Tesis de Licenciatura en Ciencias de la Computación

---

# Un Verificador de Escenarios Arquitectónicos en Tiempo de Ejecución usando Redes de Petri Coloreadas

A Verifier of Runtime Architectural Scenarios using Colored Petri Nets

---

Tesistas

Germán Aníbal Gómez
ggomez@dc.uba.ar

Marcelo Oks
moks@dc.uba.ar

Directores

Dr. Sebastián Uchitel
suchitel@dc.uba.ar

Dr.Victor Braberman
vbraber@dc.uba.ar

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales

DEPARTAMENTO
DE COMPUTACION
Facultad de Ciencias Exactas y Naturales - UBA

30 de mayo de 2012

# Dedicatoria

Dedico esta tesis de licenciatura a mi familia y a mis amigos

**Germán A. Gómez**

Dedico esta tesis de licenciatura a mis padres por apoyarme inclusive cuando están en desacuerdo, a mi esposa por su amor y por aguantarme hasta en mis peores días y a mi hija por sus sonrisas que me iluminan hasta en el día más oscuro.

**Marcelo Oks**

# Acknowledgements

We would like to thank our thesis directors for giving us the necessary technical and emotional support to carry out this piece of work. We also thank them for their patient, their pragmatic approach, their availability to us and their encouragement to finish it

**Marcelo Oks and Germán A. Gómez**

First of all I want to express my gratitude to God for making it possible and for giving me the opportunity to access to high-quality universitary education, I really feel privileged for that. I want to thank Julian for his wise words and motivation to help me finish my thesis. To my mother who gave me a solid example of responsibility and fulfilment, and who also, with all her respect to me and patient, usually asked me for the date of my graduation. I want to thank my wife Carolina for her unconditionality and support.

Many other people may deserve my thanks and gratitude and I apologize to those who I have not explicitly mentioned here due to my lack of memory.

**Germán A. Gómez**

Many people deserve my thanks and gratitude for making this work possible. I apologize in advance to those who I will not explicitly mention here due to my lack of memory.
First, I would like to thank to my parents for their support, even when we disagree. To my wife Solana for her love and for bearing me even on my worst days. And to my daughter Camila who lights my day with every smile.

**Marcelo Oks**

# Resumen

Esta tesis se basa en el proyecto ***DiscoTect*** desarrollado por David Garlan y su grupo de investigadores de la de la Universidad de Carnegie Mellon cuyo objetivo es descubrir la vista C&C (de componentes y conectores) de la arquitectura actual de un sistema en ejecución. Para ello definen ***reglas de mapeo*** en el lenguaje ***DiscoStep*** que convierten eventos de ejecución en eventos de arquitectura con los cuales se va construyendo la arquitectura del sistema.

Dada una un vista C&C de la arquitectura de un sistema en ejecución, pueden plantearse diferentes escenarios en términos de los constructos de esta vista respecto de la configuración actual de la arquitectura. Un escenario podría especificar las cantidades de componentes y conectores, el número y tipo de conexiones entre componentes, la ejecución de una componente particular, etc.

En nuestra tesis desarrollamos una técnica que permite saber si un determinado ***escenario arquitectónico*** en términos de componentes y conectores, se cumple en base a las ***reglas de mapeo*** definidas en ***DiscoStep*** para una arquitectura. Y en caso afirmativo, encontramos la traza mínima de eventos de ejecución que deben ocurrir en el sistema en ejecución para que así sea.

# Abstract

This thesis is based on the DiscoTect project developed by David Garlan and his group of researchers from the Carnegie Mellon University whose aim was to discover the C&C (components and connectors) view of the current architecture of a running system. To do that they define mapping rules in the DiscoStep language that transform runtime events into architecture events which are used to build the system architecture

Given a C&C architectural view of the architecture of a running system, we could state different scenarios in terms of the constructs of this view respect to the current configuration of the architecture. A scenario may specify the amount of components and connectors, the number and type of connections between components, the execution of a component, etc.

