Formal Verification of Component-Based Software Systems

Leandro Dias da Silva and Angelo Perkusich

Departamento de Engenharia Elétrica Universidade Federal de Campina Grande
Caixa Postal 10.105 - 58109-970 Campina Grande-PB-Brazil
Fone: +55 83 310 1137 Fax: +55 83 310 1015
leandro@dee.ufcg.edu.br, perkusic@dee.ufcg.edu.br

Abstract. In this work we present a methodology for formal modeling and verification of component based software systems. The methodology is based on software architecture, components, and reuse of Petri Nets models. With this process, we are contributing for the visual composition, verification, and formal validation of software. We also show an example of the application of this methodology into two different domains.

1 Introduction

Product lines have been used to develop different artifacts with similarities [10], promoting the development of products with less effort, time and cost. In order to have a software product line it is necessary to change the management and the production phases to take the advantages of this development process.

Software systems have been developed using product lines to obtain different products based on parts or common blocks. To do so, it is necessary a business strategy to guide the development process in order to obtain common values to be used in a known and defined domain. The developed and used values in a product line may vary from strategies established in early requirements definition and project decisions to executable code. Based on these parts, different systems can be developed with specific characteristics. A set of such systems is known as system family.

Based on such process there are two possible kind of business to be considered. The first one is the development of pieces of a system that can be used as a basis to develop a family of components on a common domain. The second one is based on the development of systems based on existing components [31,29,34,14,15].

When using components to develop a product it is necessary to have an integration strategy to define where and how the components can be integrated, how they are related, and possible integration restrictions, resulting in a software architecture [33]. Therefore a standard architecture for a family of systems that can be developed based on common components can be defined.

Thus, a system can be developed based on components as basic building blocks for the development of complex software systems. A component may be seen as an autonomous system implementing specific functionalities with a well defined interface.

Another relevant aspect is the need to have methods and techniques to manage the reuse of components. In this context formal methods can be used in order to increase the dependability on component-based systems. Furthermore, the use of formal methods in the modeling of systems aggregate several advantages such as, for instance, automatic
simulation, and proof of properties. Considering that, besides systematizing the modeling, to increase the dependability on the behavior of the system, we also want to use methods and techniques to classify, to recover, to adapt, to integrate and to perform use verification based on Petri Nets [30] and temporal logic [16].

Petri nets is a formal method with a graphical representation used to model concurrent systems. In the context of this work we consider Hierarchical Coloured Petri Nets (HCPN) as the modeling language [21,23]. It incorporates data types and hierarchy concepts to ease the modeling task. Coloured Petri Nets have been extensively used in many different applications [27,26,25,24], and the modeling activities are supported by a set of computational tools named Design/CPN [19,20].

Different phases define the life cycle of a software system, namely: requirements definition, modeling, coding, and tests. In this work we are mainly interested in the modeling phase, and in how a systematic and formal modeling approach can improve the development process of complex software systems. The aim is to use previous experiences and the definition of models and theories. Therefore, instead of solving a specific problem our objective is to define a generic strategy. We argue that the modeling phase is the most important one, and a formal modeling allow us to perform automatic formal verification before the system is coded.

The objective of this work is to introduce a systematic modeling process of component based software systems based on the reuse of Coloured Petri Nets (CPN) models. The objects of reuse are CPN software components descriptions.

The remaining of this paper is organized as follows. In Section 2 we discuss about product lines, software architecture and components. In Section 3 we discuss a life cycle for component based development and in Section 4 we describe the reuse process. In Section 5 we present some conclusions and suggestions for future works.

2 Product Lines, Components, Architecture

In this section some concepts and comments about them are presented. The use of components in a product line, with the definition of an architecture, have the objective of systematic development of complex software systems. In this work the term complex means large.

Several different definitions for components, software architecture, and product lines can be found in the literature. In this work we adopt the following definitions. A software architecture is a description of the structure of a software system with respect to its functionalities. This description specifies the relationship among the functionalities, the localization of them, and maybe interaction and relationship restrictions. Components are autonomous unities, with independent life cycles, that represent a specific functionality. A component is composed by a functionality, an interface, a contract, and possibly another non functional characteristics. Product Lines are different products, based on common assets, in a well specified domain. These assets are models of components and architecture, support tools, strategies, methods, and techniques.

In the specific context of modeling, the architecture is described by a framework to integrate models of components. The components have its functionalities modeled by means of Petri nets. Furthermore another non functional characteristics can be added to models in a repository. The management method of the repository makes the contract explicitly separated from the component model.

