Petri Net Based Specification and Verification of Globally-Asynchronous-Locally-Synchronous System

Filipe Moutinho[1,2], Luís Gomes[1,2], Paulo Barbosa[3], João Paulo Barros[2,4], Franklin Ramalho[3], Jorge Figueiredo[3], Anikó Costa[1,2], and André Monteiro[3]

[1] Universidade Nova de Lisboa, Faculdade de Ciências e Tecnologia, Portugal
[2] UNINOVA, Portugal
{fcm,lugo,jpb,akc}@uninova.pt
[3] Universidade Federal de Campina Grande, Brazil
{paulo,franklin,abrantes,andre}@dsc.ufcg.edu.br
[4] Instituto Politécnico de Beja, Escola Superior de Tecnologia e Gestão, Portugal

Abstract. This paper shows a methodology for Globally-Asynchronous-Locally-Synchronous (GALS) systems specification and verification. The distributed system is specified by non-autonomous Petri net modules, obtained after the partition of a (global) Petri net model. These modules are represented using IOPT (Input-Output Place-Transition) Petri net models, communicating through dedicated communication channels forming the GALS system under analysis. This set of modules is then automatically translated into Maude code through a MDA approach. As the modules of GALS systems run concurrently, the Maude semantics for concurrent objects is used along with message representation. Finally, as a particular case, the system state space is generated from the Maude specification of the GALS system, allowing property verification.

Keywords: GALS, Embedded Systems, Petri Nets, Maude, Verification.

1 Introduction

Embedded systems are increasingly present in people's lives, for instance in people's pockets, homes, cars and in industrial machinery. Many embedded systems are synchronous systems implemented in a single device, but there are embedded systems that can not be implemented using the synchronous paradigm, either due to the need of having multiple devices in different physical locations, or due to the simple fact that a single device is not enough to implement the system, or even when it is necessary to use devices containing multiple clock domains.

These systems are Globally-Asynchronous-Locally-Synchronous (GALS). They support features like asynchronous messaging and multiple concurrent synchronous modules with different clock domains. But these features make GALS systems development not a simple task, and with greater challenges (for example the verification of GALS systems components interaction) compared to the development of synchronous systems. This was the environment where was found the research

Luis M. Camarinha-Matos (Ed.): DoCEIS 2011, IFIP AICT 349, pp. 237–245, 2011.

question of this work, which is *How to specify, simulate, verify and implement a GALS system described through a set of IOPT Petri net sub-models supported by automatic code generation?*

The methodology proposed here to develop GALS systems for embedded systems applications was initially formulated within the FORDESIGN project [1], which had the objective to center the development effort in the system modeling, relying in a model-base development attitude and taking advantage of automatic code generation tools. However, the FORDESIGN project did not fully consider the development of GALS systems. The chosen modeling formalism was the Input-Output Place-Transition (IOPT) Petri net, a class of non-autonomous Petri nets defined in [2] that extends the well-known Place-Transition (P/T) Petri net class with inputs and outputs signals and events (among other characteristics).

The Net Splitting Operation proposed in [3], allowing model partitioning into several components, is here used to split centralized models of GALS systems into GALS components. These components will be interconnected through lossless communication channels with undetermined propagation time for all the messages.

To allow property verification of the GALS systems, modeled by IOPT net models, are performed a set of transformations from the IOPT net representation to Maude specifications [4]. Those transformations rely on a Model-Driven Architecture (MDA) [5] approach, using IOPT nets and Maude metamodels. The resulting Maude specifications support the verification of several properties.

The reference development methodology fully integrate design automation tools, namely the PNML2C [6] and PNML2VHDL [7] tools, which automatically generate, respectively, C or VHDL code from IOPT net models represented in the PNML format [8].

Section 2 briefly presents the technological contribution of the paper to sustainability. Section 3 presents the proposed methodology with the help of a running example that will be used throughout the paper for an easier understanding of the development flow. Section 4 describes the executable semantics of GALS systems. Section 5 briefly explains the GALS System representation in Maude language, and the running example verification. Section 6 gives an overview of some related work. And finally section 7 presents some final remarks.

