Composition of software artifacts modelled using Colored Petri nets

Leandro Dias da Silva, Angelo Perkusich

Department of Electrical Engineering, Federal University of Campina Grande, C.P. 10105, 58109-970, Campina Grande, PB, Brazil

Received 30 October 2003; received in revised form 1 September 2004; accepted 6 September 2004
Available online 13 December 2004

Abstract

In this work we introduce a new formal model for software components supporting behavioral interpretability based on temporal logic, Petri nets, model checking, and an assume–guarantee strategy to specify and reason about the composition of concurrent component systems. The formal specification and verification strategies, methods, and techniques presented in this work contribute to the development of more dependable component-based software systems, in a modular way. An approach based on two complementary formalisms, Hierarchical Colored Petri Nets (HCPN) and temporal logic, is introduced. HCPN are used to visualize the structure and model the behavior of software architectures and components, and temporal logic is used to specify the required properties of software architectures and component interfaces.

Keywords: Component-based software development; Hierarchical Colored Petri nets; Assume–guarantee reasoning; Model checking
1. Introduction

Component-based software development promotes the design of evolving complex software systems, on the basis of the recovery of components from a repository and plugging them together to build less fragile and more dependable systems applications [20,6]. However, as pointed out by Lumpe and Schneider in [14], part of the fragility is merely transferred from the component artifacts to the connectors and the composition process adopted. Therefore, there is no guarantee that the resulting software system is more robust, dependable, or reliable than any other one developed using other approaches. According to [20], improving the theoretical and practical foundation of composition techniques is thus essential for improving the overall component software dependability and reliability.

Another important trend that should be taken into account in the context of software engineering is the reusability of different artifacts produced during the development of a system. In addition to code, analysis and design models can also be reused. This is mandatory for improving the product time to market, software quality, and maintenance, and for decreasing development costs. In this context, software frameworks have been introduced as a possible way of supplying these needs.

Our focus is the compositional formal specification and verification of concurrent embedded systems using component models [8,15,13]. The required behavioral properties are given by a specification, which precisely defines what the system must perform. Formal methods provide a rigorous mathematical guarantee that a software system conforms to a given specification. Formal methods can be mainly classified as proof-theoretic and model-theoretic. In the first case a deductive system is used, and correctness proofs are built using a theorem prover as a model checker [1]. In the second case, a model of the run-time behavior of the software is built, and this model is automatically checked for the required properties. In this work, the emphasis is on model-theoretic methods. The use of formal models aggregates some advantages to the development process, such as automatic verification of properties before developing the system, as well as the detection of logical errors and simulation. Hierarchical Colored Petri Nets (HCPN) [10,11] is the modeling formalism adopted, and the Design/CPN [12] set of tools is used to edit and analyze the models. The use of HCPN allows more compact and organized models because of hierarchy and complex data type features. The ASK/CTL library [3] is used in the verification process. It allows the description of properties on the basis of computation tree logic and for model checking purposes [4].

The main objective of this work is to apply methods and techniques to formally specify and verify embedded systems using the concepts of components and to define a modular reasoning strategy based on an assume/guarantee paradigm. Besides focusing on this main objective, other concepts are invoked, such as: product lines [5]; software architecture [18]; and components [20,6]. These concepts are used in order to define a development process that allows one to assemble systems on the basis of a framework [7].

This work is organized as follows. In Section 2 concepts related to temporal logic are introduced. In Section 3 the development process as defined in [7] is presented. In Section 5 the concept of Component-Based Petri Nets (CBPN) and how it can be applied to the
specification and verification of embedded systems are detailed. In Sections 6 and 7 a
modular strategy analysis for CBPN and related works are presented. Finally, in Section 8
some conclusions and future work are discussed.

2. Temporal logic

Temporal logic is a modal logic that can be used to describe how events occur over
time. There are operators for describing safety, liveness, and precedence properties. Thus,
a framework for specifying software systems, particularly concurrent systems, is provided.
In this paper, a Computation Tree Logic (CTL) [4] defined for colored Petri nets named
ASK-CTL [3] is used. In what follows the basic concepts of both logics are introduced.

