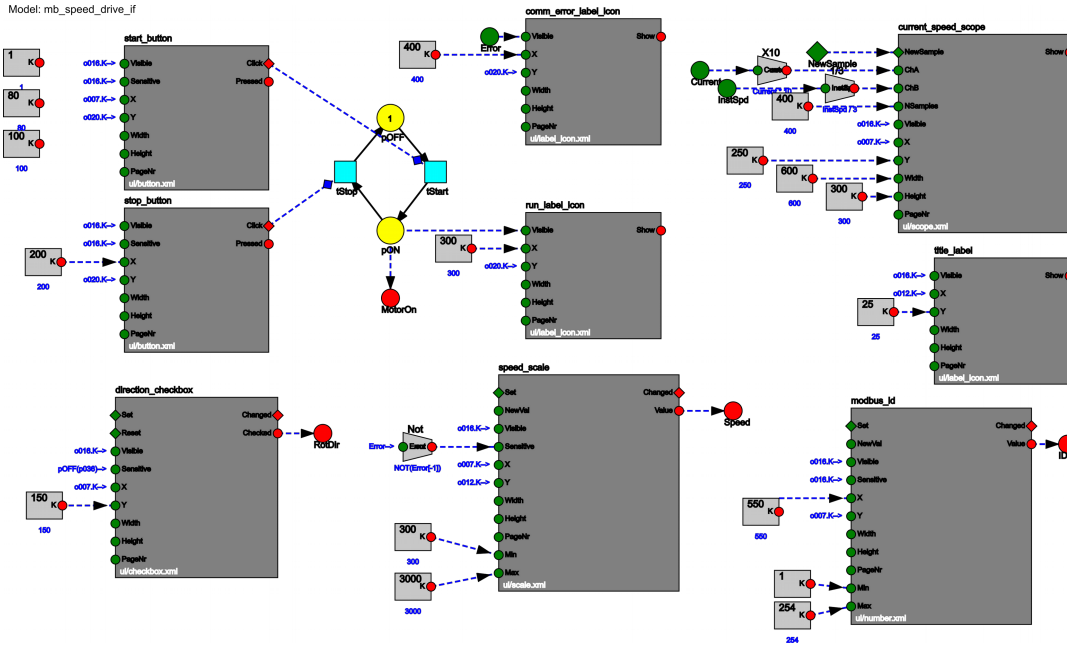


Fig. 67: The console user interface component

7.3.1 Results

In the same way as the previous validation applications, all code was generated automatically without any human intervention. The application was developed in just a few hours, including the debug of the scan cycle component and fine tuning the position of each user interface widget on the screen.

The prototype application was executed on two devices: a Linux laptop PC and a raspberry PI 2 card equipped with a «hat» board containing a 480x320 LCD and a touch screen. The prototype employed an USB/RS485 adapter to communicate with a DELTA-Automation variable-speed drive controlling the spindle motor of a CNC machine.

Contrary to the initial expectations, most of the development time was occupied with the graphical layout of the user interface: As currently there is no way of previewing the application window in the simulator, it required multiple iterations of model-edition, code generation and the respective compilation to view the change effects.

As the IOPT-Flow framework aims the rapid development of applications, this raises the need for a new user interface builder application. Such an application would start with an empty window and let the user place new widgets from a palette menu, interactively positioning and sizing each widget. As a result, the interface builder would automatically create a user interface component model containing all widget instances and the respective positioning constants. Widget outputs and unconnected widget inputs would be automatically mapped to the component external interface.

The main goal of this validation example is to demonstrate the creation of mixed applications, merging DS-Pnet models and legacy devices, cooperating with each other. Although this example focus on user interface issues, more sophisticated applications could have been designed, for example to create machine controllers involving motor controllers and programmable logic controllers with ModBUS connectivity.

Comparing with other existing technologies, a similar console could have been constructed with other tools, as Matlab/Simulink, Labview and the user interface toolkits offered by many automation vendors to design user interfaces for embedded LCD screens. In relation to Simulink and Labview, the user interfaces created using those tools usually run on personal computers, and the new application runs directly on the embedded devices, without requiring any type of licenses. Comparing with the user interface toolkits for industrial automation, the new solution offers similar functionality, although not all of these toolkits are able to display dynamic waveforms.

Regarding user interfaces, the main advantage of the new solution is the capability to transparently interact with distributed nodes from cyber-physical systems, gathering information from multiple remote nodes and controlling multiple distributed devices, just by importing remote components and connecting arcs.

A future HTML/Javascript implementation of the user interface widgets will further contribute to enhance functionality, reduce development time and lower the application hardware cost. With this solution, the same user interface models could be deployed on hardware as the LCD «hat» used in the example, or used to create remote Web user interface to run on PCs and mobile computing devices. Development time could be reduced by launching the user interface window directly from the simulator tool, without requiring compiling the “C” code. Finally, remote Web interface eliminate the need for user interface hardware on each embedded device, thus reducing manufacturing costs.

7.4 Distributed cyber-physical system simple application

A fourth validation application implements a simple distributed system. Contrary to the previous examples, this application has absolutely no practical usage and serves just to demonstrate a possible design work-flow to create distributed systems starting with a centralized model that is later split in several nodes. In addition, this example employs several distributed nodes, with two nodes containing physical components and a third computational node, forming a distributed cyber-physical system.

This is a purely academic example, focusing in the interconnection between computational and physical nodes. As a result, the nature of the computational task performed internally by the nodes is irrelevant. In this example, a computational node simply manages a lighting sequence on a row of LEDs, simulating variable speed motion.

Figure 68 presents the application model, whose development was initiated with a centralized model that was later split into three node sub-models: NodeA, NodeB and a main model. Nodes A and B were deployed on two Xilinx Zedboard cards and the main model run on a personal computer. The Zedboards were chosen due to the availability of DIP switches, LEDs and IO connectors, associated to an ARM processor running embedded Linux (Xillybus). The application uses four DIP switch inputs, to reset the application, enable or pause, increase or decrease the LED simulated motion speed.

Both nodes A and B use different instances of the same «IOX8» component, and the main node employs a controller component. The interface of IOX8 consists of 8 inputs, 8 outputs and an «enable» input. When enabled, this component just copies input values to the respective output, resetting all outputs when disabled. The controller component, running on the main node, executes a small Petri net state machine that continuously flashes a single LED, simulating a motion from left to right and back to left. Figure 69 presents the implementation models of both components.

The IOX8 component is used to provide a remote interface for input and output signals on physical devices. In NodeA there are 8 input signals, associated with hardware DIP switches, connected to the 8 inputs of the component IOX8_DIPSW. In

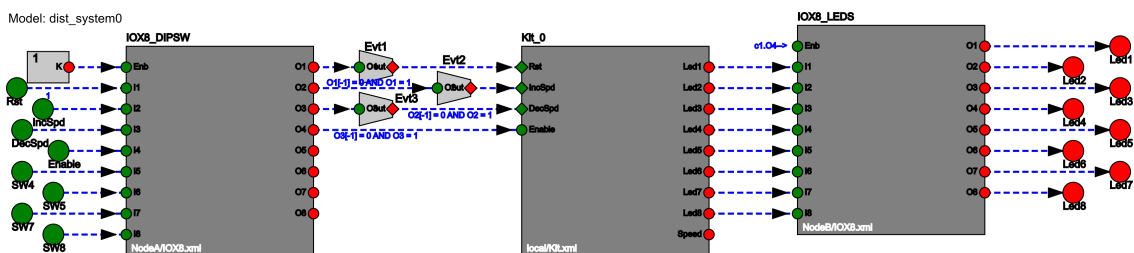


Fig. 68: Example of a distributed DS-Pnet application model

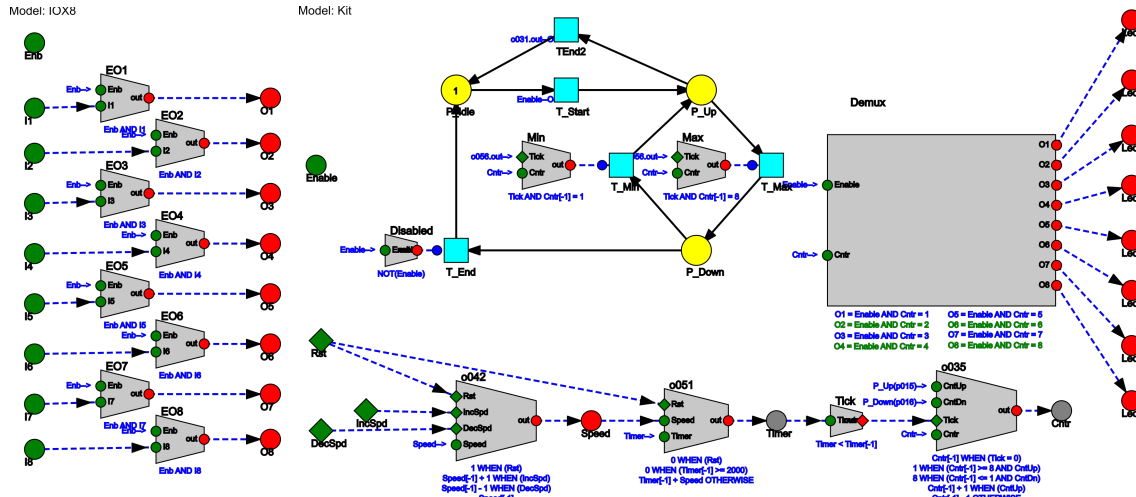


Fig. 69: Implementation models of the IOX8 (left) and Kit (right) components

contrast, the outputs of the component IOX8_LEDS on NodeB are connected to 8 Leds (wired to an IO connector). A constant true (1) value is used to permanently enable both IOX8 components.

The communication between the three components, running on different nodes, is defined using read-arcs. In the example, four arcs are used to transmit the values of the DIP switches from NodeA to the main node and another eight arcs transmit LED status information from the main node to NodeB. As the IOX8 outputs have a Boolean type and the «Reset», «IncSpeed» and «DecSpeed» inputs of the «Kit_0» component are events, three dataflow operations were inserted to detect positive edge events triggered when the DIP switch signal changes from 0 to 1.

