Chapter 3
Development of Distributed Embedded Controllers

3.1 Proposed Model-Based Development Approach

The MBD approach proposed in this work is presented in this section. This approach supports the development of GALS-DECs (a set of controllers in asynchronous interaction where each controller is synchronously executed). The distributed system is specified through a single Petri net model that simultaneously supports its documentation, validation (using simulation and model-checking tools), and implementation (using automatic code generator tools). This Petri net model is platform-independent, supporting the controller implementation in heterogeneous platforms. Additionally, this model is also network-independent, supporting the interaction through heterogeneous communication networks. Therefore this model provides high flexibility in the implementation phase (to select several types of platforms and communication networks), facilitating the achievement of the desired performance, power consumption, EMI, and cost. The proposed MBD approach is presented in Fig. 3.1 through a UML activity diagram (UML 2015).

The proposed MBD approach comprises several development steps. Each step is in the scope of: the controllers' modeling, the models' simulation, the models' behavioral verification, the controllers' implementation, or the controllers' testing. The development steps are:

- the creation or the selection of the controllers' sub-models. Each controller is specified through one or more Petri net sub-models, each one having synchronous and deterministic execution semantics. These sub-models may be or become reusable. This work proposes the use of Petri nets extended with the time-domain (TD) concept (presented in Sect. 3.4), with priorities (Sect. 3.3), and with inputs and outputs (Sect. 3.2), to create the sub-models of each controller;
- the validation (simulation and verification) of each reusable sub-model. If the sub-model does not present the desired behavior or if it is not possible to create

F. Moutinho, L. Gomes, *Distributed Embedded Controller Development with Petri Nets*, SpringerBriefs in Electrical and Computer Engineering 150,
DOI 10.1007/978-3-319-20822-0_3

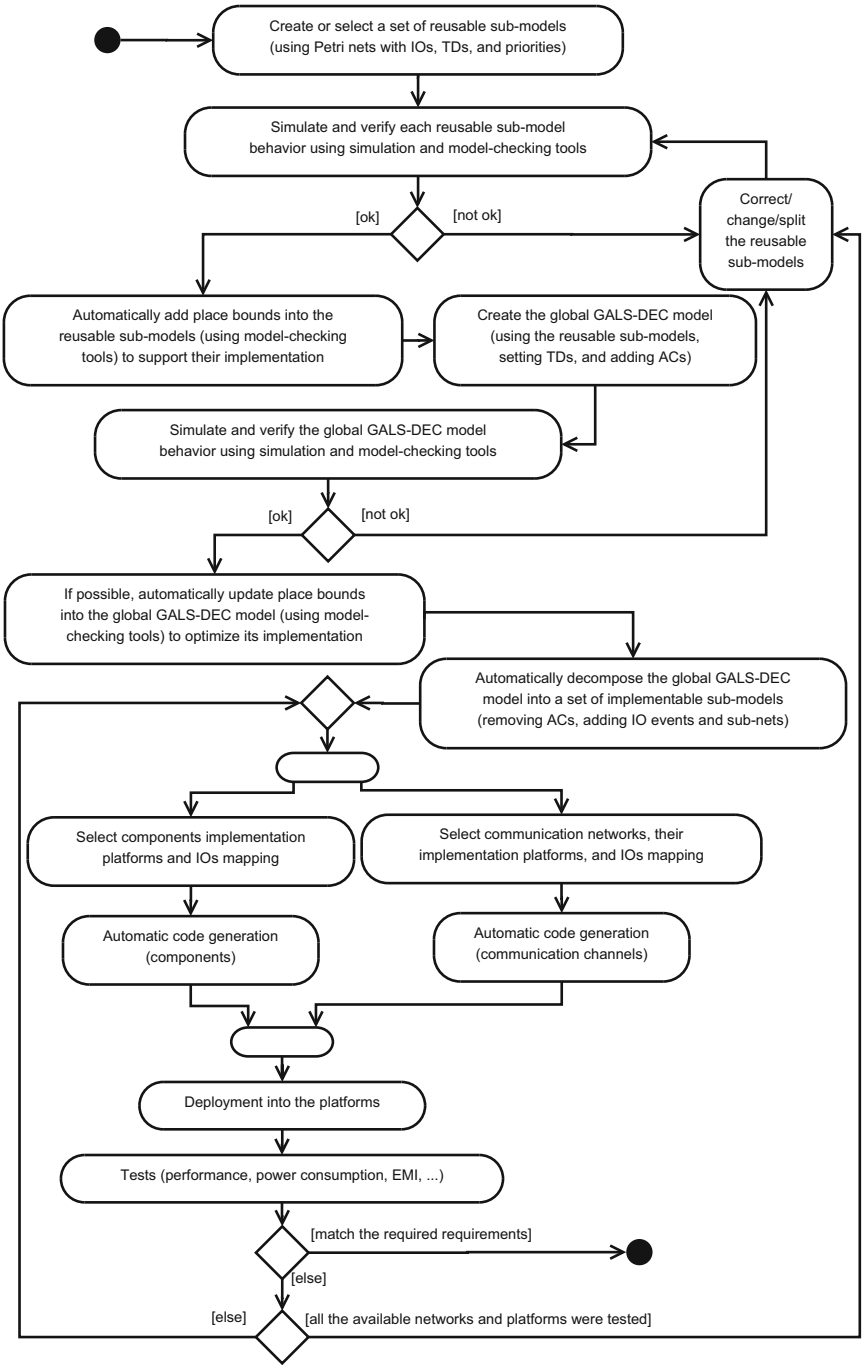

Fig. 3.1 The proposed model-based development approach for GALS-DECs

the full state-space of that sub-model, then the sub-model must be changed, corrected, or split into several sub-models, and then each sub-model must be validated again. It is required to generate the full state-space, to verify the bound of each place (the maximal number of tokens that can be in each place);

- the automatic addition of the place bounds into the reusable sub-models. If it is possible to generate the full state-space, it is possible to calculate the place bounds and automatically add them into the sub-models. The place bounds are the bounds of the memory elements that will be used to implement the places. This is required to ensure that these sub-models can be implemented;
- the creation of the global GALS-DEC model. The global model is created using the reusable sub-models, changing their time-domains (to ensure that sub-models from different components have different time-domains), and connecting asynchronous-channels (presented in Sect. 3.5) to their sources and targets (the transitions that are channel-sources and channel-targets). It is important to note that the reusable sub-models should be created and the asynchronous-channels should be connected, in such a way that it is not possible to simultaneously have more than one message in each asynchronous-channel. This ensures that it is not required to generate the full state-space of the global GALS-DEC model, in order to generate the communication nodes required to support the components interaction (the asynchronous-channels implementation);
- the simulation and the verification of the global GALS-DEC model. If the global model does not specify the desired behavior, the reusable sub-models must be corrected or changed, and the previous steps must be repeated;
- if possible, the automatic update and addition of the place bounds and asynchronous-channel bounds into the global GALS-DEC model. If it is possible to generate the full state-space of the global model, then the place bounds can be automatically updated and the asynchronous-channel bounds can be automatically added. If it is not possible to generate the full state-space, but the global model was created to ensure that the asynchronous-channel bounds are not bigger than one, then place bounds are not updated and the asynchronous-channel bounds can be automatically added and assigned the value one. This bounded model supports the components and the communication nodes implementation;
- the automatic decomposition of the global GALS-DEC model into a set of implementable sub-models. Each of these sub-models will support the implementation of a synchronous component. An algorithm to support this decomposition is presented in Sect. 3.8;
- the selection of the implementation platforms and of the communication networks, and the mapping of the models inputs and outputs, to the platform physical connectors;
- the automatic code generation. The components implementation code can be automatically generated using the tools presented in Campos-Rebelo et al. (2011), Pereira et al. (2012a), and Pereira and Gomes (2013). The communication nodes implementation code can also be automatically generated; however, currently no tools are available to perform this task;

- the code deployment into the implementation platforms;
- the platform tests. If GALS-DEC presents the desired behavior and matches the required requirements (performance, power consumption, and so on), then the distributed controller is finished, otherwise other implementation platforms must be selected and tested, or even other communication networks should be considered. If the available platforms and communication networks were tested, without matching the desired requirements, then the reusable sub-models must be changed, to check if a different distributed controller that also has the desired behavior can match the desired requirements.