In our thesis we developed a **technique** to know if a given ***architecture scenario,*** in terms of components and connectors, is fulfilled based on the ***DiscoStep mapping rules*** of an architecture. And if this is the case, we find the minimum trace of ***runtime events*** that should happen in the running system to do so.

# Contents


1. Introduction ........ 10
   1.1. Motivation ........ 10
   1.2. About this work ........ 10
   1.3. Thesis objective ........ 11
      1.3.1. Contributions ........ 11
   1.4. Related work ........ 11
   1.5. Thesis structure ........ 12
2. Preliminaries ........ 13
   2.1. Colored Petri Nets ........ 13
   2.2. DiscoTect: Architecture Discovery ........ 15
      2.2.1. Instrumentation ........ 16
      2.2.2. Runtime events ........ 16
      2.2.3. DiscoSTEP Language ........ 17
         2.2.3.1. Declaration of input and output event types ........ 17
         2.2.3.2. Architectural rules definition ........ 18
         2.2.3.3. Composition of two rules ........ 19
   2.3. Model Checking ........ 20
      2.3.1. The SPIN model checker ........ 21
         2.3.1.1. Promela language ........ 22
   2.4. Symbolic Execution ........ 22
3. The Scenario Verifier ........ 24
   3.1. Architecture of the Scenario Verifier ........ 26
4. The technique ........ 27
   4.1. Steps of the technique ........ 27
   4.2. Model a set of DiscoSTEP architectural mapping rules as a CPN ........ 28
      4.2.1. Translation rules ........ 28
      4.2.2. Rules composition ........ 33
   4.3. Specify the CPN model in Promela and C code ........ 38
      4.3.1. Places ........ 38
      4.3.2. Tokens ........ 39
      4.3.3. Colorsets ........ 41
      4.3.4. Transitions, transition guards, code segments and arc inscriptions 42
         4.3.4.1. Proctype declarations ........ 43
         4.3.4.2. Transition occurrence ........ 43
         4.3.4.3. Token consumptions from input places of the transition ........ 44
         4.3.4.4. Transition guard ........ 45
         4.3.4.5. Transition code segment ........ 45
         4.3.4.6. Token additions into output places of the transition ........ 46
         4.3.4.7. Double-oriented arcs ........ 47
      4.3.5. CPN to Promela Conversion Algorithm ........ 48
         4.3.5.1. Places ........ 49
         4.3.5.2. Tokens ........ 49
         4.3.5.3. Colorsets ........ 49
         4.3.5.4. Transitions ........ 49
   4.4. Specify an architectural scenario in Promela ........ 52
   4.5. Model check the Promela specification of the CPN ........ 52
      4.5.1. The init process ........ 52
      4.5.2. Stub ........ 53
   4.6. The Assumption Verifier ........ 54
   4.7. Summary of the key challenges raised when specifying and verifying the CPN 56
   4.8. Run the SPIN Output Analyzer ........ 57

4.8.1. The report generator ........ 57
5. Conclusions ........ 60
5.1. Concluding remarks ........ 60
5.2. Potencial applications ........ 61
5.3. Future work ........ 63
6. References ........ 65
Appendix A ........ 67
DiscoSTEP client-server mapping rules ........ 67
Appendix B ........ 69
Concrete syntax of DiscoSTEP Language ........ 69
Appendix C ........ 70
A client-server architecture example ........ 70
The CPN specification ........ 70
The architecture scenario specification ........ 70
The Assumption Verifier ........ 70
General files ........ 71

# List of Figures

Figure 1 .......... 14
Figure 2 .......... 14
Figure 3 .......... 15
Figure 4 .......... 21
Figure 5 .......... 26
Figure 6 .......... 30
Figure 7 .......... 32
Figure 8 .......... 33
Figure 9 .......... 34
Figure 10 .......... 35
Figure 11 .......... 37
Figure 12 .......... 42
Figure 13 .......... 47
Figure 14 .......... 64

# 1. Introduction

## 1.1. Motivation

Given a component and connector software architecture we could imagine many relevant scenarios it should fulfil at architectural level. A scenario could specify, for example, characteristics of an architecture configuration such as the amount of components and connectors it should have, the connections between components carried out in runtime, the execution of a component, etc. It would be really useful and interesting to know what should happen at runtime level to fulfill the proposed scenario. In this way we could take advantage of it at the architecture design time or coding time during the development process or after it to evaluate the level of conformance between the developed system and the architecture. So, we present a technique to find, if there exists, the minimum trace of runtime events that maps into a given architectural scenario specification.