The CTL temporal logic combines path quantifiers with linear time temporal logic
operators. The path quantifiers $A$ ("for all paths") and $E$ ("for some paths") should be
used as a prefix of one of the operators $G$ ("always"), $F$ ("sometimes"), $X$ ("next time"),
and $U$ ("until"). The syntax of CTL is given by the following rules:

(1) If $\varphi \in AP$, then $\varphi$ is a formula, where $AP$ is a set of atomic propositions.
(2) If $\varphi_1$ and $\varphi_2$ are formulas, then $\neg \varphi_1$, $\varphi_1 \lor \varphi_2$, and $\varphi_1 \land \varphi_2$ are also formulas.
(3) If $\varphi_1$ and $\varphi_2$ are formulas, then $EX\varphi_1$, $EG\varphi_1$, and $E[\varphi_1 U \varphi_2]$ are formulas.

Other CTL operators are expressed using the three operators $EX$, $EG$, and $E[U]$, and:
$AX\varphi \equiv \neg EX\neg \varphi$, $EF\varphi \equiv E[trueU\varphi]$, $AG\varphi \equiv \neg EF\neg \varphi$, $AF\varphi \equiv \neg EG\neg \varphi$, and
$A[\varphi_1 U \varphi_2] \equiv \neg E[\varphi_2 U (\neg \varphi_1 \land \neg \varphi_2)] \land \neg EG\neg \varphi_2$.

The semantics of CTL is defined with respect to paths in a state transition graph or a
Kripke structure [4], denoted by $M$. A path is an infinite sequence of states $(s_0, s_1, \ldots)$
such that $s_{i+1}$ is reached from $s_i$ for all $i \geq 0$. Therefore, if $\varphi$ is a CTL formula, $M, s \models \varphi$
denotes that $\varphi$ holds for $s$ in $M$.

The four most commonly used CTL operators are $EF$, $AF$, $EG$, and $AG$, their
definitions are as follows.

$EF\varphi \equiv E[trueU\varphi]$ means that there exists a path starting from state $s_0$ in which $\varphi$
holds at some state along this path.
$AF\varphi \equiv A[trueU\varphi]$ means that for all paths starting from state $s_0$, $\varphi$ holds at some state
along the path.
$EG\varphi \equiv \neg AF\neg \varphi$ means that there exists a path starting from state $s_0$ in which $\varphi$
holds at every state along this path.
$AG\varphi \equiv \neg EF\neg \varphi$ means that for all paths starting from state $s_0$, $\varphi$ holds at every state
along that path; thus $\varphi$ holds globally.

ASK-CTL is a CTL-like logic useful for specifying properties related to CPN (Colored
Petri Nets) state spaces, represented by occurrence graphs. Occurrence graphs carry
information on both nodes and edges. Hence, a natural extension for CTL is including the
possibility of expressing properties concerning the information labeling for the edges (e.g.,
edge information is needed when expressing liveness properties since liveness is expressed
by means of transition occurrence information). For this purpose, two mutually recursively
syntactic categories of formulas are defined, state and transition formulas, which are
Fig. 1. The component-based development cycle.

interpreted for the occurrence graph for states and transitions respectively [2]. Quantified state formulas and transition formulas are interpreted along paths. Path quantification is used in combination with the until operator to express temporal properties.

The ASK-CTL library is composed by two parts: one which implements the ASK-CTL logic language, and another one which implements the model checker. The ASK-CTL syntax is minimal, and in order to increase the readability of the formulas, syntactic sugars are used, e.g., \( \text{POS}(\phi) \) means that it is possible to reach a state where \( \phi \) holds, \( \text{INV}(\phi) \) means that \( \phi \) holds at every reachable state, and \( \text{EV}(\phi) \) means that \( \phi \) holds within a finite number of steps for all paths.

3. The component-based specification and verification process

In Fig. 1 we show the development cycle for component-based software systems defined in [7] and used in the context of this paper. As our focus is on a model-based development process, the modeling phase is the key to a successful design. This is mainly due to the fact that a “bad” model can compromise the implementation and validation of the system for the users or clients, thus increasing the costs. Moreover, the use of formal methods allows automatic analysis before the implementation, and can be used as an effective documentation tool. On the basis of such considerations, as stated in the introduction, HCPN is the formal method adopted in this paper. Observe that just the requirements and system phases are not considered, since they are not modeling specific.