Most of these development steps were used in the application example presented in Chap. 4, illustrating the proposed model-based development approach.

3.2 Petri Nets Extended with Inputs and Outputs

To explicitly specify the interaction between the controllers and their environment, Petri nets must integrate input and output dependencies. When these controllers interact through communication channels, the inputs and outputs also support the specification of the interaction between the controllers and the communication channels. The use of three types of inputs and outputs is proposed in this book:

- input signals and output signals;
- input events and output events; and
- channel targets (are inputs) and channel sources (are outputs).

Signals and events must be used in the reusable sub-models, in the global GALS-DEC models, and in the implementable sub-models, to specify the interaction between the controllers and the environment. In the reusable sub-models, channel targets and channel sources are used to specify how the controllers are affected by and affect the communication channels, whereas in the implementable sub-models, events (automatically introduced during the global model decomposition) are used to specify the interaction between the controllers and the communication channels. The channel targets and channel sources, which are associated with transitions, are ignored during: (1) the validation, if the transitions have asynchronous channels connected to them; and (2) the automatic code generation. In this work, such as in other works (like in Gomes et al. 2007a), it is proposed that: (1) input and output signals should be verified and assigned within Boolean expressions and assignment expressions; and (2) input and output events should be associated with transitions. The channel targets and channel sources should also be associated to transitions.

A Petri nets class with these inputs and outputs and associated expressions is given by:

$$\mathrm{PN_{IO}} = (\mathrm{PN}, \mathrm{IO}) = (P, T, F, W, M_0, \mathrm{IO}) \tag{3.1}$$

where PN is a tuple with the common sets to define a Petri nets class [Eq. (2.1)] and IO is given by:

$$\mathrm{IO} = (\mathrm{ie}, \mathrm{oe}, \mathrm{ct}, \mathrm{cs}, \mathrm{is}, \mathrm{os}) \tag{3.2}$$

ie, a partial function associating transitions with sub-sets of input events:

$$\mathrm{ie} : T' \rightarrow \mathscr{P}(\mathrm{IE}) \tag{3.3}$$

where $\mathscr{P}(\mathrm{IE})$ is the power set of IE (the set of all subsets of IE), and IE is the set of input events. This means that a set of input events can be associated with each transition.

oe, a partial function associating transitions with sub-sets of output events:

$$\mathrm{oe} : T' \rightarrow \mathscr{P}(\mathrm{OE}) \tag{3.4}$$

where $\mathscr{P}(\mathrm{OE})$ is the power set of OE, and OE is the set of output events. This means that a set of output events can be associated with each transition.

ct, a partial function identifying some transitions as being channel targets:

$$\mathrm{ct} : T' \rightarrow \mathrm{CT} \tag{3.5}$$

where CT is the set of channel targets. This means that each transition can be target of a communication channel.

cs, a partial function identifying a set of transitions as being channel sources (sources of communication channels):

$$\mathrm{cs} : T' \rightarrow \mathrm{CS} \tag{3.6}$$

where CS is the set of channel targets.

is, a partial function associating transitions with Boolean expressions:

$$\mathrm{is} : T' \rightarrow \mathrm{BE} \tag{3.7}$$

where BE is the set of Boolean expressions checking input signal values.

os, a partial function associating places with assignment expressions:

$$\mathrm{os} : P' \rightarrow \mathscr{P}(\mathrm{AE}) \tag{3.8}$$

where $\mathscr{P}(\mathrm{AE})$ is the power set of AE, and AE is the set of assignment expressions assigning the result of mathematical expressions to output signals.

Inputs constrain the net evolution (the transitions firing), whereas outputs are affected by the net evolution and by the net marking (the number of tokens). An input event that is associated with a transition disables the transition firing whenever it does not occur. Channel targets constrain transitions in a similar way as

input events. A Boolean expression (associated with a transition) disables the transition firing when false. An output event is generated when the associated transition fires. Finally, an output signal is assigned to the result of the associated expression when the associated place is marked.

3.3 Petri Nets with Priorities

Priorities are proposed in this work to solve Petri net conflicts, as in Gomes et al. (2007a). Two (or more) transitions with the same input place are in a structural conflict, which is also an effective conflict if during the net evolution there are states where both transitions are enabled, but cannot fire simultaneously. If two transitions are enabled, but cannot fire simultaneously, which one should fire? To solve this ambiguous situation and allow autonomous execution of the model, different priorities must be assigned to transitions in conflict. This way becomes clear which of the transitions will fire. Priorities simultaneously solve structural conflicts and effective conflicts. A Petri net with a priority function is given by:

$$\mathrm{PN}_P = (\mathrm{PN}, \mathrm{pr}) \tag{3.9}$$

where pr is a partial function associating transitions with positive integers ($\mathbb{N} = \{1, 2, 3, \ldots\}$), given by:

$$\mathrm{pr} : T' \rightarrow \mathbb{N} \tag{3.10}$$

The transition associated with the lower value is the one with higher priority. The priority function must ensure that any two transitions in a structural conflict must have different priorities:

$$\forall_{(p_1 \times t_1),(p_1 \times t_2) \in F} (t_1 \in T \wedge t_2 \in T \wedge t_1 \neq t_2 \Rightarrow \mathrm{pr}(t_1) \neq \mathrm{pr}(t_2)) \tag{3.11}$$

A Petri net model with one solved conflict is presented in Fig. 3.2. Transitions "T1" and "T2" are in conflict, competing for the token that is in place "P1". This conflict is solved assigning priority 1 ("pr:1") to transition "T2" and priority 2 ("pr:2") to transition "T1". This means that transition "T2" has higher priority than transition "T1".

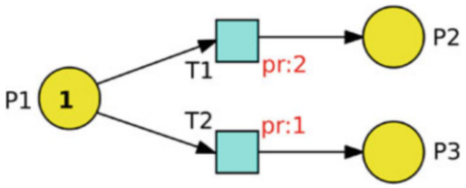

Fig. 3.2 A Petri net model with one conflict solved through priorities

3.4 The Time-Domain Concept

The time-domain concept, described in Moutinho and Gomes (2014), introduces the globally-asynchronous locally-synchronous execution semantics into Petri nets, and ensures that the created models always specify distributed systems, supporting their implementation, as desired in this work. Time-domains make Petri nets totally synchronized Petri nets with single-server semantics (such as those proposed in Moalla et al. 1978). The totally synchronized Petri nets presented in Moalla et al. (1978) are suited to model GALS systems; however, they do not ensure that the created models can be implemented as distributed systems, whereas the use of Petri nets extended with time-domains ensures that the created models have well-delimited synchronized domains, supporting their implementation as distributed controllers.