2 Contribution to Sustainability

This work aims to contribute to the development of GALS systems in a more automated way relying in the Model-Driven Architecture (MDA) approach. This allows the development of complex systems in less time, while being more reliable and less vulnerable to development bugs, due to the fact that the only development errors that are introduced in the system are the modeling errors; there is no manual code generation, either simulation, verification, or implementation codes. Systems with bugs can cause unwanted effects, as in most cases they need to be replaced, wasting energy and resources.

3 Proposed Methodology

The development flow proposed to GALS systems behavior verification comprises the following steps (described in Fig. 1):

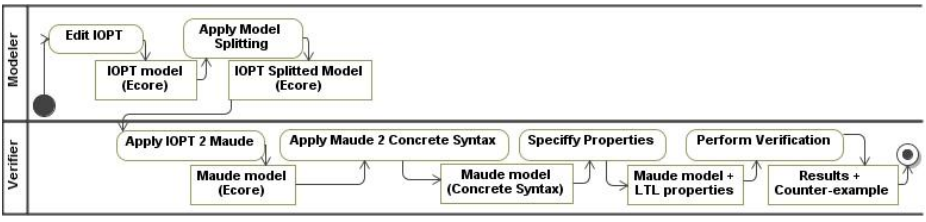

Fig. 1. Activity Diagram to GALS Systems Behavior Verification

- modeler activities: (1) modeling GALS system through IOPT nets; (2) splitting IOPT nets to obtain an IOPT net for each component of the GALS system;
- verifier activities: (3) translation from IOPT net models to Maude models; (4) translation from Maude models to Maude concrete syntax language; (5) specification of system properties to be verified; and (6) properties verification.

3.1 Running Example

The following example will be used through the paper to present the proposed methodology steps. The example is a very simplified condominium alarm system, which is used to detect events and control alarms of buildings. If an event occurs in one of the buildings, their alarm along with their neighbor buildings must ring.

In this example there are three buildings in a row, the building "1" has the neighbor building "2", the building "2" has the neighbor buildings "1" and "3", and building "3" has neighbor building "2".

3.2 System Modeling

This example was modeled by an IOPT net model, and is presented in Fig. 2. As previously mentioned, the IOPT Petri net is a class of non-autonomous Petri nets that extends the Place-Transition (P/T) Petri net class with inputs and outputs. These can be input and outputs signals and also input and output events.

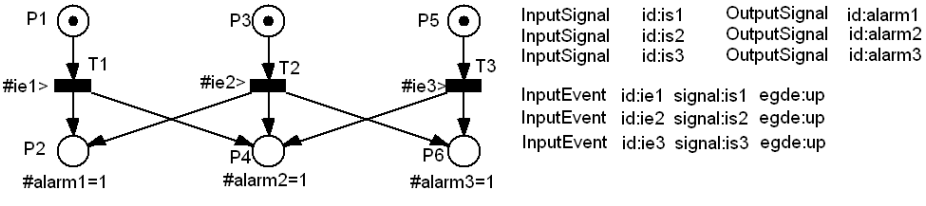

Fig. 2. IOPT model of an oversimplified condominium alarm system

Transition firing depends not only on the net marking, but also on the associated input events as well as the guard expression attached to the transition. Output expressions affecting output signals can be associated with places. When compared to Place-Transition nets, IOPT nets have other specific characteristics, as test arcs and priorities, which are described in [2].

The example has three input events (*ev1*, *ev2*, and *ev3*) associated with transitions governing the evolution of the IOPT net, and three output signals (*alarm1*, *alarm2*, and *alarm3*) that receive the value "1" when the corresponding place has one or more tokens.

3.3 Model Splitting

Considering that the system example will be implemented in a distributed way using three controllers (a controller in each building), the IOPT net model presented in Fig. 2, was divided into the three sub-models presented in Fig. 3, through the Net Splitting Operation.