In order to deploy the distributed system to the three computing devices, the model was divided with the automatic node-split tool into the three sub-models, displayed in figure 70. The main sub-model, presented at the top of the image, contains two instances of the IOX8 component, but these are just references to the remote components implemented in the other nodes, presented at the bottom.

The NodeA sub-model includes only the 8 dip-switch inputs and component IOX8_DWPSW, leaving the outputs of this component unconnected. The NodeB sub-model contains only a IOX8_LEDS component and the respective LED outputs, leaving the inputs of the component unconnected. Finally, the main sub-model, working as a maestro, includes the central component Kit_0, the three dataflow operations and two references to the remote IOX8 components implemented in the other nodes. The arcs interconnecting remote components are only present in the main sub-model.

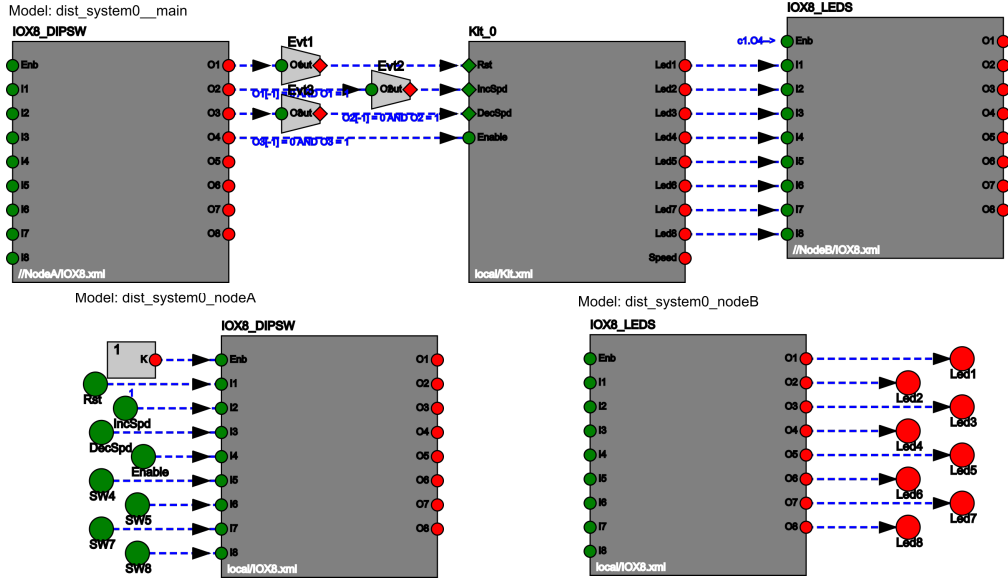


Fig. 70: Sub-models after node-split: Main maestro model(top), NodeA (bot. Left) and NodeB (bot. right)

7.4.1 Results

The three sub-models were submitted to the C code generator and compiled to build the executable programs to run on each node. The GNU GCC compiler was used in all cases, but nodes A and B were compiled with options to the ARM processor architecture and the main node for Intel X86-64.

All nodes, running versions of the Linux operating system, were connected using a local network. As the main application model connects to the JSON/HTTP servers running on the other nodes, the firewall rules of nodes A and B must be setup to open access to port 9000. In addition, a common user authentication file was installed on all nodes, defining the privileges for a «guest» user. As the main application only reads subscribed signals from nodeA, the user guest only needs to request «observer» privileges on this node. However, the main application will drive input signals on NodeB, requiring «master» privileges on this node. Finally, the resource location parameter of the two IOX8 components on the main model refer virtual node addresses «NodeA» and «NodeB». As a consequence, the main application node will require a «node_db.txt» file containing information about the real Internet addresses and port numbers of the other nodes plus the usernames used for authentication.

The distributed system was successfully tested with the two boards controlled by a personal computer, with multiple instances of the remote debugger application connected to each node, to monitor the evolution of each model in real-time.

In the same way as the two previous validation examples, the entire system was developed in just one afternoon, with 80% of the development time occupied in the

design of the controller main model. The tasks related to node division, communication to remote nodes and code generation were fully realized by automatic tools.

Comparing with the traditional programming languages and tools that are presently taught in Electrical engineering courses for embedded systems design (C, C++, Java, Python, VHDL etc.), the new formalism and tools allow the creation of distributed cyber-physical systems in a fraction of the development time and requiring much shorter learning paths, as previously discussed for the distributed game application.

In contrast, the IEC61499 standard for system distribution offers the concept of function blocks that communicate using input and output signals and events, from where the proposed formalism borrowed these concepts, would permit defining an equivalent system with comparable effort and development time. However, the proposed solution communicates directly over TCP/IP and the distributed components can be located at any place in the world with the Internet connectivity. To achieve the same results using IEC61499 would require the addition of special function blocks that implement publisher/subscriber and client/server communication on each node, resulting in more complex models.

8 Conclusions and future work

The contributions presented in this document were elaborated to provide an answer to the three research questions described in chapter 1 and validate the associated hypothesis. From these hypothesis, only the last hypothesis from the third research question was not validated, as an alternative and more advantageous solution was chosen.

8.1 Research question 1

Regarding the first research question, the DS-Pnet formalism, presented in chapter 3, offers a feature set that was selected to allow the rapid development of distributed cyber-physical systems and embedded system controllers, combining the characteristics of IOPT Petri nets, dataflows and model composition based on components. Petri nets are used to model the controller state machines that evolve according to external events. Dataflows are employed to specify data processing operations, including mathematical transformations and digital signal processing, that may be applied to calculate output signals, condition analog sensor inputs and implement linear control algorithms. Components are used as a structuring mechanism, permitting the rapid design of new applications from libraries of previously designed and debugged building-blocks. With an external interface consisting of input and output signals and events, components provide an abstraction to the design of distributed systems composed of local and remote components, running on different nodes containing both processing and physical devices. The interconnection between local and remote components is specified in a transparent way using arcs.

Petri nets have been extensively studied by the academic community for more than 50 years, with the annual production of thousands of papers. Resulting from this

research work, a well known set of properties have been studied, with applications to the analysis and model-checking of controller models. As a result, the reactive parts of a DS-Pnet models, based on Petri nets, may be analyzed by the automatic state-space calculation and model-checking tools developed in the preliminary work, for the parent IOPTnet class.

The association of dataflows to Petri nets contributes to solve one of the most frequent criticisms about pure Petri net languages, considered ill adapted to specify the interface between analog and digital parts of mixed signal systems. Using the new formalism, dataflows are used to connect input and output signals and events to the Petri net nodes, offering an explicit graphical way to specify the dependencies between signals and control decisions, including transition input events and guard conditions. In the same way, dataflows are used to calculate output values, graphically exposing the dependencies between these signals and place marking and transition firing by the means of arcs. The same consideration may be applied to the relationship between internal signals calculated using mathematical operations, whose dependencies are graphically displayed using arcs.

Data-centric models may be designed just with signals and dataflow operations, without any Petri net nodes, as happened with several component models employed in the validation applications. This is specially true for signal filters and linear control algorithms whose state typically evolves around integration, differentiation and other mathematical operations. In contrast, pure Petri net languages encode these mathematical operations using output expressions associated with idle Petri net places, hiding the dependencies between internal signals. Using these languages, a user would typically start with a dataflow model on paper, that was later translated to output expressions. With the proposed formalism, the initial dataflow can be immediately designed in the tools.

The synchronous paradigm, means that all dataflow calculations are considered instantaneous, means that any change in input signals is immediately propagated to all dependent dataflow nodes in a single execution step, affecting both output signals and system state evolution by means of transition firing. This way, designer may break large mathematical expressions in several dataflow operations connected in series, permitting the reuse of intermediary values, increasing the model readability without compromising performance.

Controllers usually exhibit a reactive behavior. In contrast, plant models typically include mechanical and other dynamic systems that are better modeled using dataflows. With the ability to model mixed signal systems, DS-Pnets are suitable to the design of

both controllers and plant systems, enabling the construction of complete autonomous systems composed of controller and plant components and the respective interconnection arcs, used for fast model-checking.

With an external interface composed of signals and events, components provide another advantage: they can be used to encapsulate foreign systems whose interface can also be specified using signals and events, bringing extended functionality to DS-Pnet models. For example, existing integrated circuits, HDL IP cores, programmable logic controllers and other automation devices, IEC61499 function blocks, may be used as foreign components to build DS-Pnet models. In the same way, existing software written with most programming languages may also be accessed using the concept of foreign components. In this case, events are employed to invoke algorithms and methods running on the foreign code and signals are used to pass parameter data.

A library of foreign components is offered to implement frequently used tasks, including user interfaces, array and file I/O and audio. In principle, foreign components should only be used to provide extended functionality that cannot be implemented using the language core. For example, to access external resources, perform operating system calls and to use hardware peripherals.

However, foreign components may be employed on other occasions, for example, in applications requiring extreme efficiency and performance levels. It is a common expression to say that a computationally intensive application spends 99% of the time executing just 1% of the code. In this case, the critical 1% part of the code may be implemented using foreign components and the remaining application using native DS-Pnet models, benefiting from the rapid development advantages of model-based design without losing significant performance.

In addition, some algorithms are better implemented using imperative (or other) languages by software engineers, for example to manipulate large quantities of data, complex data-structures, or software packages that have already been coded and well debugged. In this case, there are still advantages in encapsulating this code in foreign components: These components may be immediately inserted in new DS-Pnet models to build centralized or distributed applications, without requiring any concerns about communications details. A foreign component may be inserted in a distributed application and transparently receive requests and data from remote clients or send request events to other remote components.

8.2 Research question 2

In order to ensure deterministic execution, the bidirectional relationship between Petri net and dataflow nodes has been studied, as presented in chapter 3, leading to creation of an algorithm to execute DS-Pnet models used as a basis to the development of the automatic code generation tools presented in chapter 4.

The dependencies between signals, events, dataflow nodes and Petri net nodes are analyzed in order to determine a precise evaluation sequence used to calculate an entire execution step, employing the concepts of micro-step and nano-step numbers that are associated to all dataflow operations and Petri net transitions, according to the definitions 13 and 14.