3.4.1 Petri Nets Extended with Time-Domains

A Petri nets class extended with time-domains is given by:

$$\mathrm{PN}_{\mathrm{TD}} = (\mathrm{PN}, \mathrm{td}) = (P, T, F, W, M_0, \mathrm{td}) \tag{3.12}$$

where *td* is the time-domain function. *td* is a function associating Petri net places and transitions with positive integers ($\mathbb{N} = \{1, 2, 3, \ldots\}$), as defined in Eq. (3.13).

$$\mathrm{td} : (P \cup T) \rightarrow \mathbb{N} \tag{3.13}$$

To ensure that each sub-model cannot specify more than one component, in a Petri net model with time-domains each arc always connects two nodes (places and transitions) with the same time-domain, as defined in Eq. (3.14).

$$\forall_{(n_1 \times n_2) \in F} (\mathrm{td}(n_1) = \mathrm{td}(n_2)) \tag{3.14}$$

This ensures that the created models are structurally unambiguous and distributable, in order to support their implementation, and the use of automatic code generators. For instance, using Petri nets with time-domains, it is not possible to create models: (1) with structural ambiguities, such as the one presented in Fig. 2.7; and (2) that are not distributable, such as those that have transitions in conflict with different time-domains (conflicts must be solved locally).

A Petri net model with time-domains specifying three synchronous and independent components is presented in Fig. 3.3. This model has four (disconnected) sub-models: the sub-model with time-domain 1, where all nodes have time-domain 1 (“td:1”); the sub-models with time-domain 2, where all nodes have time-domain 2 (“td:2”); and the sub-model with time-domain 3, where all nodes have time-domain 3 (“td:3”).

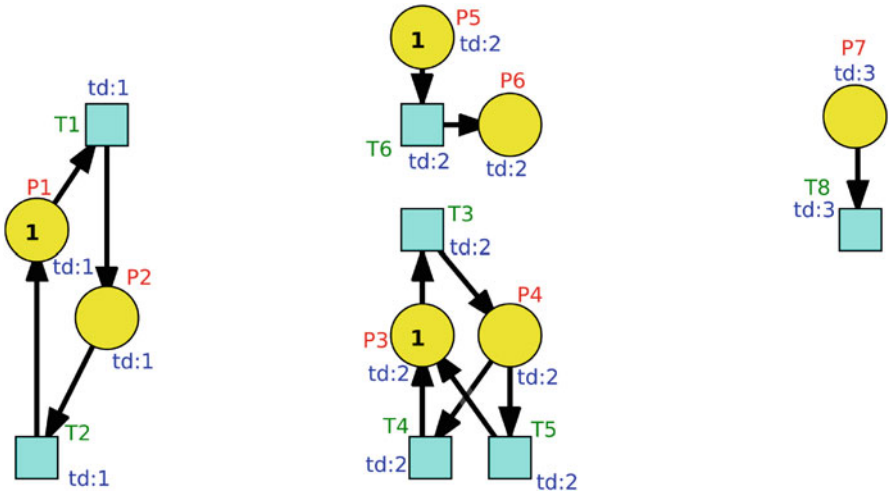

Fig. 3.3 A Petri net model with four sub-models specifying three components

3.4.2 Execution Semantics of Petri Nets with Time-Domains

Petri nets with time-domains have the execution semantics of totally synchronized Petri nets with single-server semantics (Moalla et al. 1978). In a Petri net model with time-domains all transitions have time-domain and all transitions with the same time-domain are synchronized by the same external event, which is implicit for that time-domain. In a specific execution state, when a synchronizing event occurs, all the associated transitions that are enabled and not in a conflict that prevent their firing will fire simultaneously in that instant. It was defined that transitions with different time-domains never fire simultaneously (as they have different synchronizing events). The Petri net model with time-domains presented in Fig. 3.3 has the following execution semantics:

- in the sub-model with time-domain 1, only transition "T1" is enabled, and it will fire when the associated (implicit) event occurs;
- in the sub-model with time-domain 2, both transitions "T6" and "T3" are enabled. They fire simultaneously when the associated (implicit) event occurs;
- in the sub-model with time-domain 3, no transition is enabled;
- in the initial state two things can happen: transition "T1" fires or transitions "T6" and "T3" fire. This shows that the behavior of the global distributed model is non-deterministic (as desired), because each sub-model is independent. However, the behavior of each sub-model is deterministic (in a specific state for specific input values, the sub-model has always the same next state), if the existing conflicts are solved.

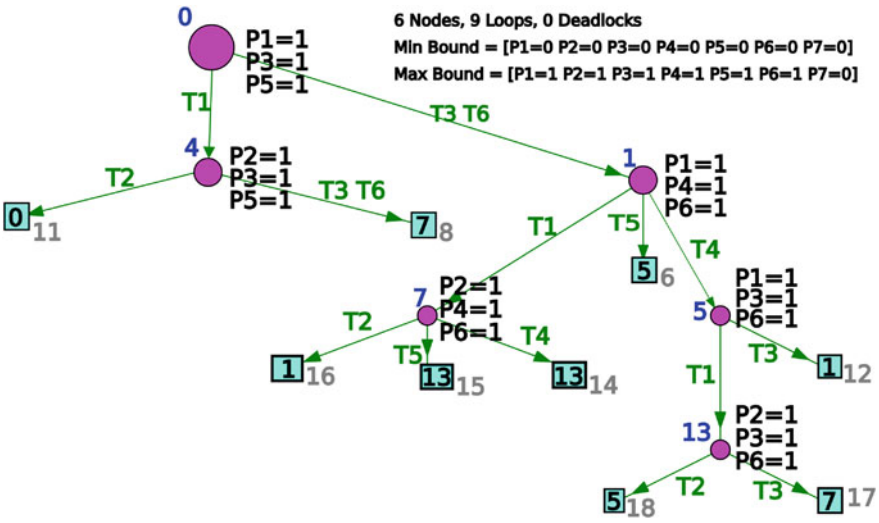

Fig. 3.4 The state-space of the Petri net model from Fig. 3.3

Transitions "T4" and "T5" are in conflict, which means that this conflict must be a-priori solved to ensure deterministic and unambiguous sub-models. As previously mentioned, this book proposes the use of priorities to solve conflicts. The state-space (also known as reachability graph) that represents the global model (Fig. 3.3) behavior is presented in Fig. 3.4.

3.5 Asynchronous-Channels

3.5.1 Introduction

Three types of asynchronous communication channels were proposed in Moutinho and Gomes (2014) to specify the interaction between Petri net sub-models with time-domains, enabling the specification of globally-asynchronous locally-synchronous distributed embedded controllers (GALS-DECs). The use of time-domains ensures that the models have well-delimited synchronized domains without structural ambiguities. However, it is required to enable sub-models interaction, to support the specification among the synchronous components. To support this interaction the following channels were proposed:

- the Simple Asynchronous Channel (SimpleAC);
- the Acknowledged Asynchronous Channel (AckAC);
- the Not-enabled Asynchronous Channel (NotAC).