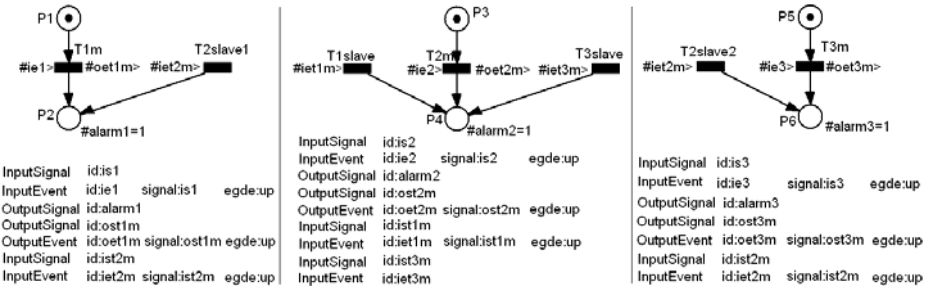

Fig. 3. IOPT sub-models resulting from the Net Splitting Operation

The first step of Net Splitting Operation is the definition of a valid cutting set, which finds a set of nodes with specific characteristics that will be used to divide the original net. The splitting was applied through the nodes *T1*, *T2*, and *T3*, generating the resulting sub-modules interconnected through the transitions *T1m/T1slave*, *T2m/T2slave1/T2slave2*, and *T3m/T3slave*, with associated output events (for instance *oet1m*) in the master transitions, that will be the input events (for instance *iet1m*) of slave transitions in other components. In this sense, the distributed execution model is composed by a set of parallel components communicating through a set of events.

4 Executable Semantics

GALS (Globally Asynchronous Locally Synchronous) systems are composed of several interacting components. Each component is synchronous, which means that its evolution is made at specific instants in time, controlled by a local clock. On the other hand, the global system is asynchronous. As there is no global clock synchronizing the components, each component is evolving at its own clock rate. The interaction between components is made sending messages through communication channels. In this sense, GALS systems have interleaving semantics.

IOPT nets were used in this work to model the whole GALS system, as well as GALS components. The firing of the transitions in one IOPT net (net evolution in one component) is done synchronously at specific instants in time (the synchronized paradigm), normally referred as tics or global clock; this means that, for that

component, between these instances, net marking will not change. The external clock or tic defines the moments in which enabled and ready transitions can fire. The enabled transition concept refers to the net marking dependency, as usual, while the ready transition concept is associated with the non-autonomous attributes evaluation. The IOPT nets have maximal step semantics, which means that all transitions that are enabled and ready at a specific instant in time will fire in that instant.

Fig. 4 presents a GALS system composed by three sub-models (components), each of them modeled with IOPT nets, representing the three components of the running example obtained through the net splitting operation (each of the clouds is associated with one sub-model of Fig. 3). Each sub-model will be potentially associated with a component running on an autonomous platform. The interaction between the various components is modeled through a set of events and accomplished through specific communication channels, for example direct connections, connections via asynchronous wrappers, NoC (network-on-chip), or any other type of networks, as common in distributed systems.

From the IOPT net model viewpoint, the border of each of these components is a set of nodes composed only by transitions. However, as far as the synchronous paradigm can not be applied to the whole system, the events used to assure the communication between components were replaced by places, modeling the separation of time instants associated with the firing of a master transition (emission of the output event from one component) and the firing of a slave transition (reception of the input event by the other component).

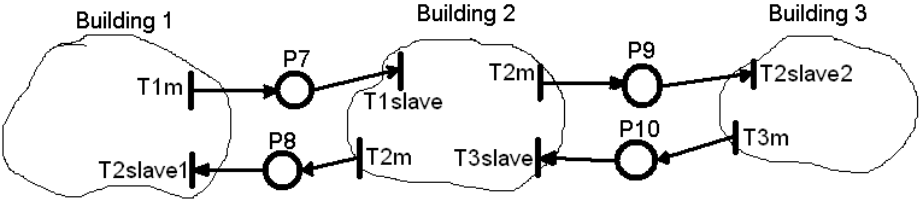

Fig. 4. IOPT nets modeling a GALS system

Each component is an IOPT net model with a maximal step execution, but the evolution tics of one component is different from the evolution tics of the other components, supporting the global asynchrony. In this sense, between components there is an interleaving execution semantics, while each component is governed by a maximal step execution semantics (each component is in a distinct execution temporal domain).