Another advantage of using a well defined and formal modeling process is the possibility of automating the phases. For instance, automatic code generation is possible, by associating code with the framework and with the models of the components. Some syntax for naming the elements of the HCPN models and then generating code automatically can be defined. Since the framework is fixed, and all the connections, restrictions, decisions, and locations are defined, code can be associated with the
framework and thus, both, the model but and the framework code can be reused. The same consideration applies to the components, but in this case each specific component has its own associated code. When the project is changed, by modifying the components, a refactoring strategy at the model level instead of the code level is being applied. Such refactoring is much simpler than that at code level. This discussion also applies if all the code is generated from scratch using a given syntax definition, as mentioned above.

In this work the architecture is formally defined by means of a framework described by an HCPN. On the basis of this framework one can then proceed to recover models from a repository or develop new ones on the basis of the requirements of the project. In some situations, recovered models of components need to be adapted to satisfy a specific use case for each project. Having the framework and all the components, it is possible to go one step further to the integration phase. Then, the models for the components are integrated in the framework. The next step is the use verification. This step is necessary to guarantee that the whole model does not violate the semantics of the individual models for the components. Moreover, the use verification is useful for validating new models specifically developed for a new project and for making them available in the repository to be used in future projects.

4. Specification and verification

In this section the formal specification and verification of an embedded system using HCPN and the process described in Section 3 are introduced. As pointed out in the introduction, HCPN promotes more compact and organized descriptions due to the fact that it incorporates concepts such as complex data types and hierarchy, that can be used by the designer to structure a model. The hierarchy is defined by two mechanisms, namely, substitution transitions and fusion places. The first can be used to represent another model, named the page. A page represented by a substitution transition is a sub-page, whereas the page where the transition appears is called a super-page. Thus, each page is a CPN and all the pages “glued” together result in an HCPN model. The pages are “glued” by means of ports in the sub-pages, that are associated with sockets in the super-page – that is, the input and output places of the substitution transition. Fusion places are distinct places that are associated with a fusion set and always have the same marking. The marking of a place is a multiset\(^1\) of tokens in a place in a given moment. Therefore, changing the marking of a place in a fusion set results in changes in the markings of all places defined for the fusion set. In the case of ports and sockets the same behavior is observed. Using these two mechanisms it is possible to organize the models by modularizing them. We remark that it is important to observe that this modularization is merely visual and the behavior of the model is interpreted on the basis of a plain CPN model.

As stated before, the application domain considered in the scope of this paper is an embedded transducer network control system, composed of a set of transducers, a communication server, and a real-time server, as detailed in [17]. The environment signals acquired by the sensors are transformed and controlled such that the real-time server

---

\(^1\) A multiset is a set where it is possible to have several occurrences of the same element.
can access and modify the information for controlling the actuators, according to the applications requirements. Several different applications may access the real-time server to acquire data and to control different devices.

An important observation concerning the specification in the scope of this work is that details related to specific technologies used to implement the components are not considered. The focus is on the specification and analysis of the architecture of the system. This is also due to the fact that one of the main objectives of this work is related to modular verification techniques. Therefore, properties of the interfaces of the components and the architecture are verified, regardless of their internal details. For instance, the particular protocol implemented for the embedded system to communicate with the control system, running in the real-time server, is not considered.

4.1. Specification

In Fig. 2 the HCPN hierarchy specifying the architecture for the system is shown. The System page models the sensors and actuators that communicate only with the embedded system. The Embedded page is the model for the embedded system, and it is shown in Fig. 3. There are several components defined for the embedded system page. The information is exchanged between the devices and the embedded system using a blackboard mechanism. The input and output interpreter, or I/O interpreter (IOParser as shown in Fig. 2), is used to instantiate the data written in the blackboard as objects. Also, this component receives objects from the system and translates them to the data format used by the devices. The next component is the data converter, DataConvert. This
component translates data from the I/O interpreter to a format used by the real-time server, as defined by the applications. Therefore, this component must be changed to satisfy particular requirements of each project. The data flow is defined in the data converter component. If the data is a control request, such as an initialization or calibration signal,
the data is then sent to the device controller. On the other hand, if the data is an information signal, it is sent to the synchronizer and then to the real-time server.

The device controller component, DeviceController, is used to control devices. An application might issue requests to change the configuration attributes for a given device, for example, the sampling rate. The device controller implements such changes. The synchronizer is a realization of the communication between the embedded system and the real-time server. When an information signal is sent by a sensor, it must be transmitted to the real-time server through the synchronizer. Thus there is one synchronizer for the embedded system and another one for the real-time server. Since this communication pattern does not change, the synchronizer is a fixed component in the architecture.