These three asynchronous-channels provide a network-independent specification of the components interaction, as they do not specify the transmission time, which can be unbounded, between zero and infinite (a communication failure). This means that a global model (with these channels) can support the implementation using different types of communication networks and protocols. Additionally, the validation of this global model provides results that are valid regardless of the implementation support. This provides high flexibility in the implementation phase, enabling the creation of several heterogeneous prototypes (using a single global model), test them, and select the most suited one (for instance, the one that provides the desired performance with lower power consumption).

Each asynchronous-channel is listening one transition of one sub-model (with a specific time-domain) and based on that sends messages to a set of transitions of another sub-model (with another time-domain). The SimpleAC, which is an improved version of the channel introduced in Moutinho and Gomes (2012a), sends a message to the target sub-model whenever the listened transition (the source transition) fires. The AckAC sends a message to the target sub-model whenever the listened transition receives a message from another asynchronous-channel. The NotAC sends a message to the target sub-model whenever the listened transition receives a message from another asynchronous-channel and does not fire (reporting that the transition is not enabled).

When a message arrives the target sub-model, it is simultaneously delivered to the target transitions of that asynchronous channel. From those transitions, the ones that can fire, will fire in the next execution step. The message is only available (to be read) during one execution step, being destroyed after that.

A Petri net model with these three types of channels is presented in Fig. 3.5. This model has three SimpleACs ("AC1", "AC3", and "AC5"), one AckAC ("AC2"), and one NotAC ("AC4"):

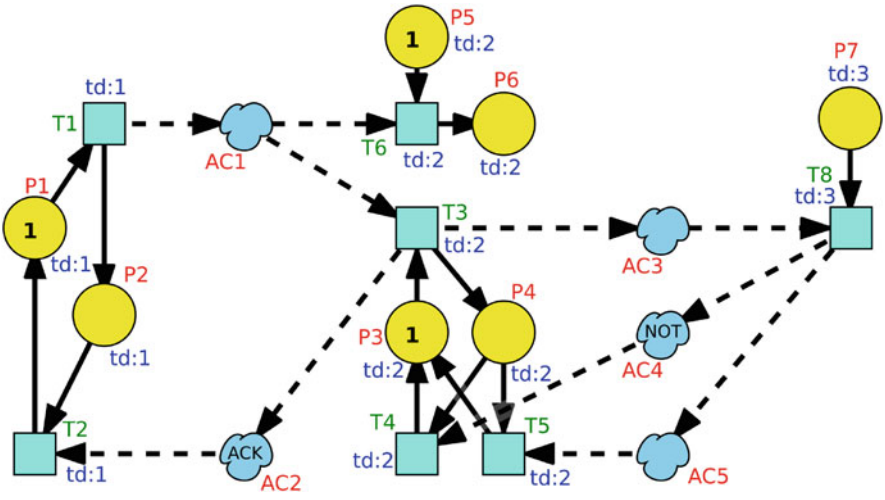

Fig. 3.5 A Petri net model with three SimpleACs, one AckAC, and one NotAC

- "AC1" connects transition "T1" of the sub-model with time-domain 1 ("td:1") to the transitions "T6" and "T3" of the sub-model with time-domain 2 ("td:2");
- "AC3" connects transition "T3" of the sub-model with time-domain 2 to the transition "T8" of the sub-model with time-domain 3 ("td:3");
- "AC5" connects transition "T8" of the sub-model with time-domain 3 to the transition "T5" of the sub-model with time-domain 2;
- "AC2" connects transition "T3" of the sub-model with time-domain 2 to the transition "T2" of the sub-model with time-domain 1;
- "AC4" connects transition "T8" of the sub-model with time-domain 3 to the transition "T4" of the sub-model with time-domain 2.

Whenever transition "T1" fires, one message is created and sent through the asynchronous-channel "AC1", to the transitions "T6" and "T3". Whenever transition "T3" receives a message (regardless of its firing), one message is created and sent through the asynchronous-channel "AC2", to the transition "T2". Finally, whenever transition "T8" receives a message and does not fire, one message is created and sent through the asynchronous-channel "AC4", to the transition "T4".

3.5.2 Asynchronous-Channel Definition

A Petri nets class extended with asynchronous-channels and time-domains is given by:

$$\mathrm{PN_{AC}} = (\mathrm{PN_{TD}}, \mathrm{AC}) = (P, T, F, W, M_0, \mathrm{td}, \mathrm{AC}) \tag{3.15}$$

where AC, a set of asynchronous-channels that includes a set of SimpleACs (SAC), a set of AckACs (AAC), and a set of NotACs (NAC), is given by Eq. (3.16).

$$\mathrm{AC} = (\mathrm{SAC} \cup \mathrm{AAC} \cup \mathrm{NAC}) \tag{3.16}$$

SimpleACs, AckACs, and NotACs associate transitions with sets of transitions, as presented in Eqs. (3.17)–(3.19), where $\mathscr{P}(T)$ is the power set of T.

$$\mathrm{SAC} \subseteq T \times \mathscr{P}(T) \tag{3.17}$$

$$\mathrm{AAC} \subseteq T \times \mathscr{P}(T) \tag{3.18}$$

$$\mathrm{NAC} \subseteq T \times \mathscr{P}(T) \tag{3.19}$$

Each asynchronous-channel connects one transition of one component (the source transition) to a set of transitions of another component (the target transitions). This means that the target transitions of each asynchronous-channel must have the same time-domain (as they belong to a single component), as presented in Eq. (3.20).

$$\forall_{t_1,t_2 \in T_a} : (t, T_a) \in \text{AC} \Rightarrow \text{td}(t_1) = \text{td}(t_2)) \tag{3.20}$$

Two asynchronous-channels cannot have the same target transition:

$$\forall_{t \in T_a} \nexists_{t \in T_b} : (t_1, T_a) \in \text{AC} \wedge (t_2, T_b) \in \text{AC} \wedge t_1 \neq t_2 \tag{3.21}$$

The AckACs and NotACs are used to provide feedback about the delivery of messages and about their influence in the target transitions. This means that: (1) the source of an AckAC is always the target of another asynchronous-channel [Eq. 3.22], and (2) the source of a NotAC is always the target of another asynchronous-channel [Eq. 3.23].

$$\forall_{(t_s,T_a) \in \text{AAC}} \exists_{(t,T_b) \in \text{AC}} : t_s \in T_b \tag{3.22}$$

$$\forall_{(t_s,T_a) \in \text{NAC}} \exists_{(t,T_b) \in \text{AC}} : t_s \in T_b \tag{3.23}$$

3.5.3 Asynchronous-Channels Execution Semantics

Asynchronous-channels were proposed to connect sub-models with different time-domains, specifying the asynchronous interaction among distributed and synchronous components. Each channel specifies the sending of a specific message from one component (the source) to another component (the target). These channels do not specify the communication network, protocol, and delay (the time taken by each message from the source to the target), ensuring network-independent specifications.

These three types of channels have similar execution semantics. The difference is that they report different events in the source components:

- the SimpleAC sends a message whenever its source transition fires;
- the AckAC sends a message whenever its source transition fires receives a message;
- the NotAC sends a message whenever its source transition fires receives a message but does not fire (because it is disabled).

The execution semantics of a Petri net model with asynchronous-channels (such as the one presented in Fig. 3.5) can be expressed through a Petri net model where the asynchronous-channels were replaced by behaviorally equivalent sub-models (such as the one presented in Fig. 3.6). In the model from Fig. 3.6, the asynchronous-channels "AC1", "AC2", "AC3", "AC4", and "AC5" from Fig. 3.5 were replaced their behaviorally equivalent sub-models (with different coloring nodes).