5 From IOPT Models of GALS Systems to Rewriting Logic Objects in Maude

5.1 Maude Language

The Maude language is a declarative language [4]. The basic programming statements are *equations* and *rules*, with simple rewriting semantics. *Rules* can be applied

concurrently, which means that *System modules* can be highly concurrent and non deterministic. In the Maude language it is possible to support objects and distributed objects interactions with rewrite *rules*. Objects interactions can be made through messages. Maude modules can be used: (1) as programs; (2) as executable specification; and (3) as models, that can be verified. In this work these modules will be used as models in which systems properties will be verified.

5.2 MDA Transformations

The MDA approach is used to make the transformation from IOPT net models to the concrete syntax of Maude language. Two transformations were made: (1) model-to-model transformation using the IOPT net metamodel proposed in [9] and Maude metamodel, and (2) model-to-text transformation to obtain Maude code. Model-to-model transformations were achieved using ATL transformation language and model-to-text transformations were made using MOFScript (a tool for model to text transformation).

5.3 GALS System Representation in Maude Language

GALS components modeled by IOPT nets are translated into Maude concurrent objects that interact through asynchronous messages. These messages represent the interleaving semantics of the *Globally Asynchronous* part of the GALS systems.

Maude code of GALS components obtained through the MDA transformation from IOPT net models have interleaving semantics (which is the naturally Maude semantics), although IOPT nets components have a maximal step semantics. This means that the behavior of this Maude code: (1) has exactly the same behavior of the IOPT net model, if and only if, in the each component IOPT net model at most one transition fires at each execution cycle, or (2) has a consistent behavior with the IOPT net model when, the change of, firing at most one transition per execution cycle over several execution cycles, rather than, firing several transitions in just on execution cycle, do not change the GALS components requirements/properties. In the running example, if the transitions *T1m* and *T2slave1* fires in the same execution cycle or if they fire in two consecutive execution cycles, the system requirements remain unchanged.

The generated Maude code for the running example is composed of 2 modules: *PETRI_NET_GALS* and *PETRI_NET_GALS_RULES*, an excerpt of it is presented below. Maude notation is presented in Maude manual in [4].

PETRI_NET_GALS module has the structure of the IOPT nets, in line 2 is made the inclusion of the *CONFIGURATION* module, which declares sorts representing concepts of objects, messages, and configurations that will be needed to represent the three IOPT nets, and the communication between them (represented by *P7*, *P8*, *P9*, and *P10*). In line 16 the three IOPT nets class identifiers are defined.

PETRI_NET_GALS_RULES module contains the transition rules, in line 31 is presented the rule for transition *T1m*, which removes one token from place *P1* and creates one token in *P2*, and one token in *P7*, that represents a message going from component 1 (building 1) to component 2 (building 2).

```
1   mod PETRI_NET_GALS is
2     including CONFIGURATION .
3     sorts LocalTokens GlobalTokens Marking IOPT .
4     ops P1 P2 P3 P4 P5 P6      : -> LocalTokens .
7     ops P7 P8 P9 P10           : -> GlobalTokens .
16    ops Petri1 Petri2 Petri3   : -> Cid [ctor] .
17  endm
18
19  mod PETRI_NET_GALS_RULES is
20    protecting PETRI_NET_GALS .
21    protecting META-LEVEL .
24    var O1 O2 O3                  : Object .
25    vars AP7 AP8 AP9 AP10         : GlobalTokens .
26    var petri                     : Oid .
27    vars AP1 AP2 AP3 AP4 AP5 AP6 : LocalTokens .
29    var S                         : String .
31    rl [T1m] : < petri : Petri1 | m (P1 AP1,    AP2) >,
                 O2, O3, (   AP7, AP8, AP9, AP10), S =>
                   < petri : Petri1 | m (   AP1, P2 AP2) >,
                 O2, O3, (P7 AP7, AP8, AP9, AP10), "T1m " .
53  endm
```