To define a domain, then a product line for this domain, and take advantages of previous projects efforts, is a way of efficiently use resources such as time and money,
guided by a development process. Moreover the use of formal methods allow us to
approach the problem of trustworthiness.

According to some works, components are a solution to develop larger systems using
building blocks [29,34,14]. This is called Component Based Development (CBD).
But, to make this a reality, the techniques and methods used until now to develop soft-
ware must be adapted or new ones defined to satisfy specific requirements of CBD. In
the context of Component Based Software Engineering (CBSE) we look for the defini-
tion of a set of disciplines that allow the CBD.

The concepts of product lines can be used to guide the component based develop-
ment. Early in the requirements analysis phase we need to keep in mind that the com-
ponents being developed will be used in another projects. Moreover, the system to be
developed must use existing components. The definition of such a process allow the use
of components because its success depends directly on the management and evolution
of a base of reusable components. Therefore changing the focus from the development
to the assembly of systems.

3 Life Cycle for Component Based Development

As pointed out in [14] the traditional life cycles used in software engineering have to
be adapted for component based development (CBD). Although many works deal with
such adaption no one emphasize the modeling phase in the development of software
systems.

The life cycle of CBD is divided into two different parts. First we have the de-
velopment of generic components for use in other applications. The major problem in
this case is the definition of the requirements because the component should be used in
several applications, even in some that were not considered while the component was
being developed. Second, when a component is used to build a new system one of the
main problem is to define how components are connected. One possible solution for
this problem is to explicitly define a software architecture.

The development of a system, keeping the reuse concept in mind, can be resumed
in the following steps:

– identification of the parts of the model of the new system (components);
– definition of a underlying framework to integrate recovered models (architecture);
– selection of the parts that need to be constructed and those that can be reused;
– description and recovery of the parts that can be reused;
– adaptation of the recovered models;
– integration of the recovered/adapted models in the framework;
– identification of reusable (sub-)models and storing them in the repository.

In Figure 1 the life cycle for the systematic development of systems based on com-
ponents defined for this work is illustrated. The modeling phase is present in almost
all the steps. As we want a high dependability and flexibility this phase is explicitly
emphasized. This figure illustrates the steps cited above.

With the requirements, an architecture for the system is then defined. This architec-
ture can specifically be developed for the given system or it can be reused from previous
projects. The specification of the architecture is in terms of the functionalities defined
in the requirements analysis, and the components satisfying the functionalities can be
identified. The definition of the architecture and the identification of the functionalities are considered in the modeling.

A search is performed to identify possible candidates. If more than one candidate for a functionality is found, it is necessary to identify which one is more adequate to the other non-functional requirements. It may be the case that will be difficult that a component satisfying all the requirements would be found. It might be necessary to adapt it before reusing it in the current design. It can also be possible that the architecture has to be adapted due to the selected components. It might be possible that some functionalities will not have components implementing them, or even might be the ones that aggregate value to the design. In that last case the functionalities may be the major difference of the system being developed compared to other systems or, even, a commercial secret of the system, and they should be developed locally.

The integration step consists of using a framework for the composition of the system based on the components. The framework defines how and where the components will be inserted and how they are related to each other. In the framework we have the part that does not change in a product line as, for instance, basic functionalities, communication mechanisms and synchronization. The recovery, adaptation, development, and integration steps are explicitly modeled. In the context of this article such steps will be discussed in Section 4. Artifacts as, for instance, code, must be associated to the models and to the framework. In this way, the modeling activity becomes the most important one in the development of the systems, changing the focus of the programming to the modeling for visual composition of software.

The last step is analysis and it may consist of several methods as, for instance, simulation, proof of properties, and tests. Even if the system is validated, changes in the requirements are possible along the time. Therefore we have a closed cycle. In other words, we are considering that a system always evolves, and this evolution is always treated by the modeling of the system. It is important to notice that, once we can have code associated to the models of the components and to the framework, and that we have a visual and formal modeling, we are performing in fact visual composition for software.

It is important to point out that the requirements phase is very important for the life cycle adopted and the requirements may not be completely known in the beginning.
of the project. Therefore, a component should be generic, and this imposes difficulties for all the requirements definition. Besides, the component can still be used in other applications that were not considered during its development. Therefore, adaptation mechanisms, such as parameterization must be taken into account.

4 Systematic Modeling

In this section we conceptually describe the model reuse based solution. In Figure 2 it is shown a diagram that illustrates the introduced solution.