The SimpleACs, the AckACs, and the NotACs behaviorally equivalent Petri net sub-models are presented in Figs. 3.7, 3.8, and 3.9. The algorithm presented in Sect. 3.6 supports the transformation of Petri net models with asynchronous-channels into Petri net models without asynchronous-channels. In all the behaviorally equivalent models:

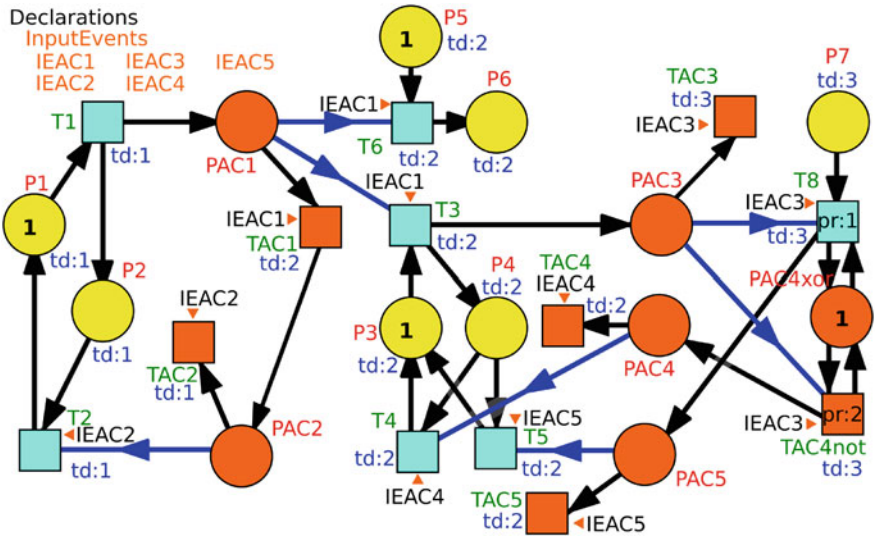

Fig. 3.6 The model that specifies the execution semantics of the model from Fig. 3.5, but without asynchronous-channels

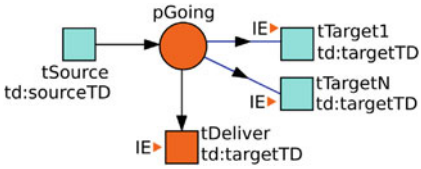

Fig. 3.7 The SimpleAC behaviorally equivalent Petri net sub-model

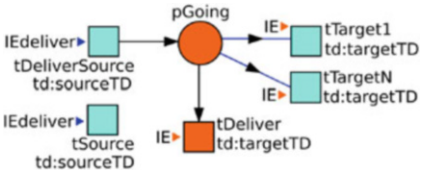

Fig. 3.8 The AckAC behaviorally equivalent Petri net sub-model

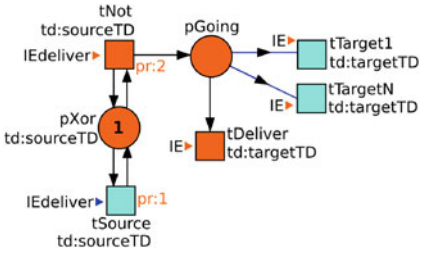

Fig. 3.9 The NotAC behaviorally equivalent Petri net sub-model

- the place *pGoing* is used to count the number of messages that were sent and that have not yet arrived into the target component;
- the transition *tDeliver* firing specifies the arriving of a message. It fires when the associated event (*IE*) occurs. When the transition *tDeliver* fires, the target transitions if enabled also fire. *tDeliver* consumes the tokens from place *pGoing*, ensuring that each message is only available (to enable the target transitions) during one execution step of the target component, being destroyed after that;
- the input event *IE* is non-deterministic, ensuring that these channels do not specify the communication time, as desired to obtain network-independent specifications.

Any of these behaviorally equivalent models has two target transitions ("tTarget1" and "tTargetN"); however, they can have one or more target transitions. Each asynchronous-channel can have a non-zero positive integer number of target transitions.

The models from Figs. 3.7, 3.8, and 3.9 have similar execution semantics; however, they react (create and send messages) to different types of events in the source transitions:

- the SimpleAC creates and sends a message whenever the source transition fires, as specified in Fig. 3.7 (when the source transition fires one token is added to place "pGoing" specifying that a message is going to the target component);
- the AckAC creates and sends a message whenever the source transition receives a message from another asynchronous-channel. When the source transition receives a message, the transition "tDeliverSource" (Fig. 3.8) also receives a message. Given that "tDeliverSource" is enabled (it is always enabled), it fires and one token is created in place "pGoing" specifying that a message is going to the target component;
- the NotAC creates and sends a message whenever the source transition receives a message from another asynchronous-channel and does not fire. When the source transition receives a message, the transition "tNot" (Fig. 3.8) also receives a message. The transition "tNot" fires if and only if the source transition does not fire ("tNot" has lower priority than the source transition and "pXor" ensures that they cannot fire simultaneously). When "tNot" fires one token is generated in place "pGoing" specifying that a message is going to the target component.

The model presented in Fig. 3.6 supports the validation of the model from Fig. 3.5, but not its implementation or documentation. It does not support its implementation because not all nodes have time-domain and the arcs do not always connect nodes with equal time-domain. However, the use of Petri nets extended with time-domains without fulfilling all the assumptions described in Sect. 3.4 is not a problem if the models are only used for validation purposes.

3.6 Distributed GALS Models Validation

Petri net models with time-domains and asynchronous-channels can be simulated (using simulation tools) and verified (using state-space model-checking tools). An algorithm, which specifies the translation of Petri net models with time-domains and asynchronous-channels into behaviorally equivalent models with time-domains but without asynchronous-channels, is presented in this section. This algorithm was implemented in the IOPT-Tools online framework (Pereira et al. 2012a), to allow the use of their simulation and model-checking tools to simulate and verify distributed GALS models. The translation algorithm is presented in Algorithm 1 and described in the following items:

- line 1—the Petri net model ("globalPNname") with asynchronous-channels is copied into the "globalPN" data structure;
- line 2—the "globalPN" is cloned into the data structure ("translatedPN") that will have the translated model;
- line 3—for each asynchronous-channel of the "translatedPN" data structure:
- lines 4, 5, and 6—it is added the place "pGoing," the transition "tDeliver," and an arc connecting them;
- lines 7 to 10—it is also added a test arc connecting the place "pGoing" to each target transition, where it is also associated an input event;
- lines 11 to 13—if it is a SimpleAC, an arc connecting the source transition to the place "pGoing" is added into the "translatedPN" data structure;
- lines 14 to 16—if it is an AckAC, an arc connecting the transition "tDeliver" of the source component to the place "pGoing" is added into the "translatedPN" data structure;
- lines 17 to 27—if it is a NotAC, the following items are added to the "translatedPN" data structure: the transition "tNot", the place "pXor", arcs interconnecting them and connecting "pXor" to the source transition, priorities to the source transition and to "tNot", an arc connecting "tNot" to "pGoing", and a test arc connecting the place "pGoing" of the behaviorally equivalent sub-model of the asynchronous-channel that is source of the current source transition;
- line 28—the asynchronous-channel is removed from "translatedPN";
- line 30—the obtained model is saved into a PNML file.