Models containing components are previously fused into flat models containing the nodes of all components. This way, the nodes belonging to different instances of the same component class are processed independently and assigned with different micro-step and nano-step numbers, according to the dependencies of each original component instance inputs.

The micro-step and nano-step numbers are only used to specify a calculation sequence, and do not impose any clocking sub-divisions, as the execution of an entire step adheres to the synchronous paradigm from synchronous dataflows. Under this paradigm, all calculations are instantaneous and the system state remains constant for an entire step, evolving in quantum instantaneous steps.

Potential calculation loops, where the results of an operation are directly or indirectly feed back to the respective inputs, violate the synchronous paradigm and are considered syntax errors. However, the circular dependencies are immediately detected during micro-step and nano-step assignment, preventing the application of the code generation algorithms. A delay operator used in dataflow output expressions may be used to break the circular calculation dependencies, using values calculated in previous execution steps instead of the values about to be calculated.

An automatic code generation tool was developed, producing software and hardware descriptions that execute the models behavior. Currently, the tools produce code for the JavaScript, C and VHDL target languages. The code generation algorithm employs a multi-stage strategy that produces intermediary files containing a XML representation of the model semantics independent of the final language, that are later transformed to the desired language syntax.

The output of the C code generator includes a client/server communication layer that permits graphical remote debug and monitoring of the systems deployed in

embedded devices and automate the communication between distributed nodes. The VHDL code generator may output modular or monolithic versions of the code. The modular versions contains separate entity files for each component calls, that may be individually reused on other VHDL projects.

Distributed models, employing components running on remote nodes form GALS [61][62][107] systems where each node employs a different execution clock. Components communicate with each other using input and output signals and events, whose propagation between different nodes is subject to variable network latency delays.

The Petri net part of any DS-Pnet models is divided into multiple independent sub-nets, that never span across different components. This way, the execution semantics of the Petri net part of a DS-Pnet system is not affected by component locations, except when transitions from different components are connected using events. When both components run on the same local node, then the event connection forms a synchronous channel and when enabled, both transitions fire on the same execution step. When the transitions are located on different nodes, then the master-slave relationship remains valid but there is a time delay between both firings.

A communication protocol based on JSON/HTTP [129], optimized to support the communication between distributed nodes to establish interconnections between remote components was designed. The output of C code generated automatically contains a client/server communication layer that automates the connection between multiple nodes forming a components network.

As the propagation of signals and events between different nodes is subject to external factors that cannot be controlled, including variable delays and connection drops, information interchange between distributed components is synchronized using events. When a component transmits a message to a remote peer, typically the message payload is placed on output signals before sending an event to notify the other side. Later to other side may respond using the same strategy.

To obtain deterministic behavior, the networking layer ensures that the receptor does not loose events and the payload signals arrive before, or on the same execution step, as the notification events. The message propagation time may vary, but messages arrive at destination unless a connection is dropped. Connection failures are dealt with «on-error» parameter values associated with input signals: when a connection drops the communication layer sets the on-error values and the components may act accordingly.

The input and output signals and events of the DS-Pnet component interfaces does not force any type of event handshaking between distributed components. However, a client/server use pattern borrowed from IEC61499[69][70] is presented, including state-machines models to implement the respective handshaking.

With the advent of the IoT [49] and CPS [47][48], distributed applications may be constructed using publicly available components, designed and offered by third party entities. This way, the public components may be simultaneously requested by multiple applications. To enable the concurrent sharing of the same component by multiple applications, an extension to the current DS-Pnet communication infrastructure is proposed.

8.3 Research question 3

The model checking sub-system developed for the parent IOPT Petri net class [29][6][9] is based on state-space graphs and a query system to automate property checking, as deadlocks and live-locks, and the detection of desired and undesired states in very large state-space graphs. Although IOPT nets do not employ dataflows, it supports input and output signals that influence transition firing, and the resulting state-space graphs frequently reach many millions of states. The IOPT-Tools state-space calculation employs a compilation strategy based on the C code generated automatically to execute the model semantics.

With the creation of a conversion tool that extracts the Petri net part of DS-Pnet models to build IOPT PNML documents, the former model-checking sub-system may be directly invoked from the IOPT-Flow editor to analyze the properties of the Petri net part of DS-Pnet models. As the core of many models and components is built around Petri nets, this tool has been successfully used to detect modeling mistakes.

However, the state-space graphs calculated based just on the Petri net part of DS-Pnet models do not account with the system evolution restrictions imposed by the dataflow part of the models, including guard conditions and input events used to inhibit transition firing. This way, the resulting graphs include many states that are never reached in the complete model, but the state-space graph of the entire system is a subset of the graph produced. This way, if any undesired states are not reached in the entire graph, then these states are also not reached in the subset corresponding to the real system.

Unfortunately, building state-space graphs of the full DS-Pnet models has been proven a very difficult task in terms of computational resources, requiring huge amounts of processing time and memory. For example, a DS-Pnet model containing just 3 analog

inputs, represented as 10 bit integer signals, would account for 10^{30} combinations on each execution step. In addition, these signals are often used by dataflow nodes to calculate other internal signals, for example to apply integration and differentiation operations whose results must be stored in other system state variables, leading to a further explosion in the size of state-space graphs.

As a result, an alternative approach has been used. Instead of studying just the controller models, the DS-Pnet formalism may be used to model both the controller and the controlled systems, called the plants, and the complete systems may be assembled using two components, for the controller and the plant, connected using arcs. As all the controller inputs are now driven by values calculated by the plant, and plant inputs are driven by the controller, then the resulting system is autonomous and the resulting state-space graph is usually reduced by many orders of magnitude. This approach has been previously employed using the NCES Petri net class.

The resulting controller+plant autonomous systems may be interactively simulated using the simulator tool, that saves the simulation history waveforms for posterior inspection. However, this task may be assisted using a state-space exploration function that has been added to the simulator, that computes all states until a repeated state is found. According to the repeated state found, the system may have reached a dead-lock or live-lock situation, and potential undesirable states may be found in the simulation history waveforms.

Although this strategy is dependent of the accuracy of the plant model, it can be applied to the rapid testing of real world models whose state-space graph calculation would consume enormous resources. When a system is composed of multiple sub-systems, the simulator and model-checking tools may be employed to debug and validate each sub-system, that are simulated using typical use-case scenarios, but the computation of the state-space graph of entire systems starting in the graphs of the components, as described in an hypothesis, has not been prosecuted, as the resulting graphs would still require huge amounts of storage.

Finally, the new simulator tool has the capability to store copies of the simulation history waveforms that may be used later to repeat simulation sessions, using revised versions of the same models, and automatically detect changes in the output waveforms. This functionality may be used to automate regression tests.

8.4 Results and comparison with other technologies

The proposed modeling formalism has been tested in the development of the validation applications and several small examples used to assist the debug of the IOPT-

Flow tools. The tools have been used by several co-workers (InoCAM), former PhD students and were used during the thesis work of a master degree student, that contributed to detect flaws and suggest future enhancements. Both the author and the beta-testers have previous experience in software development and computer programming teaching, that helped evaluate the new tools and compare the new formalism with other development languages.

As the proposed tools aim the rapid development of embedded systems and distributed cyber-physical systems, they should be evaluated against other languages and tools commonly employed for the same type of applications. Evaluation criteria include development speed, learning curve, output code efficiency and suitability for different types of applications, used to qualitatively compare the new tools with the following technologies:

8.4.1 Traditional programming languages (C/C++, Java, Python, VHDL, etc.):

These formalisms are currently taught in virtually every electrical engineering class on embedded system development and consequently should be the most effective tools to the job. Comparing with these languages, the new formalism provides faster development time and require a much shorter learning curve, with just a small reduction in efficiency, contributing to reduce development cost.

By providing a higher level modeling formalism, assisted by graphical simulation and model-checking tools, and automatic code generation tools that hide the low level coding details, including the communication between distributed nodes, the new formalism offers much faster debug cycles, avoids low-level coding mistakes and contributes to prevent hardware damage resulting from design mistakes. As most mistakes are detected using simulations in early design stages, when the development reaches the prototype implementation phase most errors have already been resolved, minimizing the risk of destroying mechanical parts and electronic devices employed in the physical parts of cyber-physical systems. The availability of a debugger application that enables the remote trace and monitoring of applications already deployed in physical devices also contributes to reduce development time, showing the evolution of the original models almost in real-time.

These results have been verified during the development of the validation applications. In the first example, the component implementation models used to build the motor controller application were tested with the simulator. In this application, the FPGA synthesis tools took more than 5 minutes to build a new bit-stream file to upload

in the development board. In contrast, the simulator tool can be started in a few seconds after correcting a model, providing a debug cycle more than 100 times faster. For software targets, this difference is shorter, but still requires software compilation and upload to the target devices.

In relation to the second application example, the entire game model fits in a single editor page, including the game engine, a graphical user interface and audio, and was created in just a few hours. An equivalent software application would require multiple pages of code just to initialize all libraries employed and would require longer development time and knowledge about the API of the different libraries. The same considerations can be applied to the other applications, with the addition of also requiring knowledge about TCP/IP communications.

Regarding the learning curve, the proposed formalism hides most low level implementation details and consequently a novice user may start producing useful models with just a few days of training, including distributed applications. In contrast, building equivalent applications using traditional programming languages would typically require several semester classes, including an introduction to computer programming, an optional second class on algorithms and data-structures, another class on operating systems, inter-process communication and computer networks and other class in embedded system design. As a consequence, the new tools may be taught to persons without a computer science background and may appeal to developers coming from other fields as mechanical engineering and industrial automation technicians.

The new formalism is not as flexible as traditional programming languages and many algorithms and applications may be more easily expressed using the sequential paradigm offered by imperative languages. This limitations are related to several choices made during the design of the new formalism, that aimed to offer the same execution semantics on both software and hardware targets. For example, arrays and matrices were excluded from the core language and later implemented as foreign components, due to the fact that the hardware implementation of DS-Pnet models execute one step per clock cycle and only permit to access a single array position per step. In addition, an expert developer will usually create more efficient code than the one produced by the automatic generation tools, that may require the usage of more expensive hardware. However, except for mass production applications, in cases when the number of produced copies is not very large, the gains obtained in development time and flexibility surpass the increments on hardware cost.