The real-time server, RealTime, is used to mediate the communication between the embedded system and a set of applications. A database with information about the network and the applications is used to promote this communication. The applications can read or write information in order to control a system. In Fig. 4 the components of the real-time server are shown. The synchronizer is identical to the one in the embedded system. The data controller, DataController, is used to control the data flow from and to the applications. The user interface module (UI Module) component makes services available to the applications that access the system.

In Figs. 3 and 4 the dashed lines indicate components that can be replaced, or hot spots, whereas the continuous lines indicate components that do not need to be changed, or frozen spots.

On the basis of this architecture, and the process described in Section 3, one can specify any control system as defined in the context of this paper, and then allow a product line evolution based on the reuse of models of components. Moreover, this strategy allows the practice of refactoring at a model level and not only code level.

4.2. Verification

In this section the models introduced are used to prove properties on the basis of model checking. To do this, in a first step, properties to be proven are identified and defined. As the properties to be proven are related to the architecture and to the interface of the components, the model of the framework is used without taking into account internal details of the components. Therefore, functional scenarios are identified in order to define relevant properties to be proven. Message Sequence Charts (MSC) are used to describe the scenarios. They are automatically generated during the simulation of the HCPN model.

The verification strategy defined in this work consists of the following steps: (i) identification of scenarios; (ii) automatic MSC generation for each scenario, based on the model; (iii) definition of the properties to be proven on the basis of the scenarios; (iv) definition of atomic propositions and temporal logic formula specifications; and (v) performing model checking.

Let us assume that the devices send an initial signal when the system is turned on or a new device is plugged in. Also, suppose that the system performs some task when it receives this kind of message, and sends back to the device an acknowledgement, calibration, or even an initialization message. The MSC shown in Fig. 5 relates to this scenario. When a device sends an initialization signal, the data converter sends it to the device controller
Fig. 4. A real-time server.

Fig. 5. Data converter flow for the control signal.
to perform the associated control tasks. Since an MSC diagram is generated on the basis of simulation it captures a single execution sequence or information flow in the model. Therefore, it might be the case that there is some situation where this expected sequence or flow is violated. Thus, the scenario must be verified for all possible situations to guarantee that the expected flow is always satisfied. To do this, model checking is used.

Considering again the scenario shown in Fig. 5, it must be proven that when an ini message is sent by some device, the flow is always through the device controller. The formula \( \text{AG}(PA \rightarrow \text{AF}(PB)) \) specifies this scenario. The proposition \( PA \) is true when there is a token in place IODConIn (input of the Data Converter) for the page Embedded, with an initialization signal (ini); and \( PB \) is true when there is a token in place DevControlIn (input of the Device Controller) for the page Embedded. Thus, the formula is true if \( PA \) is true and eventually \( PB \) is also true. This means that if there is a token in the input of the data converter, this token is sent to the input of the device controller. The device controller is the component that implements control tasks such as initialization, calibration, and changing working parameters for the devices. The evaluation of this formula to true means that this part of the model behaves as expected for all possible executions. The same reasoning can be used to prove that the information flow back to the device also occurs as expected, for all possibilities.

Let us suppose now that the signal is a normal measurement signal sent by a sensor. This information, that is, what is measured by the sensor, must be sent to the synchronizer and then to the real-time server. In Fig. 6 the MSC for this scenario is shown. For the real-time server the information is manipulated by a specific control application, depending on the sensor and the information itself. It might be the case that the information is returned to the embedded system to make it available to an actuator. The formula specifying this scenario is: \( \text{AG}(PA \rightarrow \text{AF}(PB)) \). The proposition \( PA \) is evaluated to true if there is a token in place IODConIn for the page Embedded, that is, the data converter input; and proposition \( PB \) is evaluated to true if there is a token in place DConOutSync for the page Embedded. Thus, the formula is evaluated to true if \( PA \) is evaluated to true and eventually \( PB \) is evaluated to true. Thus, if true, it proves that when the token for an information signal is in the input of the data converter, it is always sent to the synchronizer in order to be transmitted to the real-time server.