The algorithm that describes how to generate the state-spaces (also known as reachability graphs) of Petri net models with time-domains was proposed in Moutinho and Gomes (2011) and refined in Moutinho (2014). This algorithm, which was implemented in the IOPT-Tools, generates the state-spaces and saves them into hierarchical XML files. To analyze and verify the state-spaces (searching proprieties), standard tools for XML, like XPath and XQuery (W3C 2013) and the IOPT query engine (Pereira et al. 2012a) can be used. The state-spaces support not only the behavioral verification, but can also provide the places bounds, which are the sizes of the memory resources needed to implement the controllers.

Algorithm 1 The translation algorithm that replaces asynchronous-channels by their behaviorally equivalent sub-models

Require: *globalPNname*
1: *globalPN* ← *Read*(*globalPNname*)
2: *translatedPN* ← *globalPN*
3: **for all** *ac* ∈ *translatedPN.AC* **do**
4: *translatedPN.AddNewPlace*(*pGoing*, *ac.id*)
5: *translatedPN.AddNewTransition*(*tDeliver*, *ac.id*, *translatedPN.td*(*ac.Targets*[0]), *IE*)
6: *translatedPN.AddNewArc*(*pGoing*, *tDeliver*, *ac.id*)
7: **for all** *tTarget* ∈ *ac.Targets* **do**
8: *translatedPN.AddNewTestArc*(*pGoing*, *tTarget*, *ac.id*)
9: *translatedPN.AddEventToTransition*(*tTarget*, *IE*, *ac.id*)
10: **end for**
11: **if** *ac* ∈ *globalPN.SAC* **then**
12: *translatedPN.AddNewArc*(*ac.Source*, *pGoing*, *ac.id*)
13: **end if**
14: **if** *ac* ∈ *globalPN.AAC* **then**
15: *translatedPN.AddNewArc*(*tDeliver*, *ac.Source*, *pGoing*, *ac.id*)
16: **end if**
17: **if** *ac* ∈ *globalPN.NAC* **then**
18: *translatedPN.AddNewTransition*(*tNot*, *ac.id*, *IEdeliver*, *ac.Source*)
19: *translatedPN.AddNewPlace*(*pXor*, *marking* = 1, *ac.id*)
20: *translatedPN.AddNewArc*(*tNot*, *pXor*, *ac.id*)
21: *translatedPN.AddNewArc*(*pXor*, *tNot*, *ac.id*)
22: *translatedPN.AddNewArc*(*ac.Source*, *pXor*, *ac.id*)
23: *translatedPN.AddNewArc*(*pXor*, *ac.Source*, *ac.id*)
24: *translatedPN.AddNewPriorityHigherLower*(*ac.Source*, *tNot*, *ac.id*)
25: *translatedPN.AddNewArc*(*tNot*, *pGoing*, *ac.id*)
26: *translatedPN.AddNewTestArc*(*pGoing*, *ac.Source*, *tNot*, *ac.id*)
27: **end if**
28: *translatedPN.RemoveAC*(*ac*)
29: **end for**
30: *SaveNewPNML*(*translatedPN*)

The state-space of the model from Fig. 3.5, which of course is also the state-space of the behaviorally equivalent model presented in Fig. 3.6, is presented in Fig. 3.10. This state-space was generated in the IOPT-Tools state-space generator for GALS (Moutinho and Gomes 2012b), which implements the algorithm proposed in Moutinho and Gomes (2011) and refined in Moutinho (2014).

The model from Fig. 3.5 has the following behavior:

- whenever transition "T1" fires, one message is sent through the asynchronous-channel "AC1" (in the behaviorally equivalent model one token is inserted in place "PAC1");
- when the "AC1" message arrives the target component (in the behaviorally equivalent model, the event "IEAC1" occurs), one message is sent through the asynchronous-channel "AC2" (in the behaviorally equivalent model one token is inserted in place "PAC2"), and transitions "T6" and "T3" fire (if enabled);

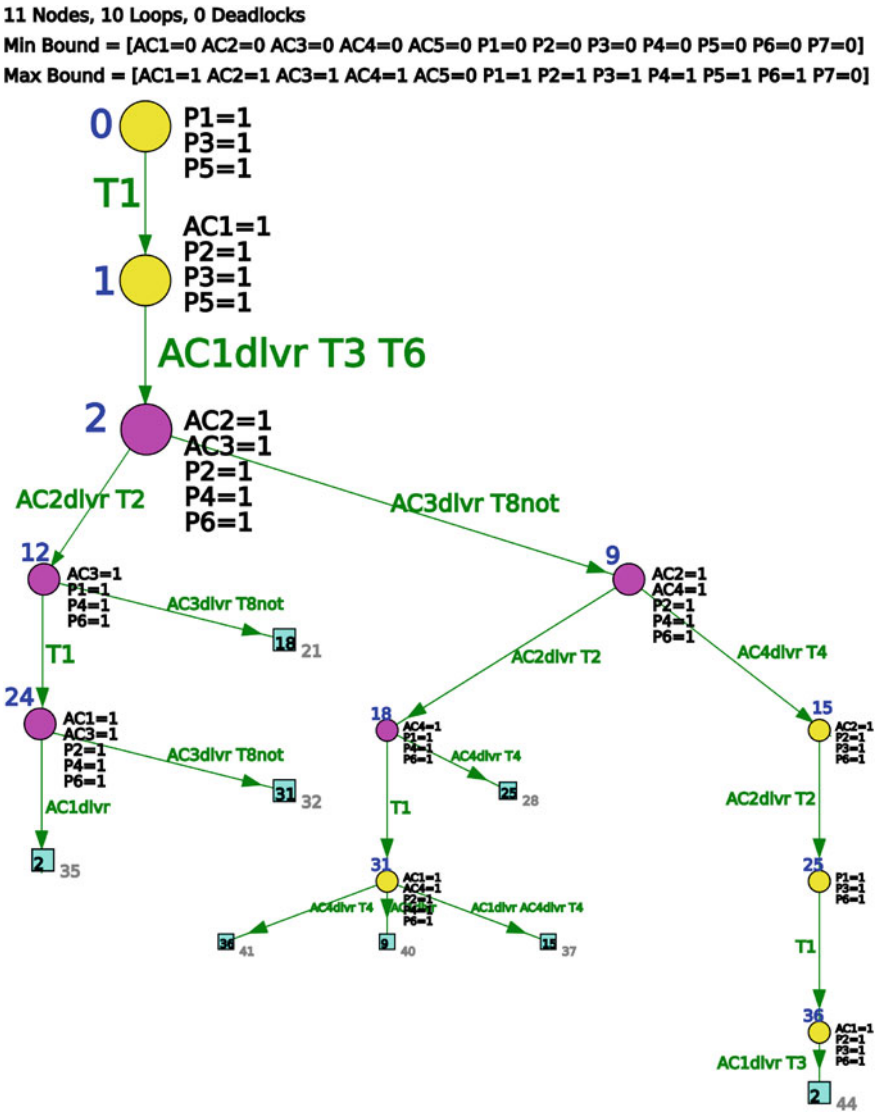

Fig. 3.10 The state-space (reachability graph) of the models from Figs. 3.5 and 3.6

- when "AC2" message arrives to the target sub-model (in the behaviorally equivalent model, this is specified by the occurrence of event "IEAC2"), the transition "T2" fires (if enabled);
- when transition "T3" fires, one message is sent through the asynchronous-channel "AC3";

- when the "AC3" message arrives to the target component (in the behaviorally equivalent model, the event "IEAC3" occurs), one message is sent through the asynchronous-channel "AC4" (in the behaviorally equivalent model one token is inserted in place "PAC4") because transition "T8" is disabled;
- however, if transition "T8" was enabled, then a message would be sent through the asynchronous-channel "AC5" instead of "AC4".