## 1.2. About this work

Our research is mainly based in [4] which is about the discovery of a system architecture at runtime. To address it, the compiled implementation of a system is instrumented using aspects which emit ***runtime events*** (in the form of XML elements) when a specific part of the system is executed. These runtime events are captured and consumed by a runtime engine called ***DiscoTect*** which generates architectural events. To do that, ***DiscoTect*** also takes as input a file called ***DiscoSTEP Mapping Specification*** which contains rules written in ***DiscoSTEP*** language which specify how to map these runtime system-level events into architectural ones, and are formally defined with a ***Colored Petri Net***. Architectural events are then fed to an ***Architecture Builder*** that incrementally creates a model of the system architecture.

Our work is based on the ***DiscoSTEP Mapping Specification*** (or directly ***mapping rules*** from now on) mentioned in the above paragraph. These rules establish mappings between runtime events and architectural ones. In [4] a way of translating these mapping rules into a ***Colored Petri Net*** (***CPN*** from now on) is provided. We extend this translation and show a generic way of specifying ***CPN*** constructs in Promela, the verification modeling language understood by the well-known model checker, ***SPIN***. In this way, the mapping rules can be model checked together with an architectural scenario specification returning, if there exists, the trace of runtime events that fulfills this scenario. While model checking, we may find a set of constraints related to each event instead of finding just one, and so there may be more than one possible configuration for the trace. To cope with it, we decided to use a ***symbolic execution*** approach, so we accumulate all these constraints and finally we verify them in order to know if there exists such trace of runtime events that maps with the proposed scenario. If such trace effectively exists, the presented technique finds the minimum one, considering the size of a trace as the amount of instructions included in it; so the minimum trace is the one with the least amount of instructions.

# 1.3. Thesis objective

The aim of the research is to develop a **technique** to find, if exists, the minimum trace of ***runtime events*** that should be generated by a running system to fulfill a given ***architectural scenario*** specification regarding the system´s components and connectors, and based on their ***mapping rules*** written in ***DiscoSTEP***.

## 1.3.1. Contributions

We identified in our work the following contributions:

1. We present a general form of modeling mapping rules written ***DiscoSTEP*** as a Colored Petri Net. Note that in [4] Garlan presents a basic idea of this, we deepen and extend this.

2. We present a general way of specifying the Colored Petri Net in Promela, the modeling language of the model checker ***SPIN***.

3. We create a stub that models an instrumented system and simulates random sequences of execution events.

4. We support the verification of architectural scenarios in terms of type and quantity of components and connectors, and relationships between them.

5. We developed an Assumption Verifier that checks all the conditions collected during ***SPIN*** verification as a consequence of the adoptions of the ***symbolic execution*** approach. Note that this Verifier is independent of the architecture characteristics.

6. We developed a tool that, in case of effectively finding a minimum trace of runtime events that complies with the architecture scenario, it interprets the output trail thrown by ***SPIN*** during verification and generates a simple report that contains a human readable version of the this trace, the conditions evaluated and information analysis results.

7. Finally, we present a technique that gathers all these above artifacts to provide a general way to cope with the verification of C&C architecture scenarios based on ***DiscoSTEP*** mapping rules specification.

# 1.4. Related work

This work is mainly based in [4], [5] and [6]. We take as starting point the ***mapping rules*** specified in ***DiscoSTEP*** for the components and connectors of a system architecture (C&C view).

Additionally we decided to adopt a ***symbolic execution*** approach based on [11] and [12] so that when executing the CPN using ***SPIN*** verifier to find the minimum trace of runtime events that complies the architecture scenario, all the implied

conditions are accumulated to be evaluated later instead of being evaluated when found.

The theoretical background about CPN constructs and modeling aspects were foundational for our work, and we based on [1], [2] and [3] for this.