For the synchronizer there are two complementary models, one for the embedded system and another one for the real-time server – thus, modeling a full-duplex communication channel. When a token is in the input of the embedded system synchronizer
it will always be in the output of the real-time server synchronizer. The reasoning holds also for the inverse flow. In Fig. 7 the MSC for this scenario is shown, and the formula specifying it is $AG(P_A \rightarrow AF(P_B))$. The proposition $P_A$ is evaluated to true if there is a token in place $SyncIn$ for the page $Embedded$, the input of the embedded system synchronizer. Proposition $P_B$ is evaluated to true if there is an identical token in place $SyncOutRT$ for the page $RealTime$, the output of the real-time synchronizer. The complementary part of this flow can be proven following the same reasoning. It is only necessary to change $P_A$ to verify the input of the real-time synchronizer and $P_B$ to verify the output of the embedded system synchronizer.

4.3. Considerations concerning the verification

As stated before, in this work the focus is on the interaction among components and the architecture regardless of specific technical component details. It is important to observe that the proofs are performed at the interface level of the components. This can be considered a modular reasoning mechanism. Following the concepts of components, reuse of models, model-based development, design patterns, refactoring, software architecture, and product line, a strategy for software modeling has been introduced. This strategy has been applied to an embedded control system. It is important to point out that a specific data format can be used in an application, such as real numbers. Therefore, the data converter needs to deal with this. In this case, besides the control and information flow, how data is manipulated must be verified. For example it is necessary to prove that the data converter always performs the correct data transformation.

The properties to be proven depend on the kinds of components used and also on the system being developed. The formal specification and verification approach presented in this work and its application to the embedded system domain represent a contribution to software development in relation to modular reasoning and model-based development and refactoring.

5. Composition

In this section the concept of Component-Based Petri Nets (CBPN) is introduced. Also the application of CBPN to model systems based on components is discussed. The basic
The description language is Colored Petri Nets (CPN), for which there is no semantics defined for communication and composition of modules. In Fig. 8 we show how individual components modeled on the basis of CPN are put together in a hierarchy. The System model is the top level model defined for the hierarchy. Associated with this model there is a substitution transition that represents the CPN model Embedded. The communication between these two components is effected by interface places. The input and output places of the transitions for the model System are associated with the input and output ports for the Embedded model. Using this mechanism, the hierarchy shown at the bottom of the figure is defined. It is important to observe that there is an exact point where the model System communicates with the model Embedded. This is the role of the architecture. Actually, the Embedded model is itself composed of several models. The System, Embedded, and the RealTime models for the hierarchy are fixed and they define the architecture of the system. The other models specify components that perform the features or functionalities of the system. For analysis purposes, all modules are put together, thus resulting in a flat
Using assume–guarantee techniques it is possible to define and use, for verification purposes, the behavior of a model without having knowledge of its internal structure. This is called black-box reuse. For such a technique it is not necessary to calculate the whole internal state space for all the models. Each model has the assumptions and guarantees specified by a CPN model. Thus, for each component model there is one or more assume–guarantee model associated with it. For verification purposes an assume–guarantee model can be substituted for the original model. This idea is illustrated Fig. 9. For this example, the assumption is that whenever there is an information token, represented by the field info, in place IODConIn, it is guaranteed that this token is put in place DConOutSync. This is one possible assume–guarantee specification for the Data Converter component. Therefore, there are assume–guarantees specifications for all the features of a component and they are used in conjunction with the initial marking. Therefore, when the initial marking for the model is defined, it is possible to know, and automatically generate, the assume–guarantee specification for each model for such a situation. Thus, to perform model checking, the models for the components are replaced by their assume–guarantee specifications and the state space that is calculated on the basis of this technique is much smaller than when using the original models. The formal definitions of the assumptions and guarantees in the context of this work are as follows.

**Definition 1.** Let $P$ be a finite set of places, and $M$ a finite set of markings for a given CPN model. An assume–guarantee specification is a function $AG : P_i \times M_i \Rightarrow P_o \times M_o$ where:

- $P_i \in P$ is an input place, and $P_o \in P$ is an output place;
- $M_i \in M$ is the marking for place $P_i$, and $M_o \in M$ is the marking for place $P_o$.