One advantage of the new formalism over traditional programming languages is the possibility of producing code for different targets. The same model may be tested

once and executed in software or used to synthesize hardware, maintaining the same execution semantics.

Fortunately, the previous described limitations can be mitigated using foreign components to implement the performance critical parts of the systems, and also to specify the algorithms and sub-systems that are better expressed using traditional languages. This way, new applications can benefit from the advantages of the new formalism without major performance penalties. The modeling capabilities of the new formalism may be extended with the addition of new library folders, suitable for different application fields. For example, a folder containing optimized matrix operations as addition, subtraction and multiplication may be necessary to implement computer vision systems and advanced signal processing operations.

8.4.2 IOPT-Tools

The design of the new formalism and tool framework were inspired in the parent IOPT net class and the associated IOPT-Tools framework whose development started in the GRES research group many years ago [29][54][55][56][57][58][59][60][61][62]. The parent framework also aims the rapid development of embedded controllers, and offers most of the same benefits as the new one, including simulation, model-checking, automatic code generation and remote debug capabilities [22]. However, the current version of the tools do not offer any form of model composition and data-manipulation operations are performed by mathematical expressions associated with places and transitions.

As the new formalism builds on top of the former, it offers the same benefits and adds extended functionality. The addition of dataflows brings enhanced modeling capabilities to specify mathematical transformations and graphically express the dependencies between internal signals, becoming better suited to implement data-centric models and signal processing.

The addition of components permits the creation of libraries of previously designed and well debugged components that contribute to accelerate development time. Models may be sub-divided into multiple sub-systems, contributing to reduce screen clutter and let the developer focus on individual components. Finally, components provide an abstraction to implement distributed systems, where the communication between components located on different network nodes is specified by drawing arcs between the input and output signals and events of different components. As the new C code-generation tools include an client/server communication layer, the creation of distributed cyber-physical systems is fully automated.

8.4.3 Industrial automation development languages

The development languages used to create industrial automation applications based on programmable logic controllers, include several graphical formalisms as Ladder diagrams, Grafcet, sequential function charts and function block diagrams (IEC61131-3). Later the IEC61499 standard introduced a new generic model for the design of distributed control systems, but continues to employ the former IEC6113-3 languages.

These graphical formalisms enabled the rapid application development of applications and some of them offer a short learning curve. For example, Ladder diagram is very popular among automation technicians as it permits the design of simple automation applications without much training. PLC vendors provide design tools that work from personal computers to enable the edition, simulation and graphical debug and monitoring of systems running in the PLC devices, connected to the PC via field-bus protocols. Most PLC vendors also supply user interface devices and support applications.

The proposed tool-chain offers a single formalism based on Petri nets and dataflows, supporting model structuring based and components. However, the elements of the new formalism may be combined in different ways, providing a flexible solution that can emulate the constructs of the former languages. For example, a library folder containing normally On/Off contacts and timers, may be used to draw horizontal dataflow graphs, similar to Ladder diagrams. In the same way, persons familiar with Grafcet and SFC diagrams may employ Petri nets to build state-machines, taking advantage of automated editor functions to define complementary places (limit marking to a single token) and insert semaphores around critical sections.

Even the function blocks offered by IEC61499 may be replaced by DS-Pnet components. With a similar external interface composed of signals and events, a DS-Pnet component might theoretically be inserted into an IEC61499 system and an IEC61499 function block might be used in a DS-Pnet distributed application, just requiring the creation the communication protocol compatibility code.

IEC61499 composite function blocks containing multiple interconnected basic function blocks may be replicated using components and dataflow arcs connecting the component interface signal and events, and even the IEC61499 execution control charts (ECC) could be replaced by Petri nets used to control the execution of components, in the same way as ECC trigger the execution of the execution of events, with the advantage of graphically displaying the dependencies between the algorithms and the

control chart. Finally the service interface function blocks (SIFB) may be implemented using DS-Pnet foreign components, to bring enhanced functionality and permit the insertion of legacy code.

As a conclusion, the DS-Pnet formalism provides equivalent functionality to the IEC61499 standard, but does not require learning the syntax rules of multiple languages (ECC, SFC, FBD, Ladder, etc.), as it employs just a single language, flexible enough to mimic all of the former. In addition, any DS-Pnet component may be directly connected to other components placed on remote nodes, communicating through the Internet, without requiring any interface components, and an automatic split tool may be used to break centralized models into multiple distributed nodes. In contrast, IEC61499 requires the manual insertion of special-purpose publisher/subscriber and client/server function blocks at the boundaries of each node. As a result, the new formalism may appeal to designers with previous experience on industrial automation formalisms.

Finally, the new tool-chain may be used to design both software and hardware solutions and can be used to create general purpose digital systems. In addition, the new software code generators may be used to create applications to run on embedded devices, but may also be employed to create software applications to run on personal computers, as the game validation example. In contrast, the languages used for the development of automation solutions are not usually employed for other purposes.

8.4.4 Labview and Matlab/Simulink

The Labview and Matlab/Simulink software packages have been heavily used by the academic community to build prototypes and have made some incursions in industrial environments. These are huge frameworks, supporting multiple development formalisms, mathematical solvers, and very large libraries of existing modules that provide a very fast path to create new applications. For example, Simulink supports graphical simulation and automatic code generation for both software and hardware target and third party add-ons offer support for external hardware.

However, these are commercial products, requiring expensive license costs both in terms of development tools and also in run-time licenses for the resulting solutions. In addition, the resulting applications must be attached to personal computers running the development environment to allow debug and monitoring, as compared to the proposed solution that used Web technology for the same purposes and may be monitored from mobile computing devices.

Although Simulink supports Petri nets and dataflows, the new formalism has been designed specifically to support the design of embedded and distributed cyber-physical

systems, with a precise execution semantics and a feature set designed to permit high performance implementations. For example, the VHDL code generator directly translates all dataflow operations to combinatorial circuits.

The task of reading and writing input and output signals and events connected to physical devices is automatically dealt by the automatic code generation tools, without requiring the insertion of additional blocks supplied by the hardware vendors. In the same way, Matlab and Simulink offer blocks to implement TCP/IP client/server communication that permits the creation of distributed solutions, but the proposed tool-chain offers dedicated tools to split applications after centralized simulation and allows the design of distributed applications almost in a transparent way.

Finally, the new tool framework has a very low footprint, and the current version of the tools occupies less than 5MBytes. As a result, the entire framework may be installed directly on the embedded device, with most computing intensive tasks running on the users Web browser, except the IOPT State-space generation that may be off-loaded to servers on the cloud. This way, a technician may use a simple smart-phone or tablet computer and connect to the embedded devices using wireless communications or a local network, and immediately troubleshoot systems deployed in places without Internet access.

8.5 Future work

During the beta-test and evaluation phase many flaws in the tools, automatic code generators and communication layer have been found and solved, and opportunities for future enhancements have been identified. As a result, the current version of the tools has been used as a proof-of-concept and has not yet reached the quality level of commercial software packages. New bugs certainly will appear and many usability features should be added. The following items for future improvement have been identified:

- Add support for fixed point arithmetics to all code generators
- Add support for foreign components to the JavaScript code generator
- Support form remote components to the JavaScript code using JSON/AJAX
- Implement the «standard» library components in JavaScript (as foreign components) to enable the creation of Web user interfaces
- Design a bridge between Software and Hardware components to automate co-design solutions
- New version of the communication protocol using HTTP keep-alive sessions for improved performance and authentication security

- Create a query window to inspect large simulator history waveforms
- Open secondary windows to display the evolution of component contents in the simulator and remote debugger tools
- User interface builder to interactively create user-interface DS-Pnet models
- Web-based task manager to compile and launch models with capability to start, stop, pause and set start-on-boot (automatic C code compilation from the tools, avoiding OS command line and compiler IDEs)
- Usability improvements: create operations from expressions, buses with multiple signals, etc.

The dissemination of the proposed formalism and associated tools depends on the availability of hardware platforms ready to be immediately used by potential developers. The ModBUS gateway foreign component and the isolated input/output board presented in figure 11 represent two steps in that direction, as both can be employed to build industrial applications.

A package composed of a very low cost processing board, as the Raspberry PI Zero-W, offering digital I/O, USB, Ethernet and wireless communications, able to run an embedded Linux operating system, attached to an IO board as the proposed, can provide a very competitive solution to implement distributed automation solutions. As the entire tool-chain requires less than 5MBytes of disk-space, the entire tool-chain may be installed directly on the device.

Finally, a new task-manager application must be added to the tool-chain, to compile and run the code generated automatically without requiring any third party compiler tools and command line usage. A similar task has already been performed using the former IOPT-Tools framework, that employs the GCC compiler to run the state-space generator code. In the same way, the tools have already been installed in embedded platforms, running the Apache web server on embedded Linux operating systems.

The combination of a ready-to-use hardware platform with a complete tool-chain that can execute models on the hardware device without any manual intervention, may appeal to users that have no previous experience in electrical engineering and software development, opening the field of distributed cyber-physical design to a broad audience.

Possible future research in related fields, include:

a) Experiments with high-level Petri nets and dataflows. Under certain restrictions, high level Petri nets where tokens hold information may be translated into DS-Pnet systems, where low-level nets are responsible for the token evolution, but

token data is processed by associated dataflow nodes. After applying a pre-processor transformation, to translate the high-level nodes into a DS-Pnet model, the entire tool-chain could be processed using the remaining IOPT-Flow tools. High level nets offer enhanced capabilities to model factory production systems, where tokens hold the identification of real parts moving on the production plants.

b) As future trends evolve in the direction of publicly available CPS components, the same component may need to be simultaneously used by many client applications. This way, component sharing architectures for concurrent applications, should be researched, as proposed at chapter 5. For example, new multi-processor/multi-core platforms may run multiple copies of the same component. Under such an architecture, each copy would take a single request from the queue and process it in parallel with the other copies. However, client applications would see only a single shared component.

c) Dynamic node reconfiguration. The capability to dynamically disconnect remote components and connect to different nodes may be used for different purposes. For instance, for fault tolerance and load balancing, but also to let users interactively choose different nodes.

d) Library enhancement. The suitability of the new tools for each specific field of application depends on the existence of library folders with frequently used components and algorithms. This is an opportunity for experts in other fields to build new components, probably resorting to «foreign» coding.

e) Social experiments with novice users to evaluate the training difficulties, potential enhancements and future areas of research. These experiments should have been done during this work, but were not performed due to lack of resources: time and a pool of students with no previous experience in software development. Instead the beta-testers were chosen among people with extensive knowledge about these fields.