It was also very useful for us, some work about ***SPIN*** model checking taken from [7] to [10].

## 1.5. Thesis structure

In Chapter 1 we give a general introduction to our work.
In Chapter 2 we present some base knowledge needed to understand our work.
In Chapter 3 we show the components of the Scenario Verifier.
In Chapter 4 we describe the full technique that we developed as a part of this thesis. Here we explain the passages from DiscoStep mapping rules to its corresponding CPN and from it to its Promela specification.
In Chapter 5 we show some potential applications of our work.
In Chapter 6 we report our final conclusions and future work
In Chapter 7 we present the work in which we based our thesis separated in themes.
In Appendix A we show the DiscoStep mapping rules for a client-server architecture.
In Appendix B Chapter 11we show the concrete syntax of DiscoStep Language.
In Appendix C we enumerate and explain all the files used in the implementation of the scenario verifier, the Promela specification of the CPN for a client-server example including the architecture scenario.

# 2. Preliminaries

In the following subsections the theoretical aspects involved in our work are explained. Then the outline of this document is presented.

## 2.1. Colored Petri Nets

A Colored Petri Net (also known as CP-net or CPN) is a graphical oriented modeling language for design, specification, simulation and verification of systems.

The development of Colored Petri Nets has been driven by the desire to develop a modeling language – at the same time theoretically well-founded and versatile enough to be used in practice for systems of the size and complexity found in typical industrial projects. To achieve this, CPNs combine the strength of Petri nets with the strength of programming languages. Petri nets provide the primitives for describing synchronization of concurrent processes, while programming languages provide the primitives for definition of data types and manipulation of their data values. Colored Petri Nets are an extension to Petri nets with the added possibility of setting a value to a Token. Furthermore in PN the Tokens are indistinguishable.

It is in particular well-suited for systems in which communication, synchronization and resource sharing are important. Typical examples of application areas are communication protocols, distributed systems, imbedded systems, automated production systems, workflow analysis and VLSI chips
The ellipses and circles are called **places**. They describe the states of the system. The rectangles are called **transitions**. They describe the actions. The arrows are called **arcs**. The **arc expressions** describe how the state of the CPN changes when the transitions occur. Each place contains a set of markers called **tokens**. In contrast to low-level Petri nets (such as Place/Transition Nets), each of these tokens carries a data value, which belongs to a given **type**. Token values are referred as token **colors** and we also refer to data types as **colour sets.** Tokens of a CPN are distinguishable from each other and hence “colored” – in contrast to ordinary low-level Petri nets which have “black” indistinguishable tokens. A state of a CPN is called a **marking**. It consists of a number of tokens positioned on the individual places. Each token carries a value which belongs to the type of the place on which the token resides. By convention, initial marking is underlined, next to the place. When the specification of the initial marking is lengthy, we may omit the underlining.

The pre-set of a transition ***t***, pre-set(***t***), is the set of its input places and its post-set, post-set(***t***), is the set of its output places. Analogously the pre-set, pre-set(***s***), of a place ***p*** is the set of its input transitions and its post-set, post-set(s), is the set of its output transitions.

The weight in an incoming arc of a place indicates how many tokens are going to be added if the connected transition occurs. Analogously the weight in an outgoing arc of a place indicates how many tokens are going to be removed if the connected transition occurs.

A Petri Net is said to be ordinary if the weight of all its arcs is 1. The absence of weight in an arc implies it has weight 1.

A finite capacity Petri Net is that in which there is a maximum of tokens defined for each place.

During the execution of a CPN each place will contain a varying number of **tokens**. Each of these tokens carries a data value that belongs to the type associated with the place.