5.4 Verification

As described in Section 3 about the example: (1) if an event occurs in building "1", the alarms of buildings "1" and "2" should begin to ring; (2) if an event occurs in building "2", the alarms of buildings "1", "2", and "3" should begin to ring; and (3) if an event occurs in building "3", the alarms of buildings "2" and "3" should begin to ring. These are the 3 properties that should be verified. Alarm of building "1" rings if the marking of place P1 is equal or greater than 1, and so on.

The generated Maude code, presented in section 5.3, was used in the Maude system to verify the three mentioned properties. To verify property one, it was checked all the possible final states of the system after event "1" occurs. To do this the associated state space containing all reachable states was generated and analyzed. To generate the state space in Maude, the search command *(search < petri1:Oid : Petri1 | m (P1, empty) >, < petri2:Oid : Petri2 | m (P3, empty) >, < petri3:Oid : Petri3 | m (P5, empty) >, (none, none, none, none), "" =>! Any:Net .*) was used in the code presented in section 5.3, and to show it the command (*show search graph .*) was used. It was verified that all possible final states of the system after event "1" occurrence are *(P2=1, P3=1, P4=1, P5=1)*, *(P2=2, P4=2, P5=1, P6=1)*, *(P2=1, P3=1, P4=2, P6=1)* or *(P2=2, P4=3, P6=2)*. Analyzing them, it can be concluded that the places *P2* and *P4* have always one or more tokens, which means that alarms of buildings "1" and "2" ring in this situation, and can be concluded that property one is successfully verified. Properties two and three were also successfully verified.

Due to space limitations, it is not possible to present the complete state space, even for such simple example in order to make evident the referred verified properties. Instead, a simplified system composed by buildings 1 and 2 is considered. Fig. 5 presents the partial state space of this simplified system, considering the sub-models of Fig. 3 and Fig. 4 associated with buildings 1 and 2, which means the models with the transitions *T1m*, *T1slave*, *T2m*, and *T2slave1*, and places *P1*, *P2*, *P3*, *P4*, *P7*, and *P8*. This partial state space has 9 states, presented in Fig. 5, while the full state space

has 45 states. Each node of the state space presented in Fig. 5 shows the marking of the relevant places, where the four nodes with *P7* and *P8* holding no tokens are the ones observed in the initial model of Fig. 2.

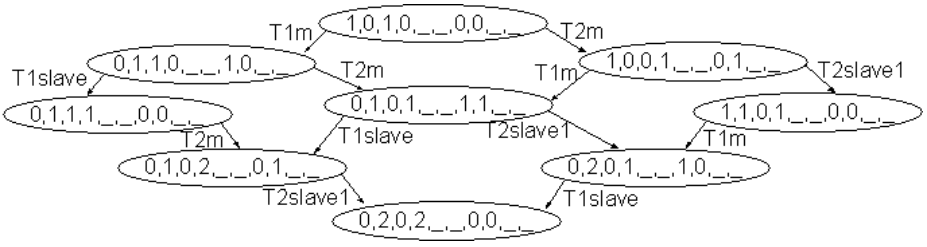

Fig. 5. Partial state space of the example

6 Related Work

Since most of the embedded systems circuitry is made of synchronous circuits, they became the starting point in the development of GALS systems. With this in mind, there are several works proposing architectures, property verification, implementations, and prototyping for GALS systems.

Some authors propose a verification approach for GALS systems (e.g. [10]), taking as starting point the description of the systems' behavior using textual languages, instead of graphical-based descriptions like the ones being proposed in this paper. Other authors use Petri nets to represent GALS systems behavior and to verify its properties (e.g. [11]); however, the methodology does not cover the entire GALS systems development flow, that starts with Petri net models. In [1], a full development flow for embedded systems was proposed through automatic code generation, from Petri net models, but without attempting to answer the specific questions of GALS systems.