3.7 Bounded Petri Nets

The state-space analysis not only supports the controllers' verification, but also supports their implementation. Each Petri net place will be implemented as a memory resource (such as a software variable or a hardware register). To select the variable type or to implement the register, it is required to know its size, which is given by the place bound. The bound of a place is the maximal number of tokens that will be simultaneously in that place. The bound of each place can easily be checked in the state-space. Figure 3.10 not only presents the state-space, but also the bounds of places and asynchronous-channels.

The model-based development approach (MBD) proposed in Sect. 3.1 includes steps where the bounds are calculated and updated. After the second step (the reusable sub-models' verification) of proposed MBD approach the places bound are added into the sub-models. Later, after the verification of the global GALS-DEC model (where the state-space is generated), the bounds are updated, but only if it is possible to generate the full state-space. Otherwise the bounds added in the second step will remain the same. Given that the global GALS-DEC model was created to have asynchronous-channels bounded to one, it is not required to generate the full state-space to ensure its proper implementation. This is the major difference between the MBD approach proposed in this book and the MBD approach proposed in Moutinho (2014).

The places and the asynchronous-channels bounds are given by Eq. (3.24).

$$\forall_{p\in(P\cup\mathrm{PAC})}(\mathrm{bound}(p) = \max(\forall_{m\in[0..n]}(\#M_m(p)))) \tag{3.24}$$

where:

- P is the set of places that excludes PAC;
- PAC is the set of places of the behaviorally equivalent sub-models that specify the asynchronous-channels;
- $n + 1$ is the number of state-space nodes;
- m is the order of a state-space node;
- $\#M_m(p)$ is the number of tokens that are in the place p in the node m of the state-space.

A bounded Petri nets class extended with time-domains and asynchronous-channels is given by Eq. (3.25).

$$\mathrm{PN}_{\mathrm{GALS}} = (\mathrm{PN}_{\mathrm{AC}}, \mathrm{bound}) \tag{3.25}$$

bound is a function associating places with non-negative integers:

$$\mathrm{bound} : (P \cup \mathrm{AC}) \rightarrow \mathbb{N}_0 \tag{3.26}$$

where $\mathbb{N}_0 = \{0, 1, 2, 3, \ldots\}$.

3.8 Decomposition into Implementable Sub-models

After the global GALS-DEC model creation and validation, it must be decomposed into a set of implementable sub-models that support the components implementation code generation, potentially using automatic code generators, such as those from IOPT-Tools. The algorithm that supports this decomposition is presented in this section. This algorithm (Algorithm 2) reads the global GALS-DEC PNML file and creates a set of PNML files, where each file contains the sub-models that specify each component. These files that fully specify the synchronous components can be used as inputs in automatic code generators. A decomposition tool, which implements this algorithm, was added into the IOPT-Tools (Pereira et al. 2012a). The created files can be used as inputs in automatic code generators, such as those from IOPT-Tools: (1) C code generators (Campos-Rebelo et al. 2011; Pereira et al. 2012a) and (2) VHDL code generators (Gomes et al. 2007b; Pereira and Gomes 2013).

To create the sub-models of each component (with a specific time-domain), the algorithm:

- reads the PNML file of the global GALS-DEC model;
- removes the nodes (places and transitions) that do not have the time-domain of that component;
- removes the arcs that were connected to the removed nodes;
- removes the asynchronous-channels and introduces: (1) additional sub-models; and (2) additional input events and output events (to specify the interaction between the components and the communication nodes);
- saves the resulting sub-models into a new PNML file.

Describing the algorithm in more detail:

- line 1—the global Petri net model uploaded into the *globalPN* data structure;
- line 2—a list with all time-domains of the *globalPN* is created;
- lines 3 to 48—for each time-domain, the model of the associated component is created;
- line 4—new data structure (*componentPN*) is created with a copy of the global model (at the end this new structure will contain the component model);

Algorithm 2 The decomposition algorithm that reads the global model and creates the implementation sub-models of each component

Require: *globalPNname*
1: $globalPN \leftarrow Read(globalPNname)$
2: $timedomainList \leftarrow GetTimeDomains(globalPN)$
3: **for all** $timeD \in timedomainList$ **do**
4: $componentPN \leftarrow globalPN$
5: **for all** $a = (x, y) \in componentPN.A$ **do**
6: **if** $td(x) \neq timeD \vee td(y) \neq timeD$ **then**
7: $componentPN.RemoveArc(a)$
8: **end if**
9: **end for**
10: **for all** $p \in componentPN.P$ **do**
11: **if** $componentPN.td(p) \neq timeD$ **then**
12: $componentPN.RemovePlace(p)$
13: **end if**
14: **end for**
15: **for all** $ac = (t_s, T_t) \in componentPN.AC$ **do**
16: **if** $td(t_s) = timeD \wedge ac \in SAC$ **then**
17: $componentPN.AssignOutEvToTransition(t_s, ac)$
18: **end if**
19: **if** $\exists(t_t \in T_t) : td(t_t) = timeD$ **then**
20: **for all** $t_t \in T_t$ **do**
21: $componentPN.AssignInEvToTransition(t_t, ac)$
22: **end for**
23: **if** $\exists(aac = (t_t, T) \in AAC)$ **then**
24: $componentPN.AddNewTransition('tdeliver', ac, timeD)$
25: $componentPN.AssignInEvToTransition('tdeliver', ac)$
26: **end if**
27: **for all** $aac = (t_t, T) \in AAC$ **do**
28: $componentPN.AddNewOutEvToTransition('tdeliver', ac, aac)$
29: **end for**
30: **end if**
31: $componentPN.RemoveAC(ac)$
32: **end for**
33: **for all** $t \in componentPN.T$ **do**
34: **if** $componentPN.td(t) \neq timeD$ **then**
35: $componentPN.RemoveTransition(t)$
36: **else**
37: **if** $\exists(nac = (t, T) \in NAC)$ **then**
38: $componentPN.AddNewTransitionWithLowerPriority('tnotenabled', t, timeD)$
39: $ac : ac = (t_x, T_x) \in AC \wedge t \in T_x$
40: $componentPN.AddNewInEvToTransition('tnotenabled', t, ac)$
41: $componentPN.AddNewPlace('pxor', marking = 1)$
42: $componentPN.AddNew4Arcs(t,' pxor',' tnotenabled')$
43: **end if**
44: **for all** $nac = (t, T) \in NAC$ **do**
45: $componentPN.AddNewOutEvToTransition(`tnotenabled', t, nac)$
46: **end for**
47: **end if**
48: **end for**
49: $CreateNewPNMLfile(componentPN)$
50: **end for**

- lines 5 to 9—each arc that connects one node with a different time-domain (from another component) is removed;
- lines 10 to 14—each place with a different time-domain (from another component) is removed;
- line 15—for each asynchronous-channel (AC)
- lines 16 to 18—if its source transition has the (component) time-domain and the AC is a SimpleAC, an output event is associated with the source transition;
- line 19—if any of the target transitions have the (component) time-domain;
- lines 20 to 22—an input event is associated with each of its target transitions;
- lines 23 to 26—if any of the target transitions is source of an AckAC, a new transition ("tdeliver") with an associated input event is added;
- lines 27 to 29—for each AckAC that is source of this target transition, a new output event is associated with the new transition ("deliver");
- line 31—the asynchronous-channel is removed;
- line 33—for each transition ("t");
- lines 34 to 35—if it has a different time-domain (from another component) is removed;
- lines 36 to 43—else, if the transition is source of a NotAC, a sub-net is added. This sub-net:
  - has a new transition ("tnotenabled") with an input event and with lower priority than transition "t';
  - has a new place "pxor" with one token;
  - has four new arcs: (1) one connecting the transition "t" to "pxor"; (2) one connecting "pxor" to the transition "t"; (3) one connecting "tnotenabled" to "pxor"; and (4) one connecting "pxor" to "tnotenabled";
- lines 44 to 46—for each NotAC, a new output event is associated to transition "tnotenabled";
- line 49—the component model is saved into a PNML file.