Let's see some examples:

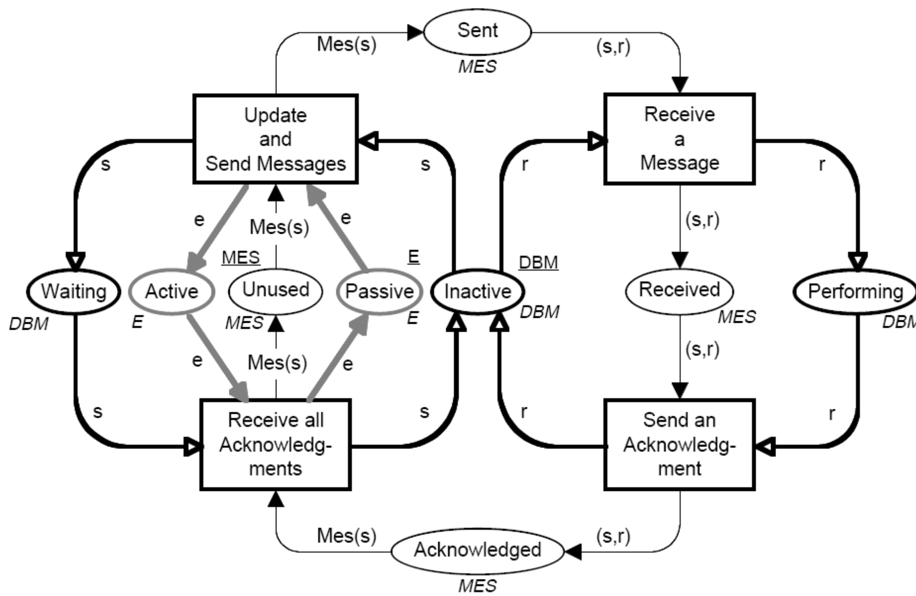



Figure 1

**Example Diagram**

The diagram below has an example of each inscription and region we will discuss in this chapter. The global declaration node is not an inscription, but is included because it is an essential part of any model.

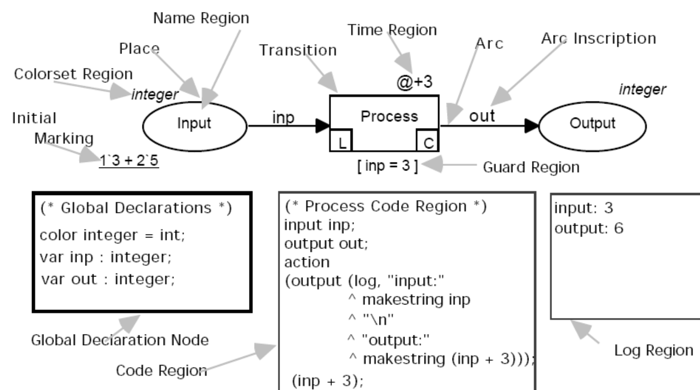
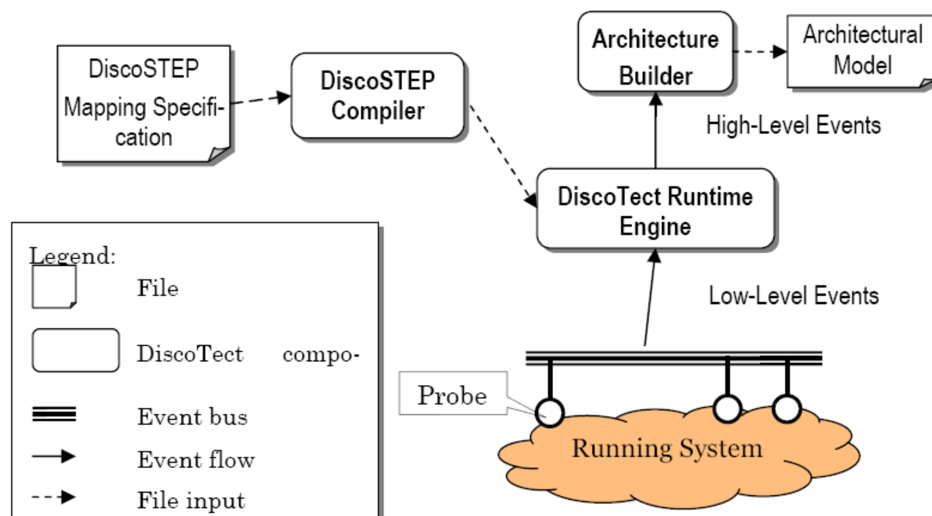



Figure 2

## 2.2. DiscoTect: Architecture Discovery

A relatively unexplored, technique is to determine the architecture of a system by examining its runtime behaviour. The key idea is that a system's execution can be monitored. Observations about its runtime behaviour can then, in principal, be used to infer its dynamic architecture. This approach has the advantage that it applies to any system that can be monitored, it gives an accurate image of what is actually going on in the real system, it can accommodate systems whose architecture changes dynamically, and it imposes no a priori restrictions on system implementation or architectural style.