9 References

[1] Pereira, F.; Gomes, L.; Redondo, L.; "Gerador de impulsos de alta tensão controlado por FPGA com interface gráfica e sistema de monitorização integrado", REC2011, VII Jornadas sobre Sistemas Reconfiguráveis, Faculdade de Engenharia de Universidade do Porto, 3-4 de Fevereiro de 2011

[2] Moutinho, F.; Pereira, F.; Gomes, L.; "Interface para Leitura e Escrita Concorrente de Memória RAM DDR2 em o Plataforma baseada em FPGA", REC2011, VII Jornadas sobre Sistemas Reconfiguráveis, Faculdade de Engenharia de Universidade do Porto, 3-4 de Fevereiro de 2011

[3] Pereira, F.; Gomes, L.; Moutinho, F.; "Automatic generation of run-time monitoring capabilities to Petri nets based Controllers with Graphical User Interfaces", Second IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2011, Costa de Caparica, Portugal, February 22-24, 2011, Proceedings, Series: IFIP Advances in Information and Communication Technology, Vol. 349, Camarinha-Matos, Luis M. (Ed.), ISBN: 978-3-642-19169-5

[4] Moutinho, F.; Pereira, F.; Gomes, L.; "Automatic generation of graphical user interfaces for VHDL based controllers", 2011 IEEE International Symposium on Industrial Electronics (ISIE), 1491-1496

[5] Ribeiro, J.; Moutinho, F.; Pereira, F.; Barros, J.P.; Gomes, L.; "An Ecore based Petri net type definition for PNML IOPT models" 9th IEEE International Conference on Industrial Informatics, INDIN 2011, Caparica, Lisbon

[6] Pereira, F.; Moutinho, F.; Gomes, L.; Ribeiro, j.; Campos-Rebelo, R.; "An IOPT-net state-space generator tool", 9th IEEE International Conference on Industrial Informatics, INDIN 2011, Caparica, Lisbon

[7] Pereira, F.; Moutinho, F.; Gomes, L.; Campos-Rebelo, R.; "IOPT Petri net state space generation algorithm with maximal-step execution semantics", 9th IEEE International Conference on Industrial Informatics, INDIN 2011, Caparica, Lisbon

[8] Campos-Rebelo, R.; Pereira, F.; Moutinho, F.; Gomes, L.; "From IOPT Petri nets to C: An automatic code generator tool", 9th IEEE International Conference on Industrial Informatics, INDIN 2011, Caparica, Lisbon

[9] Pereira, F.; Moutinho, F.; Gomes, L.; "A State-Space Based Model-Checking Framework for Embedded System Controllers Specified Using IOPT Petri Nets", Third IFIP WG 5.5/SOCOLNET Doctoral Conference on Computing, Electrical and Industrial Systems, DoCEIS 2012, Caparica, Portugal, Feb. 2012, Technological Innovation for Value Creation, 123-132, Springer Boston, ISBN 978-3-642-28254-6

[10] Pereira, F.; Gomes, L.; Redondo, L.M.; "FPGA controller for power converters with integrated oscilloscope and graphical user interface", IEEE International conference on Power Engineering, Energy and Electrical Drives (POWERENG), 2011, Malaga, Spain

[11] Pereira, F.; Gomes, L.; Redondo, L.M., "Multifunctional Controller Architecture for Solid-State Marx Modulator Based on FPGA," Plasma Science, IEEE Transactions on, vol.42, no.10, pp.2991,2997, Oct. 2014

[12] Pereira, F.; Gomes, L.; "FPGA based speed control of Brushless DC Motors using IOPT Petri Net models." Industrial Technology (ICIT), 2013 IEEE International Conference on. IEEE, 2013

[13] Gomes, L.; Moutinho, L.; Pereira, F.; "IOPT-tools - A Web based tool framework for embedded systems controller development using Petri nets." Field Programmable Logic and Applications (FPL), 2013 23rd International Conference on. IEEE, 2013

[14] Pereira, F.; Gomes, L.; "Automatic synthesis of VHDL hardware components from IOPT Petri net models." Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE. IEEE, 2013.

[15] Pereira, F.; Moutinho, F.; Gomes, L.; "Model-checking framework for embedded systems controllers development using IOPT Petri nets." Industrial Electronics (ISIE), 2012 IEEE International Symposium on. IEEE, 2012.

[16] Pereira, F.; Moutinho, F.; Gomes, L.; "IOPT-Tools - Towards cloud design automation of digital controllers with Petri nets". ICMC'2014 International Conference on Mechatronics and Control. July 03-05 2014, Jinzhou, China

- [17] Gomes, L.; Moutinho, F.; Pereira, F.; Ribeiro, J.; Costa, A.; Barros, J.P.; "Extending Input-Output Place-Transition Petri nets for Distributed Controller Systems development", ICMC'2014 International Conference on Mechatronics and Control, July 03-05 2014, Jinzhou, China
- [18] Pereira, F.; Gomes, L.; "Minimalist Architecture to Generate Embedded System Web User Interfaces." *Technological Innovation for the Internet of Things*. Springer Berlin Heidelberg, 2013. 239-249.
- [19] Pereira, F.; Gomes, L.; "Cloud Based IOPT Petri Net Simulator to Test and Debug Embedded System Controllers." *Technological Innovation for Cloud-Based Engineering Systems*. Springer International Publishing, 2015. 165-175.
- [20] Costa, A.; Barbosa, P.; Moutinho, F.; Pereira, F.; Ramalho, F.; Figueiredo, J.; Gomes, L.; "MDA-Based Methodology for Verifying Distributed Execution of Embedded Systems Models." *Formal Methods in Manufacturing Systems: Recent Advances*. IGI Global, 2013. 112-135. Web. 9 Jan. 2015. doi:10.4018/978-1-4666-4034-4.ch006
- [21] Gomes, L.; Costa, A.; Barros, J.G.; Moutinho, F.; Pereira, F.; "Merging and Splitting Petri Net Models within Distributed Embedded Controller Design." *Embedded Computing Systems: Applications, Optimization, and Advanced Design*. IGI Global, 2013. 160-183. Web. 9 Jan. 2015. doi:10.4018/978-1-4666-3922-5.ch009
- [22] Pereira, F.; Melo, A.; Gomes, L.; "Remote operation of embedded controllers designed using IOPT Petri-nets", 13th IEEE International Conference on Industrial Informatics, INDIN 2011, Cambridge, 22-24 July 2015, Cambridge, UK
- [23] Pereira, F., Moutinho, F., Barros, J. P., Costa, A., & Gomes, L. (2015). Executable models for Embedded Controllers Development-A Cloud Based Development Framework. In *P&D@ MoDELS* (pp. 40-43)
- [24] Pereira, F.; Moutinho, F.; Ribeiro, J.; Gomes, L.; "Web based IOPT Petri net Editor with an extensible plugin architecture to support generic net operations." *IECON 2012-38th Annual Conference on IEEE Industrial Electronics Society*. IEEE, 2012
- [25] Pereira, F.; Moutinho, F.; Gomes, L.; "IOPT Tools User Manual - Version 1.1", 2014, available at http://gres.uninova.pt/iopt_usermanual.pdf
- [26] Pereira, F.; Gomes, L.; "Combining data-flows and petri nets for cyber-physical systems specification", *Technological Innovation for Cyber-Physical Systems - 7th IFIP WG 5.5/SOCOLNET Advanced Doctoral Conference on Computing, Electrical*

and Industrial Systems, DoCEIS 2016, Proceedings. Vol. 470 2016. p. 65-76 (IFIP Advances in Information and Communication Technology; Vol. 470).

[27] Pereira, F.; Gomes, L.; "The IOPT-Flow framework pairing Petri nets and data-flows for embedded controller development", IECON 2016 - 42nd Annual Conference of the IEEE Industrial Electronics Society, Florence, 2016, pp. 4832-4837. doi: 10.1109/IECON.2016.7794152

[28] Pereira F.; Gomes, L.; "The IOPT-Flow Modeling Framework Applied to Power Electronics Controllers," in IEEE Transactions on Industrial Electronics, vol. 64, no. 3, pp. 2363-2372, March 2017. doi: 10.1109/TIE.2016.2620101

[29] Gomes, L.; Barros, J.; Costa, A.; Nunes R., "The Input-Output Place-Transition Petri Net Class and Associated Tools", INDIN'2007 - 5th IEEE International Conference on Industrial Informatics, 23-26 July 2007, Vienna, Austria

[30] Reisig, W., "Petri nets: an introduction"; New York, USA: SpringerVerlag New York, 1985

[31] Petri, C. A. (1980). Introduction to general net theory. In Net theory and applications (pp. 1-19). Springer Berlin Heidelberg.

[32] Peterson, J. L. (1977). Petri nets. ACM Computing Surveys (CSUR), 9(3), 223-252.

[33] Murata, T. (1989). Petri nets: Properties, analysis and applications. Proceedings of the IEEE, 77(4), 541-580.