The decomposition of the global model presented in Fig. 3.5 produces the sub-models presented in Fig. 3.11. These sub-models support the implementation of the components of the GALS-DEC specified in Fig. 3.5. They can be used as inputs in automatic code generators, such as the C code generator (Pereira et al. 2012a) and the VHDL code generator (Pereira and Gomes 2013), available in the IOPT-tools (Pereira et al. 2012a).

3.9 The Meta-Model of PNs Extended with TDs and ACs

The meta-model of Petri nets extended with time-domains and asynchronous-channels is presented in Fig. 3.12. The proposed meta-model, which is specified through UML class diagrams and OCLs, extends PT-nets (the PT-net meta-model is presented in Fig. 2.3), and complements meta-model definition for IOPT-nets,

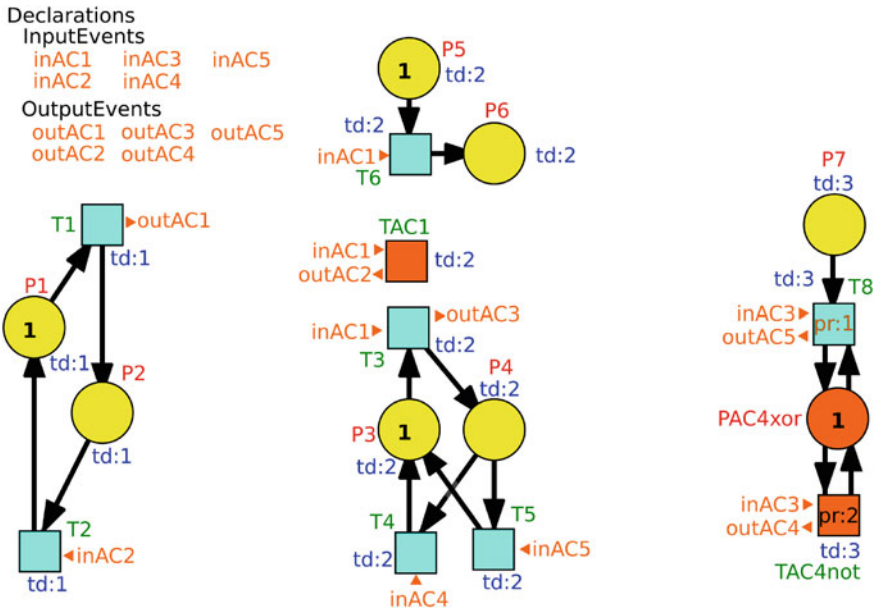

Fig. 3.11 The components implementation sub-models (resulting from the decomposition of the global model presented in Fig. 3.5)

as in Moutinho et al. (2010) and Gomes et al. (2014). OCLs are used to express constraints that cannot be expressed in the UML class diagrams. As defined in Sects. 3.4 and 3.5, this meta-model also defines that:

- each node (a place or a transition) has a time-domain;
- each arc connects two nodes with the same time-domain;
- a reference transition must always refer a transition with the same time-domain;
- a reference place must always refer a place with the same time-domain;
- each asynchronous-channel can be a simple AC, an acknowledged AC, or a not-enabled AC;
- each asynchronous channel has one source transition and one or more target transitions;
- each transition cannot be target of more than one asynchronous-channel;
- all target transitions of an asynchronous-channel must have the same time-domain;
- the source transition of an acknowledged AC or not-enabled AC must be the target of another asynchronous-channel.

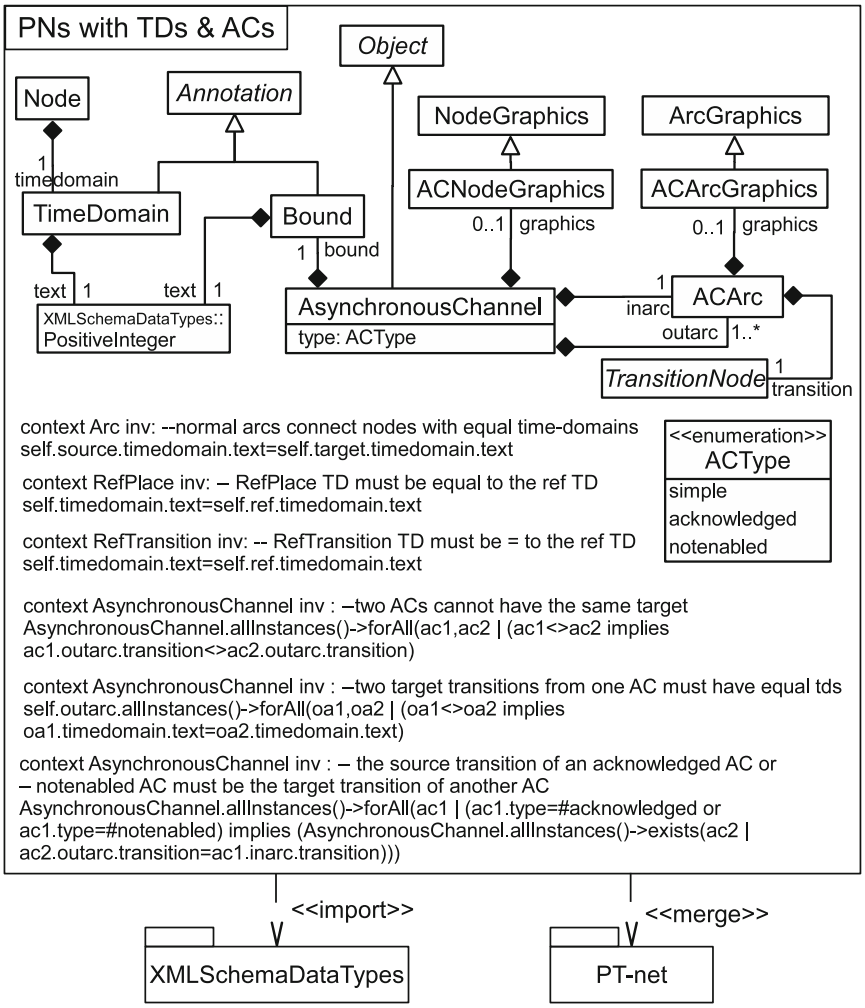

Fig. 3.12 The meta-model of Petri nets extended with time-domains and asynchronous-channels