To the best of our knowledge, no works addresses the complete development flow (modeling, simulation, verification, and implementation) of GALS systems through automatic code generation based on non-elementary Petri nets.

7 Conclusions

The methodology for specification and verification of GALS systems using IOPT nets as modeling formalism was shown to be adequate in the testing phase of this work. In all the validation examples it was possible to model distributed execution of embedded systems as a GALS system using IOPT nets, and to verify systems properties with Maude. Maude code was always automatically generated from IOPT nets through model-to-model and model-to-text transformations.

The main conclusion is that the proposed development methodology has several advantages compared to a development methodology that does not use models. It also takes advantage from the usage of Petri nets as the underlying model of computation,

namely (1) the model clearly describes the system's behavior; (2) it is possible to start modeling the system with one centralized model, which is then partitioned into a set of modules, the components of the GALS system; and (3) the system verification and implementation codes are automatically generated from the model, which decreases the development time and the potential errors of manual code generation.

Acknowledgments. This work is supported by the cooperation project funded by Portuguese FCT through the project ref. 4.4.1.00-CAPES, and by Brazilian CAPES through the project ref. 236/09. The first author work is supported by a Portuguese FCT (Fundação para a Ciência e a Tecnologia) grant, ref. SFRH/BD/62171/2009.

References

1. Gomes, L., Barros, J.P., Costa, A., Pais, R., Moutinho, F.: Formal Methods for Embedded Systems Co-design: the FORDESIGN Project. In: Proceedings of Workshop Reconfigurable Communication-centric Systems-on-Chip, ReCoSoC (2005)
2. Gomes, L., Barros, J., Costa, A., Nunes, R.: The Input-Output Place-Transition Petri Net Class and Associated Tools. In: Proceedings of the 5th IEEE International Conference on Industrial Informatics (INDIN 2007), Vienna, Austria (2007)
3. Costa, A., Gomes, L.: Petri net partitioning using net splitting operation. In: 7th IEEE International Conference on Industrial Informatics (INDIN 2009), Cardiff, UK (2009), http://dx.doi.org/10.1109/INDIN.2009.5195804
4. Clavel, M., Durán, F., Eker, S., Lincoln, P., Martí-Oliet, N., Meseguer, J., Talcott, C.: Maude Manual (Version 2.5), http://maude.cs.uiuc.edu/maude2-manual/edn
5. OMG-MDA, Omg mda guide version 1.0.1. formal doc.: (June -03- 2001), http://www.omg.org/cgi-bin/doc?omg/03-06-01 (accessed January, 2010)
6. PNML2C: PNML2C - A translator from PNML to C, http://www.uninova.pt/fordesign/PNML2C.htm (accessed March 30, 2010)
7. PNML2VHDL: PNML2VHDL - A translator from PNML to VHDL, http://www.uninova.pt/fordesign/PNML2VHDL.htm (accessed March 30, 2010)
8. Billington, J., Christensen, S., van Hee, K.M., Kindler, E., Kummer, O., Petrucci, L., Post, R., Stehno, C., Weber, M.: The Petri Net Markup Language: Concepts, Technology, and Tools. In: van der Aalst, W.M.P., Best, E. (eds.) ICATPN 2003. LNCS, vol. 2679, pp. 483–505. Springer, Heidelberg (2003)
9. 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. In: 36th Annual Conference of the IEEE Industrial Electronics Society, IECON 2010, Phoenix, AZ, USA, November 7-10 (2010)
10. Doucet, F., Menarini, M., Kruger, I., Gupta, R.: A Verification Approach for GALS Integration of Synchronous Components. (2005), http://www.irisa.fr/prive/talpin/papers/fmgals05a.pdf (accessed July 25, 2010)
11. Dasgupta, S., Yakovlev, A.: Modeling and Performance Analysis of GALS Architectures. In: International Symposium on System-on-Chip 2006, Tampere, Finland (2006)