[34] Kukkala, P.; Riihimäki, J.; Hannikainen, M.; Hamalainen, T.D.; Kronlof, K., "UML 2.0 profile for embedded system design," in Design, Automation and Test in Europe, 2005. Proceedings, vol., no., pp.710-715 Vol. 2, 7-11 March 2005 doi: 10.1109/DATE.2005.321

[35] Gomes, L.; Costa, A., "From use cases to system implementation: statechart based co-design," in Formal Methods and Models for Co-Design, 2003. MEMOCODE '03. Proceedings. First ACM and IEEE International Conference on, vol., no., pp.24-33, 24-26 June 2003 doi: 10.1109/MEMCOD.2003.1210083

[36] Shourong, L.; Halang, W.; Zhang, L.; "A component-based UML profile to model embedded real-time systems designed by the MDA approach," in Embedded and Real-Time Computing Systems and Applications, 2005. Proceedings. 11th IEEE International Conference on, vol., no., pp.563-566, 17-19 Aug. 2005 doi: 10.1109/RTCSA.2005.6

- [37] Berthomieu, B., Ribet, P. O., & Vernadat, F. (2004). The tool TINA—construction of abstract state spaces for Petri nets and time Petri nets. *International Journal of Production Research*, 42(14), 2741-2756.
- [38] Cheng, A., Christensen, S., & Mortensen, K. H. (1997). Model checking Coloured Petri Nets-exploiting strongly connected components. *DAIMI Report Series*, 26(519).
- [39] Hanisch, H. M., & Lüder, A. (2000). A signal extension for Petri nets and its use in controller design. *Fundamenta informaticae*, 41(4), 415-431.
- [40] Jensen, K.; "Coloured Petri Nets. Basic Concepts, Analysis Methods and Practical Use" - Volume 1 Basic Concepts. Berlin. Germany.: SpringerVerlag., 1997.
- [41] Kummer, O.; Wienberg, F.; Duvigneau, M.; Cabac, L.; "Renew – User Guide", University of Hamburg, Department for Informatics, Theoretical Foundations Group, Release 2.2, August 28, 2009
- [42] Jensen, K., & Kristensen, L. M. (2015). Colored Petri nets: a graphical language for formal modeling and validation of concurrent systems. *Communications of the ACM*, 58(6), 61-70.
- [43] Barros, J.; Gomes, L., "Teaching Concurrency Through Petri Nets and Model Composition"; *TeaConc'2006 - Workshop on Teaching Concurrency*; Turku; Finland; 27 June 2006; <http://www.uninova.pt/teaconc2006>, ISBN 1-4244-0681-1
- [44] Costa, A.; Gomes, L., "Module Composition within Petri Nets Model-based Development"; *SIES'2007 – 2nd IEEE International Symposium on Industrial Embedded Systems*; 4-6 July 2007; Hotel Costa da Caparica, Lisbon, Portugal
- [45] Hamez, A.; Hillah, L.; Kordon, F.; Linard, A.; Paviot-Adet, E.; Renault, X.; Thierry-Mieg, X.; "New features in CPN-AMI 3: focusing on the analysis of complex distributed systems," *Application of Concurrency to System Design, 2006. ACSD 2006. Sixth International Conference on*, pp.273-275, 28-30 June 2006 doi: 10.1109/ACSD.2006.15
- [46] Starke, P.H.; Hanisch, H.-M., "Analysis of signal/event nets," in *Emerging Technologies and Factory Automation Proceedings, 1997. ETFA '97.*, 1997 6th International Conference on, vol., no., pp.253-257, 9-12 Sep 1997 doi: 10.1109/ETFA.1997.616278
- [47] Gunes, V.; Peter, S.; Givargis, T.; Vahid, F.; A Survey on Concepts, Applications, and Challenges in Cyber-Physical Systems. *KSII Transactions on Internet and Information Systems (TIIS)*, 8.

[48] Khaitan, S. K.; McCalley, J. D.; "Design Techniques and Applications of Cyberphysical Systems: A Survey," in IEEE Systems Journal, vol. 9, no. 2, pp. 350-365, June 2015, doi: 10.1109/JSYST.2014.2322503

[49] Botta, A., De Donato, W., Persico, V., & Pescapé, A. (2016). Integration of cloud computing and internet of things: a survey. *Future Generation Computer Systems*, 56, 684-700.

[50] Baheti, R., & Gill, H. (2011). Cyber-physical systems. The impact of control technology, 12, 161-166.

[51] Lee, E. A. (2008, May). Cyber physical systems: Design challenges. In *Object oriented real-time distributed computing (isorc)*, 2008 11th IEEE international symposium on (pp. 363-369). IEEE.

[52] Monostori, L. (2014). Cyber-physical production systems: Roots, expectations and R&D challenges. *Procedia CIRP*, 17, 9-13.

[53] Lee, E. A. (2007). Computing foundations and practice for cyber-physical systems: A preliminary report. University of California, Berkeley, Tech. Rep. UCB/EECS-2007-72.

[54] Moutinho, F.; Gomes, L.; Ramalho, F.; Figueiredo, J.; Barros, J.; Barbosa, P.; Pais, R.; Costa, A., "Ecore Representation for Extending PNML for Input-Output Place-Transition Nets", IECON'2010 - 36th Annual Conference of the IEEE Industrial Electronics Society, 2010, Phoenix, AZ, USA

[55] Nunes, R.; Gomes, L.; Barros, J., "A Graphical Editor for the Input-Output Place-Transition Petri Net Class"; ETFA'2007 - 12th IEEE Conference on Emerging Technologies and Factory Automation, September 25-28, 2007; Patras, Greece

[56] Gomes, L.; Barros, J., "Automated Code Generation from Petri Nets Based System Specification", 5th WSES/IEEE World Multiconference on Circuits, Systems, Communications & Computers, CICC'2001; 8-15 Julho 2001; Creta, Grécia

[57] Gomes, L.; Rebelo, R.; Barros, J.; Costa, A.; Pais, R., "From Petri net models to C implementation of digital controllers"; ISIE'2010-IEEE International Symposium on Industrial Electronics, 4-7 July 2010, Bari, Italy;

[58] Gomes, L.; Costa, A.; Barros, J.; Lima, P., "From Petri net models to VHDL implementation of digital controllers", IECON'2007 - 33rd Annual Conference of the IEEE Industrial Electronics Society, November 5-8, 2007, The Grand Hotel, Taipei - Taiwan

- [59] Gomes, L.; Lourenço, J., "Rapid Prototyping of Graphical User Interfaces for Petri-Net-Based Controllers", IEEE Transactions on Industrial Electronics, pp. 1806-1813. ISSN 0278-0046; URL: <http://dx.doi.org/10.1109/TIE.2009.2031188>
- [60] Moutinho, F.; Gomes, L., "From models to controllers integrating graphical animation in FPGA through automatic code generation," in Industrial Electronics, 2009. ISIE 2009. IEEE International Symposium on, vol., no., pp.712-717, 5-8 July 2009 doi: 10.1109/ISIE.2009.5218315
- [61] Costa, A.; Gomes, L., "Petri net Splitting Operation within Embedded Systems Co-design"; INDIN'2007 - 5th IEEE International Conference on Industrial Informatics, 23-26 July 2007, Vienna, Austria
- [62] Costa, A.; Gomes, L., "Partitioning of Petri net models amenable for Distributed Execution"; ETFA'2006 - 2006 IEEE International Conference on Emerging Technologies and Factory Automation; 20-22 September 2006, Prague, Czech Republic; IEEE Catalog Number: 06TH8897C;
- [63] Dabney, J. B., & Harman, T. L. (2004). Mastering simulink. Pearson.
- [64] Bolton, W. (2015). Programmable logic controllers. Newnes.
- [65] David, R. (1995). Grafcet: A powerful tool for specification of logic controllers. Control Systems Technology, IEEE Transactions on, 3(3), 253-268.
- [66] Charousset, D.; Hiesgen, R.; Schmidt, T.C; Revisiting actor programming in C++, In Computer Languages, Systems & Structures, Volume 45, 2016, Pages 105-131, ISSN 1477-8424, <https://doi.org/10.1016/j.cl.2016.01.002>.
- [67] MODBUS over serial line specification and implementation guide V1.0, 2002, http://www.modbus.org/docs/Modbus_over_serial_line_V1.pdf (accessed Sep. 2017)
- [68] Feng Xia; Zhi Wang; Youxian Sun, "Towards component-based control system engineering with IEC61499," in Intelligent Control and Automation, 2004. WCICA 2004. Fifth World Congress on, vol.3, no., pp.2711-2715 Vol.3, 15-19 June 2004, doi: 10.1109/WCICA.2004.1342091
- [69] <http://www.iec61499.com> (accessed Sep. 2017)
- [70] Vyatkin, V., & Instrument Society of America. (2007). IEC 61499 function blocks for embedded and distributed control systems design (p. o3neida). ISA-Instrumentation, Systems, and Automation Society.

- [71] Lee, J., Bagheri, B., & Kao, H. A. (2015). A cyber-physical systems architecture for industry 4.0-based manufacturing systems. *Manufacturing Letters*, 3, 18-23.
- [72] Cassandras, C; Lafortune, S, "Introduction to Discrete Event Systems", second edition, 2008, Springer US, ISBN 978-0-387-33332-8 doi: 10.1007/978-0-387-68612-7
- [73] Popova-Zeugmann, L. (2013). Time Petri Nets (pp. 31-137). Springer Berlin Heidelberg.
- [74] Li, Z., & Zhou, M. (2009). Deadlock resolution in automated manufacturing systems: a novel Petri net approach (Vol. 1430, No. 9491). Springer Science & Business Media.
- [75] Peterson, J.L (1981) Petri Net Theory and the Modeling of Systems, Prentice Hall PTR Upper Saddle River, NJ, USA ©1981 ISBN:0136619835
- [76] Yamalidou, K., Moody, J., Lemmon, M., & Antsaklis, P. (1996). Feedback control of Petri nets based on place invariants. *Automatica*, 32(1), 15-28.
- [77] Boucheneb, H., & Hadjidj, R. (2006). CTL* model checking for time Petri nets. *Theoretical Computer Science*, 353(1), 208-227.
- [78] Valmari, A. (1991). Stubborn sets for reduced state space generation. In *Advances in Petri Nets 1990* (pp. 491-515). Springer Berlin Heidelberg.
- [79] Patil, S., Vyatkin, V., & Sorouri, M. (2012, September). Formal verification of intelligent mechatronic systems with decentralized control logic. In *Emerging Technologies & Factory Automation (ETFA), 2012 IEEE 17th Conference on* (pp. 1-7). IEEE
- [80] Grahlmann, B. (1997, January). The PEP tool. In *Computer Aided Verification* (pp. 440-443). Springer Berlin Heidelberg.
- [81] Mäkelä, M. (2002). Maria: Modular reachability analyser for algebraic system nets. In *Application and Theory of Petri Nets 2002* (pp. 434-444). Springer Berlin Heidelberg.
- [82] Schmidt, K. (2000). Lola a low level analyser. In *Application and Theory of Petri Nets 2000* (pp. 465-474). Springer Berlin Heidelberg.
- [83] Gardey, G., Lime, D., & Magnin, M. (2005, January). Romeo: A tool for analyzing time Petri nets. In *Computer Aided Verification* (pp. 418-423). Springer Berlin Heidelberg.