The process of reusing building blocks has been extensively used in engineering to efficiently develop new designs and artifacts. The reuse promotes resource savings, such as money and time, besides reducing the possibility of human mistakes. In the last years several researches in the context of software engineering have been developed in order to define techniques and methods to turn the software development process as similar as other engineering processes. One of the solutions studied nowadays is known as components. Based on such solutions, a software system can be developed starting from components used as building blocks. Therefore, the importance of the reuse in the developments of complex systems in very high, specially in the case of component-based software development.

As said in the introduction, during the development of several projects it has been verified that within a specific application domain always exist common characteristics. Those characteristics can be any artifact from code to models describing them based on a formal language. In this work we are dealing with Coloured Petri Nets models.
as artifacts to be reused. It is in the identification of those characteristics inside of a
specific context that the reuse assumes an important role. First, because once identified
something that was probably already made it is possible to search a candidate with or
without adaptations. Second, once an artifact that can be reused is developed it can be
available for possible future reuse.

Once identified a piece of a model that might already have been developed, a search
may be performed. As shown in Figure 2 this activity is supported in the recovery phase.
Then, a recovered model may need some adaptation before being reused, as shown in
the figure in the second phase of the reuse process. Thus, the model is ready for reuse,
the third step is to integrate the model with others to develop the global model. Finally
it is necessary to perform the use verification of the integrated models, as a last stage in
the reuse process in order to verify if the integrated models were correctly used.

In the case of formal methods, it is possible to develop automatic manipulation
methods and techniques for all the phases presented. In fact, all the steps shown in the
Figure 2 were implemented and are fully automatic.

For the development of this work we used the Design/CPN tool set for the develop-
ment and analysis of the models. We used ASK/CTL [6] library to do model checking
[18,7,8,9]. The specifications of the properties for the recovery, adaptation, and use
verification are given in temporal logic [16]. While the integration of models is accom-
plished through functions implemented for Design/CPN.

It is important to point out that besides the traditional reuse activities we also con-
siderer a new activity named use verification. It consists in the execution of model
checking with all the models integrated to verify the specific use case. The specific as-
pects related to the maintenance activity for the repository and the adaption are detailed
discussed in [28] and in [17], respectively. In what follows we discuss each phase of the
reuse process with more details.

4.1 Recovery

Having identified the elements of the system, we can use the repository to recover candi-
dates for reuse. That identification is a result of the requirements and architecture of the
system phases as shown in Figure 1. As it was defined in [28], the recovery consists in:
to describe the properties using temporal logic; to verify the properties described against
a meta-model (the repository); if more than a candidate is found evaluate through simu-
lation and descriptions its behavior and characteristics that are not functional to decide
the best candidate to be used in the current design. In the case that was not possible to
find a candidate for reuse the model must be developed.

After the recovery of the models of components, the next step may be the integration
of them in a framework, or their adaptation. Notice that we can perform the recovery
of all the models before continuing with the integration or adaptation, or to recover a
model and integrate it before recovering another one. This is a designer decision and
has no impact on the introduced solution.

In the following we give an explanation of how the search is made. For the CPN
shown in Figure 3, that models a stack, the following formula defines the behavior of a
stack of integers, that is: if a there is a 4 and a 2 in the stack (place BUFFER), when an
item is removed the 4 will remain on the stack.
fun PAn=Mark."BUFFER" 1 n = ((1,[4,2])!!empty));
fun PBn=(Mark."RESULT" 1 n=((1,value 4)!!empty));
val formula=POS(AND(NF("buffer",PA),
EXIST_NEXT(NF("result",PB))));

The proposition \( PA \) specify that there exists a marking \( n \) with one token \([4,2]\) in place \( \text{BUFFER} \), instance 1. The same reasoning can be applied to \( PB \) but the investigated place is now \( \text{RESULT} \) and the expected token is 4. The string \( \text{page} \) is used as a parameter to the real page name of the model. In a hierarchical CPN model a page is one node in the hierarchy. Such node is indeed a CPN model, together with special constructions to connect it in the hierarchy. For technical details refer to [22].