An assume–guarantee specification is a function from an input place and its marking to an output place and its marking. Informally speaking, an assume–guarantee specification can be expressed as: *whenever there are specific tokens for specific input places, there will always be specific tokens for the specific output places.*
It is important to observe that there can be no, one, or several assume–guarantee specifications for any CPN model; this is to reflect all the possible behaviors of the model. Also these specifications must be used depending on what is supposed to be proved. Thus, it might be the case that there is no need to specify all the assume–guarantee models for each CPN model.

On the basis of Definition 1 a CBPN is defined as follows.

**Definition 2.** Let \( CBPN = (CPNM, AGS, MR) \) be a component-based colored Petri net; then:

- \( CPNM \) is a finite set of CPN models for the components;
- \( AGS \) is a finite set of assume–guarantee specifications according to Definition 1;
- \( MR : CPNM \Rightarrow 2^{AGS} \) defines the set of assume–guarantee specifications associated with a given CPN model.

In the context of this work an assume–guarantee specification is represented by a CPN model, as illustrated in Fig. 9. Also, it is important to observe that a composition of components is still a component. Thus, it is possible to say that the architecture itself is a component, and the components plugged into the architecture can still be seen as a component.

### 6. A modular reasoning strategy

In this section we describe a modular reasoning strategy to verify the composition of component-based software systems. Observe that the advantages of the use of modular techniques and reasoning are not important only at the specification level. Also, it is important to have modular verification techniques. For the component-based software domain it is important and recommended to apply modular reasoning strategies, methods, and techniques to verify component-based software systems. CBPN is adopted as the description language. As stated in Section 5, the main idea of the CBPN formalism is to promote the development of models for software artifacts and to compose them on the basis of reusable basic models. In order to build models, the process presented in Section 3 and detailed in [7] is adopted. An important remark is that the basic models depend on the application domain. For each domain it is necessary to define a set of basic models and a framework for integrating new systems for the new domain. In the following a modular reasoning approach for verifying component-based software systems described using CBPN is discussed.

Let \( P_x \) be a set of atomic propositions, where

\[
x = \{ A_i, B_i, C_i, \ldots, Z_i \}, \ 1 \leq i \in \mathbb{N}.
\]  

(1)

The set of components is given by

\[
C = \{ C_1, C_2, \ldots, C_n \}, \ 1 \leq n \in \mathbb{N}.
\]  

(2)

Since the components are described by CPN models, as given in Definition 2, it is possible to rewrite Eq. (2) as follows:

\[
CPNM = \{ CPNM_1, CPNM_2, \ldots, CPNM_n \}, \ 1 \leq n \in \mathbb{N}.
\]  

(3)
Let us assume that a property of a specific subset of a system is to be proven. If the system is composed of components, one can prove properties on the basis of the interfaces of the components regardless of the internal details of the individual components. In Fig. 10 an illustration of a composition of components is shown. Using the model described in Section 4.1, assume that the components in Fig. 10 are the I/O interpreter, the data converter, and the device controller. These components are used when a control signal is sent by a device. For a specific purpose, a proof related to the interface, say $PA$ and $PB$, is sufficient; that is, it is only necessary to prove properties related to the interface. But suppose that a proof for the complete flow related to the control activity for this model must be found. In this case it is necessary to prove the interface properties for all the components for the flow, say, C1, C2, and C3. The scenario for this case is illustrated by the message sequence chart shown in Fig. 11.

The proof is performed by proving properties related to the interfaces of the components defined in Fig. 10, by using $PA_1$ and $PB_1$, $PA_2$ and $PB_2$, $PA_3$ and $PB_3$. The granularity of the proof depends on what is supposed to be proven. Thus, the proof is defined as a set of proofs:

$$V = \{V_1, V_2, \ldots, V_m\}, \ 1 \leq m \in \mathbb{N}. \tag{4}$$

Considering that the components can be represented by their assumptions and guarantee specifications as described in Section 5, Eq. (4) can be written as

$$V = \{V_{AG1}, V_{AG2}, \ldots, V_{AGm}\}, \ 1 \leq m \in \mathbb{N}. \tag{5}$$
On the basis of the scenario shown in Fig. 11 and the model of the embedded system presented in Fig. 8, the atomic propositions and the formulas for performing the proof can be specified. Also the atomic propositions associated with the interfaces of components $C_1$, $C_2$, and $C_3$ are

$$PA_1 = IOInHard, PB_1 = IOOutComp$$
$$PA_2 = IODConIn, PB_2 = DConOutDevC$$
$$PA_3 = DevControlIn, PB_3 = DevControlOut$$