- [84] Halme, J., Hiekkänen, K., & Pyssysalo, T. (1995). PROD reference manual (p. 56). Espoo, Finland: Helsinki University of Technology, Digital Systems Laboratory.
- [85] Heitmann, F; Moldt, D; "Petri Nets Tools Database", accessible on-line at <http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/db.html> (accessed Sep. 2017)
- [86] Starke, P.; Roch, S., "Analysing Signal-Net Systems", Humboldt-Universität zu Berlin, Institut für Informatik, September 2002
- [87] Minas, M.; Frey, G., "Visual PLC-programming using signal interpreted Petri nets," in American Control Conference, 2002. Proceedings of the 2002, vol.6, no., pp.5019-5024 vol.6, 2002, doi: 10.1109/ACC.2002.1025461
- [88] Frey, G. (2003). Hierarchical design of logic controllers using signal interpreted Petri nets. Proceedings of the IFAC AHDS 2003, Saint-Malo (France), 12, 401-406.
- [89] Schumacher, F., Schröck, S., & Fay, A. (2013, June). Transforming hierarchical concepts of GRAFCET into a suitable Petri net formalism. In Manufacturing Modelling, Management, and Control (Vol. 7, No. 1, pp. 295-300).
- [90] Cortadella, J., Kishinevsky, M., Kondratyev, A., Lavagno, L., & Yakovlev, A. (1997). Petrify: a tool for manipulating concurrent specifications and synthesis of asynchronous controllers. IEICE Transactions on information and Systems, 80(3), 315-325.
- [91] Yakovlev, A., Lavagno, L., & Sangiovanni-Vincentelli, A. (1992, November). A unified signal transition graph model for asynchronous control circuit synthesis. In Proceedings of the 1992 IEEE/ACM international conference on Computer-aided design (pp. 104-111). IEEE Computer Society Press.
- [92] L. Gomes, A. Steiger-Garção; 1995; "Programmable controller design based on a synchronized colored Petri net model and integrating fuzzy reasoning"; in Application and Theory of Petri Nets'95; Lecture Notes in Computer Science; vol. 935; Giorgio De Michelis, Michel Diaz(eds.); Springer, Berlin; pp 218-237
- [93] Liu, F., Heiner, M., & Yang, M. (2012, December). An efficient method for unfolding colored Petri nets. In Proceedings of the Winter Simulation Conference (p. 295). Winter Simulation Conference.
- [94] Gomes, L., & Barros, J. P. (2003, September). On structuring mechanisms for Petri nets based system design. In Emerging Technologies and Factory Automation, 2003. Proceedings. ETFA'03. IEEE Conference (Vol. 2, pp. 431-438). IEEE.

- [95] Mortensen, K. H. (2001, August). Efficient data-structures and algorithms for a coloured Petri nets simulator. In Third Workshop and Tutorial on Practical Use of Coloured Petri Nets and the CPN Tools, DAIMI PB-544 (pp. 57-74).
- [96] Travis, J., & Kring, J. (2007). LabVIEW for everyone. Prentice-Hall.
- [97] Gomes, L., & Barros, J. P. (2005). Structuring and composability issues in Petri nets modeling. *Industrial Informatics, IEEE Transactions on*, 1(2), 112-123.
- [98] Rohr, C., Marwan, W., & Heiner, M. (2010). Snoopy—a unifying Petri net framework to investigate biomolecular networks. *Bioinformatics*, 26(7), 974-975.
- [99] Rausch, M., & Hanisch, H. M. (1995, October). Net condition/event systems with multiple condition outputs. In *Emerging Technologies and Factory Automation, 1995. ETFA'95, Proceedings., 1995 INRIA/IEEE Symposium on* (Vol. 1, pp. 592-600). IEEE.
- [100] Mertke, T.; Frey, G., "Formal verification of PLC programs generated from signal interpreted Petri nets," in *Systems, Man, and Cybernetics, 2001 IEEE International Conference on*, vol.4, no., pp.2700-2705 vol.4, 2001 doi: 10.1109/ICSMC.2001.972974
- [101] Bollue, K., Abel, D., & Thomas, W. (2009, August). Synthesis of behavioral controllers for discrete event systems with nces-like petri net models. In *Control Conference (ECC), 2009 European* (pp. 4786-4791). IEEE.
- [102] Bollue, K., Slaats, M., Abrahám, E., Thomas, W., & Abel, D. (2010). Synthesis of behavioral controllers for des: Increasing efficiency. *11th Int'l WODES*, 27-34.
- [103] IEEE Standard VHDL Language Reference Manual," in *IEEE Std 1076-2008* (Revision of IEEE Std 1076-2002), vol., no., pp.c1-626, Jan. 26 2009 doi: 10.1109/IEEESTD.2009.4772740
- [104] <http://www.isagraf.com/index.htm> (accessed Sep. 2017)
- [105] <http://www.eclipse.org/4diac/> (accessed Sep. 2017)
- [106] Yoong, L. H., Shaw, G. D., Roop, P. S., & Salcic, Z. (2012). Synthesizing globally asynchronous locally synchronous systems with IEC 61499. *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, 42(6), 1465-1477.
- [107] Moutinho, F.; Gomes, L., "Asynchronous-Channels Within Petri Net-Based GALS Distributed Embedded Systems Modeling," in *Industrial Informatics, IEEE*

Transactions on, vol.10, no.4, pp.2024-2033, Nov. 2014
doi: 10.1109/TII.2014.2341933

[108] Moutinho, F.; Ribeiro, J.; Gomes, L., "Distributed controllers modeling through Petri nets with multi-asynchronous-channels," in Industrial Technology (ICIT), 2015 IEEE International Conference on, vol., no., pp.1564-1569, 17-19 March 2015
doi: 10.1109/ICIT.2015.7125319

[109] Vyatkin, V., & Hanisch, H. M. (2001, October). Formal modeling and verification in the software engineering framework of IEC 61499: a way to self-verifying systems. In Emerging Technologies and Factory Automation, 2001. Proceedings. 2001 8th IEEE International Conference on (Vol. 2, pp. 113-118). IEEE.

[110] Hanisch, H. M., & Vyatkin, V. (2005, April). Modeling and verification of distributed control systems. In International conference "Design, Analysis, and Simulation of Distributed Systems (DADS'2005)", Proceedings, San Diego.

[111] Vyatkin, V., "Execution Semantic of Function Blocks based on the Model of Net Condition/Event Systems," in Industrial Informatics, 2006 IEEE International Conference on, vol., no., pp.874-879, 16-18 Aug. 2006
doi: 10.1109/INDIN.2006.275692

[112] Li Hsien Yoong; Roop, P.S.; Vyatkin, V.; Salcic, Z., "A Synchronous Approach for IEC 61499 Function Block Implementation," in Computers, IEEE Transactions on, vol.58, no.12, pp.1599-1614, Dec. 2009
doi: 10.1109/TC.2009.128

[113] Li Hsien Yoong; Roop, P.; Vyatkin, V.; Salcic, Z., "Synchronous Execution of IEC 61499 Function Blocks Using Esterel," in Industrial Informatics, 2007 5th IEEE International Conference on, vol.2, no., pp.1189-1194, 23-27 June 2007
doi: 10.1109/INDIN.2007.43849

[114] Dubinin, V.N.; Vyatkin, V., "Semantics-Robust Design Patterns for IEC 61499," in Industrial Informatics, IEEE Transactions on, vol.8, no.2, pp.279-290, May 2012, doi: 10.1109/TII.2012.2186820

[115] Ramesh, S., "Efficient translation of statecharts to hardware circuits," in VLSI Design, 1999. Proceedings. Twelfth International Conference On, vol., no., pp.384-389, 7-10 Jan 1999, doi: 10.1109/ICVD.1999.745186

[116] Steinberg, D., Budinsky, F., Merks, E., & Paternostro, M. (2008). EMF: eclipse modeling framework. Pearson Education.

[117] Bambagini, M.; Di Natale, M., "A code generation framework for distributed real-time embedded systems," in *Emerging Technologies & Factory Automation (ETFA)*, 2012 IEEE 17th Conference on, vol., no., pp.1-10, 17-21 Sept. 2012

doi: 10.1109/ETFA.2012.6489586

[118] Oldevik, J. (2006). MOFScript user guide. Version 0.6 (MOFScript v 1.1. 11).

[119] Jouault, F., & Kurtev, I. (2006, January). Transforming models with ATL. In *satellite events at the MoDELS 2005 Conference* (pp. 128-138). Springer Berlin Heidelberg.

[120] Ličko, M., Schier, J., Tichý, M., & Kühn, M. (2003). Matlab/simulink based methodology for rapid-fpga-prototyping. In *Field Programmable Logic and Application* (pp. 984-987). Springer Berlin Heidelberg.

[121] Harel, D. (1987). Statecharts: A visual formalism for complex systems. *Science of computer programming*, 8(3), 231-274.

[122] Rumbaugh, J., Jacobson, I., & Booch, G. (2004). *Unified Modeling Language Reference Manual*, The. Pearson Higher Education.

[123] Staines, T. S. (2008, March). Intuitive mapping of UML 2 activity diagrams into fundamental modeling concept Petri net diagrams and colored Petri nets. In *Engineering of Computer Based Systems, 2008. ECBS 2008. 15th Annual IEEE International Conference and Workshop on the* (pp. 191-200). IEEE.

[124] Pais, R., Gomes, L., & Barros, J. P. (2011). Towards Statecharts to Input-Output Place Transition Nets Transformations. In *Technological Innovation for Sustainability* (pp. 227-236). Springer Berlin Heidelberg.