Now, we are going to analyze the formula that uses \( PA \) and \( PB \). \( NF \) specify a node formula, that is, the formula refers to the nodes of the strongly connected components (SCC) graph [23,2]. This graph is calculated based on the occurrence graph of the model, that is the representation of the state space for this model. The reserved word \( \text{EXIST}_\text{NEXT} \) means that the model checking algorithm will proof the formula in \( NF \) to the next node concerning the current one. The next node can be any one in any way since it is directly connected to the current node. The operator \( \text{AND} \) is a well known boolean operator and \( \text{POS} \) means that it is possible to find at least one way in the graph that satisfies its argument.
The formula \( \text{formula} \) is evaluated to true if, and only if, there exists at least one way in the graph where \( \text{PA} \) is true in a node and \( \text{PB} \) is true in a node that is an immediate successor of the first in any possible way in the graph.

4.2 Adaptation

When a model is recovered from the repository it might not satisfy completely the needs or properties for a given domain. For instance, a recovered model may have the behavior of an infinite capacity stack and in the specific model being designed the capacity might be restricted to a certain number of elements. Therefore, depending on the use case it can be adapted.

The adaptation technique used and developed in [17] is used to automatically adapt a Coloured Petri Net model. The technique consists in writing properties in temporal logic for the behavioral restrictions, then verify if the model can be adapted to satisfy such properties, and finally adapting the model. The first step is related to the needs of the designer and are described using the temporal logic language ASK-CTL. The verification is accomplished by model checking. In the case the adaptation is possible, it is performed by functions implemented in Design/CPN in CPN/ML [5].

In the following we present the ASK/CTL predicate the express the need to bound the stack data structure capacity to at most two elements.

\[
\begin{align*}
\text{fun MaximumLimit(n) = (let val list_s = ref ""; val list_c = ref [] in list_s := (st_Mark.Fork-Lift’BUFFER 1 n); list_c := explode(!list_s); if (count(fc(!list_c, #$\{\$\}, 0) < 3) then true else false end);}
\end{align*}
\]

For this predicate, \( \text{BUFFER} \) is the place of the stack model that has the list with the inserted items. In the following we present the ASK/CTL formula that express the property that indicate a maximum limit of two items for the stack structure.

\[
\text{POS(\neg(\text{NF(\text{"It is possible to limit buffer?"}, MaximumLimit))})}
\]

4.3 Integration

After a model is obtained or recovered it can be integrated in the framework. The integration is linked to the framework. It defines how and where components can be connected, how they are related to each other and with the framework, and possible integration restrictions. The framework depends on the specific domain being considered. For each domain it should be possible to develop a specific framework. The integration consists in the selection of a file name with the diagram to be integrated in the model of the framework, the integration environment creation and the integration to the model. The integration environment consists of the places, transitions and arcs, of the Coloured Petri Net model, as well as the respective inscriptions. While the integration consists in the definition of the hierarchy of the model. All of these integration functionalities are fully implemented in the Design/CPN tool.

Now, we present a description of the integration algorithm. Initially the designer is asked the name of the file with the diagram to be integrated in the framework. Then, some functions will be automatically executed to build the integration environment, it is, the places, transitions, arcs, and its respective names, color sets, and inscriptions. The next step is the definition of the input
Algorithm 1 Model Integration

1: Select the model file
2: Make places, transitions and arcs
3: Define the ports
4: Define the substitution transition
5: Associate the sockets to the ports
6: Adapt the port colors to sockets colors
7: Insert model declarations in the global declaration node

and output ports in the diagram under integration. After this step, the substitution transition will be defined, and the sockets in the superpage will be associated to its respective ports, previously defined in the subpage. The last step is to select the box with the model declarations, in the model page, to define the ports colors based on the sockets colors, and to append this information in the global declaration node.

In a CPN hierarchy, a transition can represent another page. Thus, the represented page is called a subpage, and the page that contains the transition (substitution transition) is called a superpage. The input and output places of the transition are called sockets. The input places of the subpage are called input ports, and the output places of the subpage are called output ports. Historically, the data types of a CPN model is called colours. Therefore the colours of the input and output ports must be consistent with sockets.

The file selection needs the user interaction, while all other steps are fully automatic executed. For the definition of the algorithm described above some restrictions must be satisfied:

– Unique page name;
– The prefix in the color sets names indicates the page name;
– The suffix in the place names indicates if or not it is a port;
– The auxiliary box with model declarations is in dot-dashed line pattern and is unique.
– Declarations of the port places must be the first ones in the auxiliary box;

The first restriction guarantees that the page name for the integrated model is unique, and is a designer responsibility. The model being integrated must have its color set names prefixed with page name. Another restriction is about port places. The input and output places must have names suffixed by IN or OUT, respectively.