The formulas are

$$V_1 = (\wedge (PA_1, EV(PB_1)))$$
$$V_2 = (\wedge (PA_2, EV(PB_2)))$$
$$V_3 = (\wedge (PA_3, EV(PB_3)))$$

where $\wedge$ is the boolean conjunction operator and $EV$ is the temporal eventuality operator, as defined in Section 2. Finally, the proof is given by

$$V = (\wedge (V_1, V_2, V_3)).$$

In this work more general operators for composing models of components are defined. Therefore proofs can be composed, and a model of a system can be obtained by composition. Let $C$ be a set of components as defined is Eq. (2), and $V$ a set of verifications as defined in Eq. (4). Also, let $A$ be the architectural part of the system. The integration operator $\oplus$ represents the integration of the components together with the architecture. The operator $\sqcup$ is the component composition operator. Thus, a system $S$ can be defined as follows:

$$S = (A \sqcup C).$$

(6)
From Eqs. (2) and (3), Eq. (6) is rewritten as follows:

\[ \text{CPNM} = \left( A \uplus \left( \text{CPNM}_1 \uplus \cdots \uplus \text{CPNM}_n \right) \right). \]  

(7)

As stated before, a composition of components is still a component, and an architecture is a component itself. Thus, \( \mathcal{A} \) can also be seen as a component \( \text{CPNM}_x \). We used different notation and different operators for the architecture and the components to make explicit the fact that the composition of components is performed using an architecture to guide the process.

The compositional verification for a component-based system is given by

\[ V = (V_1 \uplus V_2 \uplus \cdots \uplus V_m). \]  

(8)

And from Eq. (5) it is possible to rewrite Eq. (8) as

\[ V_{AG} = (V_{AG1} \uplus V_{AG2} \uplus \cdots \uplus V_{AGm}). \]  

(9)

It is important to observe that in the context of this work a modular verification using flow properties has been used, but other kinds of properties can be verified on the basis of the strategy introduced. This can be accomplished by defining the appropriate atomic propositions.

7. Related work

Petri nets have been applied as the formal foundation for component-based systems in different research works. Some research works also pointed out the need to use a complementary approach to deal with the inherent complexity associated with the development of a component-based software system. He et al. [9] introduce a general software architecture development framework based on Petri nets and temporal logic. Petri nets are used to visualize the structure and model the behavior of software architectures, while temporal logic is used to specify the required properties for software architectures. Although the approach is very similar to the one introduced in this paper, aspects related to modular verification are not taken into account. In [19] the authors introduced an approach to component composition and validation. The approach is very similar to the process discussed in Section 3. Other research applying Petri nets is reported by Ziaei and Agha in [21], using the name SynchNets. The major idea is to promote the separation of the coordination aspects from computational aspects. In [16] the authors introduce a design strategy for evolvable architectures based on net components, named FEVA net, that are based on Petri nets. These nets are applied for modeling business processes as well as their consistent composition. In both cases the compositional verification is not addressed.

8. Conclusion

In this paper, a contribution towards a new formal component model supporting behavioral interpretability based on the use of temporal logic, Petri nets, model checking, and an assume–guarantee strategy for specifying and reasoning about the composition of concurrent component systems is introduced. It is assumed that in addition to code,
Design models of software artifacts can be reused and composed. The formal specification and verification strategies, methods, and techniques presented in this work contribute to the development of more dependable component-based software systems. Also, the components can be reused from previous successful projects and it is not necessary to prove specific internal component properties for every new project. Instead of this, only properties related to the new assembled or integrated model at the architectural and interface level must be proved.

Unlike in other temporal logic and Petri net-based methods for software components, exhaustive state-space enumeration, which can result in the well known state-explosion problem, is avoided. To deal with the latter problem, the idea of a modular strategy analysis approach for a CBPN model was introduced.

The approach discussed in this paper has been applied to a realistic and complex embedded software system, including a real-time database server and a transducer network [17]. Currently, there is a software team using the model specified and verified as presented in this work to write down the application code, thus linking the modeling and implementation phases, associating code with the framework and with the components.

Another important aspect is that it is possible to perform model level refactoring, allowing the insertion of new versions of a component or even the insertion of a new component without changing the code directly. Since time is a very important issue, how timing properties can be taken into account for the approach introduced is being investigated.

References