[125] Pais, R., Gomes, L., & Barros, J. P. (2011, November). From UML state machines to Petri nets: History attribute translation strategies. In *IECON 2011-37th Annual Conference on IEEE Industrial Electronics Society* (pp. 3776-3781). IEEE.

[126] Barros, J. P., & Gomes, L. (2000). From activity diagrams to class diagrams. In *Workshop Dynamic Behaviour in UML Models: Semantic Questions In conjunction with Third International Conference on UML*, York, UK.

[127] Bernardi, S., Donatelli, S., & Merseguer, J. (2002, July). From UML sequence diagrams and statecharts to analysable petri net models. In *Proceedings of the 3rd international workshop on Software and performance* (pp. 35-45). ACM.

- [128] White, C., Quin, L., & Burman, L. (2001). Mastering XML Premium Edition. Sybex.
- [129] Bray, T. (2014). The JavaScript Object Notation (JSON) Data Interchange Format.
- [130] Biron, P., Malhotra, A., & World Wide Web Consortium. (2004). XML schema part 2: Datatypes. World Wide Web Consortium Recommendation REC-xmlschema-2-20041028.
- [131] Clark, J., & Murata, M. (2001). {Relax NG} Specification.
- [132] Boag, S., Chamberlin, D., Fernández, M. F., Florescu, D., Robie, J., Siméon, J., & Stefanescu, M. (2002). XQuery 1.0: An XML query language.
- [133] Clark, J., & DeRose, S. (1999). XML path language (XPath) version 1.0.
- [134] Clark, J. (1999). Xsl transformations (xslt). World Wide Web Consortium (W3C). URL <http://www.w3.org/TR/xslt> (accessed Sep. 2017)
- [135] Tidwell, D. (2008). Xslt. " O'Reilly Media, Inc."
- [136] Weber, M., & Kindler, E. (2003). The petri net markup language. In Petri Net Technology for Communication-Based Systems (pp. 124-144). Springer Berlin Heidelberg.
- [137] Hillah, L. M., Kindler, E., Kordon, F., Petrucci, L., & Treves, N. (2009). A primer on the Petri Net Markup Language and ISO/IEC 15909-2. Petri Net Newsletter, 76, 9-28.
- [138] Alanen, M., & Porres, I. (2005, September). Model interchange using OMG standards. In Software Engineering and Advanced Applications, 2005. 31st EUROMICRO Conference on (pp. 450-458). IEEE.
- [139] Ackerman, W. B. (1982). Data flow languages. Computer, 2(15), 15-25.
- [140] Micheli, G. D. (1994). Synthesis and optimization of digital circuits. McGraw-Hill Higher Education.
- [141] Williamson, M. C., & Lee, E. (1996, November). Synthesis of parallel hardware implementations from synchronous dataflow graph specifications. In Signals, Systems and Computers, 1996. Conference Record of the Thirtieth Asilomar Conference on (pp. 1340-1343). IEEE.
- [142] Lee, E.A.; Messerschmitt, D.G., "Synchronous data flow," in Proceedings of the IEEE, vol.75, no.9, pp.1235-1245, Sept. 1987
doi: 10.1109/PROC.1987.13876

[143] Benveniste, A., Caspi, P., Edwards, S., Halbwachs, N., Le Guernic, P., & De Simone, R. (2003). The synchronous languages 12 years later. *Proceedings of the IEEE*, 91(1), 64-83.

[144] Halbwachs, N., Caspi, P., Raymond, P., & Pilaud, D. (1991). The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9), 1305-1320.

[145] Benveniste, A., Le Guernic, P., & Jacquemot, C. (1991). Synchronous programming with events and relations: the SIGNAL language and its semantics. *Science of computer programming*, 16(2), 103-149.

[146] Berry, G., & Gonthier, G. (1992). The Esterel synchronous programming language: Design, semantics, implementation. *Science of computer programming*, 19(2), 87-152.

[147] Bhattacharyya, S. S., Murthy, P. K., & Lee, E. A. (1999). Synthesis of embedded software from synchronous dataflow specifications. *Journal of VLSI signal processing systems for signal, image and video technology*, 21(2), 151-166.

[148] Colaço, J. L., Pagano, B., & Pouzet, M. (2005, September). A conservative extension of synchronous data-flow with state machines. In *Proceedings of the 5th ACM international conference on Embedded software* (pp. 173-182). ACM.

[149] Harel D.; Pnueli A., 1989, On the development of reactive systems. In *Logics and models of concurrent systems*, Krzysztof R. Apt (Ed.). *Nato Asi Series F: Computer And Systems Sciences*, Vol. 13. Springer-Verlag New York, Inc., New York, NY, USA 477-498.

[150] Halbwachs, N. (2013). *Synchronous programming of reactive systems* (Vol. 215). Springer Science & Business Media.

[151] Benveniste, A., & Berry, G. (1991). The synchronous approach to reactive and real-time systems. *Proceedings of the IEEE*, 79(9), 1270-1282.

[152] Benveniste, A; Caillaud, B; Guernic, P, "Compositionality in Dataflow Synchronous Languages". *Inf. Comput.* 163, 1 (November 2000), pp 125-171. doi=10.1006/inco.2000.9999, <http://dx.doi.org/10.1006/inco.2000.9999>

[153] Tripakis, S., Bui, D., Geilen, M., Rodiers, B., & Lee, E. A. (2013). Compositionality in synchronous data flow: Modular code generation from hierarchical sdf graphs. *ACM Transactions on Embedded Computing Systems (TECS)*, 12(3), 83.

- [154] Halbwachs, N., Lagnier, F., & Ratel, C. (1992). Programming and verifying real-time systems by means of the synchronous data-flow language LUSTRE. *Software Engineering, IEEE Transactions on*, 18(9), 785-793.
- [155] Geilen, M., Basten, T., & Stuijk, S. (2005, June). Minimising buffer requirements of synchronous dataflow graphs with model checking. In *Proceedings of the 42nd annual Design Automation Conference* (pp. 819-824). ACM.
- [156] Biernacki, D., Colaço, J. L., Hamon, G., & Pouzet, M. (2008, June). Clock-directed modular code generation for synchronous data-flow languages. In *ACM Sigplan Notices* (Vol. 43, No. 7, pp. 121-130). ACM.
- [157] Yoong, L. H., Roop, P. S., Vyatkin, V., & Salcic, Z. (2009). A synchronous approach for IEC 61499 function block implementation. *Computers, IEEE Transactions on*, 58(12), 1599-1614.
- [158] Duma, R., Dobra, P., Abrudean, M., & Dobra, M. (2007, June). Rapid prototyping of control systems using embedded target for TI C2000 DSP. In *Control & Automation, 2007. MED'07. Mediterranean Conference on* (pp. 1-5). IEEE.
- [159] Simulink Verification and Validation Products
<http://www.mathworks.com/products/simverification/> (accessed Sep. 2017)
- [160] Davis II, J., Goel, M., Hylands, C., Kienhuis, B., Lee, E. A., Liu, J., ... & Smyth, N. (1999). Overview of the Ptolemy project (Vol. 99). ERL Technical Report UCB/ERL.
- [161] Kim, K. D., & Kumar, P. R. (2012). Cyber-physical systems: A perspective at the centennial. *Proceedings of the IEEE*, 100(Special Centennial Issue), 1287-1308.
- [162] https://www.nsf.gov/news/special_reports/cyber-physical/ (accessed Sep. 2017)
- [163] https://www.nsf.gov/funding/pgm_summ.jsp?pims_id=503286&org=CISE&from=home (accessed Sep. 2017)
- [164] CyPhERS - Cyber-Physical European Roadmap and Strategy
<http://www.cyphers.eu/> (accessed Sep. 2017)
- [165] <https://www.eurocps.org/> (accessed Sep. 2017)
- [166] <https://ec.europa.eu/programmes/horizon2020/en/h2020-section/smart-cyber-physical-systems> (accessed Sep. 2017)
- [167] Brettel, M., Friederichsen, N., Keller, M., & Rosenberg, M. (2014). How virtualization, decentralization and network building change the manufacturing

landscape: An industry 4.0 perspective. *International Journal of Mechanical, Industrial Science and Engineering*, 8(1), 37-44.

[168] John, K. H., & Tiegelkamp, M. (2010). IEC 61131-3: programming industrial automation systems: concepts and programming languages, requirements for programming systems, decision-making aids. Springer Science & Business Media.

[169] IEC 61131-3:2013 Programmable controllers, Part 3: Programming languages, <https://webstore.iec.ch/publication/4552> (accessed Sep. 2017)

[170] <https://www.xilinx.com/products/silicon-devices/soc.html> (accessed Sep. 2017)

[171] Gamma, R.; Helm, R.; Johnson, R.; Vlissides, J.; Design Patterns: Elements of Reusable Object-Oriented Software

<http://www.awprofessional.com/bookstore/product.asp?isbn=0201633612&rl=1> (accessed Sep. 2017)

[172] Kerberos: The Network Authentication Protocol, <https://web.mit.edu/kerberos/> (accessed Sep. 2017)

[173] Harrison, R.; Lightweight directory access protocol (LDAP): Authentication methods and security mechanisms, 2006

[174] Hill, J.; An analysis of the RADIUS authentication protocol. *InfoGard Laboratories*, 2001

[175] Campos-Rebelo, R.; Costa, A.; Gomes, L.; "Event life time in detection of sequences of events," 2015 IEEE International Conference on Industrial Technology (ICIT), Seville, 2015, pp. 3144-3149; doi: 10.1109/ICIT.2015.7125562

[176] <https://run.unl.pt/handle/10362/19193> (accessed Sep. 2017)

[177] Carrola Rocha, S.A.; "Utilização de Sinais bioelétricos em controladores modelados com redes de Petri IOPT"; MiEEC, FCT/UNL, MSc dissertation on Electrical and Computer Engineering, 2016

[178] Mills, D. L.; Internet time synchronization: the network time protocol. *IEEE Transactions on communications*, 39(10), 1482-1493, 1991.

[179] Deininger, D., Dimitrova, R., Majumdar, R. Symbolic Model Checking for Factored Probabilistic Models. In *International Symposium on Automated Technology for Verification and Analysis* (pp. 444-460). Springer International Publishing, 2016, October.