The last integration restriction is that in the model being integrated must exist an auxiliary box with dot-dashed line pattern. This box must contain all the declarations for this page. The declaration of the ports color set must be the first ones in this box. This is necessary to guarantee the consistency of the color sets for the integrated model. The final step is to append the contents of the auxiliary box, with the adapted ports color sets, in the global declaration node.

4.4 Use Verification

Besides the activities already discussed we also consider another one named use verification. Such activity is considered because when we model based on reuse it is necessary to guarantee that the semantics of the resulting model satisfies the individual semantics of the reused models.

Therefore, the use verification consists in performing model checking in the framework including the individual models to be verified already integrated. The use verification is performed on the entire model and guarantees that the original specifications were preserved, or the components are being correctly used. Such use verification can be seen as contract verification in the context of components. It is important to point out that a new framework is necessary for a new modeling domain. The construction of such frameworks cannot take into account or violate the specific functionalities of each individual model to be integrated by the reuse process. On the other hand models can be reused in several different domains, and the level of the interface
between the framework and the models of the components can be modified for each domain to satisfy the specific use situations of each model. Then, both the definition of a new framework as well as possible changes in the interface may result in an incorrect use case of the integrated models. Therefore the use verification activity has to be introduced in the reuse process.

Another reason for the definition of the use verification activity is when no model can be recovered from the repository, then it is necessary to build or obtain it by another means. When it is modeled or obtained the model can then be inserted into the repository. It is important to point out that only verified models are stored in the repository. Therefore the use verification may be used to validate a new model before inserting it in the repository.

The use verification consists of executing the occurrence graph tool of the Design/CPN; then select the file that contains the specifications of the properties of the model to be verified and execute the model checking for the framework including all the models already integrated. If the properties are still valid in the final model then the use verification is complete. Otherwise the designer is informed that some mistake exists in that specific case of use for that model.

Algorithm 2 Use Verification

1: Run the occurrence graph tool
2: Select the file with the specifications to be verified
3: Run model checking

In the following we present a description of the algorithm for use verification. Initially, the designer must execute the occurrence graph tool in the Design/CPN. The next step is to select the file that has the properties specifications for the model to be verified. These properties are described by means of a temporal logic called ASK/CTL. The algorithm executes at this point the model checking in the framework including all the models already integrated. If the properties was kept in the resulting model the use verification is successful. In the opposed case the designer receives one warning that there exists an error in this specific use case. In this case the error is not pointed neither corrected by the procedure of use verification. Therefore it is the designer responsibility to find out and to fix the possible errors.

The steps one and two must to be done manually by the user, while the third one is automatically executed based on the specifications contained in the file indicated by the designer, that receives a message saying if the properties was satisfied or not.

The implementation of the ASK/CTL model checking algorithm operates on the Strongly Connected Components (SCC) graph [23,2]. This graph must be generated after the generation of the occurrence graph, still using occurrence graph tool in Design/CPN. When the SCC graph is generated and the ASK/CTL library loaded the use verification is performed. To do this it is necessary to specify the properties we want to prove. At this moment the designer is asked for the file with the specifications. After the designer indicates this file the model checking is performed. The following code fragment implements the use verification.

```plaintext
useCase := eval_node formula InitNode; if (!useCase = true)
    then DSUI_UserAckMessage("Correct Use")
    else DSUI_UserAckMessage("Incorrect Use");
```

The first line command executes the model checking for the formula starting at the initial node of the SCC graph. The command if decides the message to be shown to the designer in case of success or not.

1 An occurrence graph is a representation for the state space for a Coloured Petri Net.
5 Conclusion

In this paper we introduced and discussed a reuse process for Petri nets systematic modeling. Based on such process, the modeling does not need to be developed from scratch, but using models of components and an integration framework. The verification of properties, based on simulation or model checking, increases the dependability on the resulting model. Besides, we have the formal definition for the contracts of the components, separate from the models. With that we can perform verification of the contracts, besides foreseeing the behavior of the system.

The support for the development of systems based on components with a systematic solution and formal methods introduced in this work is a contribution for the consolidation of the discipline of component based software engineering.

Currently we are working on strategies to tie the modeling phase with the implementation phase to make possible automatic code generation based on formal models. Also, we are planning the development of techniques, methods and tools for the modular verification of properties. With modular reasoning it is not necessary to reason about a complete model, but instead one can reason based on individual components.

The application examples discussed in this paper helped us to validate the approach. Particularly, the multi-agent system example have been important to verify the application of the methodology in a practical way. Currently we are studying the integration of component-based software engineering with multi-agents systems in order to develop such kind of systems with components.

References