A technique to solve the problem of dynamic architectural discovery for a large class of systems was developed. The key idea is to provide a framework that allows one to map implementation styles to architecture styles. This mapping is defined conceptually as a Colored Petri Net [1] that is used at runtime to track the progress of the system and output architectural events when predefined runtime patterns are recognized. Thus the mapping provides a way to identify when a program performs "architecturally significant" actions that produce architectural structures. An important additional feature of the approach is the ability to reuse such mappings across systems. In particular, they exploit regularity in implementation and architectural styles so that a single mapping can serve as an architectural extractor for a large collection of similar systems, thereby reducing the cost of writing each abstraction mapping, while still providing flexibility. They implemented a tool called ***DiscoTect***, and the ***DiscoSTEP*** language used for specifying mappings. Then they outline a formal semantics for ***DiscoSTEP*** that specifies its meaning in terms of Colored Petri Nets.




(This figure is taken from [4])
Figure 3

### 2.2.1. Instrumentation

In ***DiscoTect***, events such as method calls, CPU utilization, network bandwidth consumption, memory usage, etc. are captured. To generate them, the running system is probed, or instrumented; for which it is used ***resource monitoring tools***, or ***code instrumentation tools*** such as AspectJ and AspectC++ that allow to inject code into the target system.

These frameworks implement Aspect Oriented Programming in Java and C++. Using AOP to generate events is clean and less invasive than other methods.

Commercial technologies are used to instrument a system to produce runtime events. For Java-based systems AspectJ was used to define ***instrumentation aspects*** that are weaved into the compiled bytecode of the programs. These aspects emit events when methods of interest are entered or exited, and when objects are constructed. Any implementation of AOP, like Spring AOP, could be used to generate the runtime events with the same results as with AspectJ.

### 2.2.2. Runtime events

In the case of instrumentation, the aspects mentioned in the section before can reflectively retrieve information about the runtime environment of, for example, a call, to ascertain the calling object, the instance of the object that was called, the argument values and types that were passed to the method, the method signature, etc. The aspects are written to emit ***XML elements*** that conform to a schema expected by ***DiscoTect***.

For example, to instrument the `ChatServer` below, we could weave in aspects to emit events when methods were called and when objects were constructed.

```
public class ChatServer {

	static class ClientThread extends Thread {...}
		public void run() {
		…
		}
	}

	private static Vector clients = new Vector();

	public ChatServer() {
		ServerSocket serverSocket = new ServerSocket(1111);
		while (true) {
			// Wait for clients to connect
			Socket socket = serverSocket.accept();
			new ClientThread(socket, clients).start();
		}
	}

	public static void main(String[] args)throws IOException {
		new ChatServer();
	}
}
```

After running this application, some of the runtime events generated would be:

```
<init constructor_name="ServerSocket" instance_id="10">

<call method_name="ServerSocket.accept" callee_id="10"
return_id="11" …/>
```

The init event is generated when the following sentence is executed:

```
ServerSocket serverSocket = new ServerSocket(1111);
```

The call event was triggered by the execution of the following method call:

```
Socket socket = serverSocket.accept();
```

As you can see, multiple `ClientThreads` can run concurrently, so some of the runtime events will be generated in random order and hence interleaved with each other.

The generated runtime events can be fed into ***DiscoTect*** either in real time or off-line, after the program has completed running. These events are then input into a ***DiscoSTEP*** specification which produces architectural events, generated as a result of processing the runtime events, which in turn are used to produce the software architecture.

### 2.2.3. DiscoSTEP Language

***DiscoSTEP*** is a language to write architectural rules. Basically, a ***DiscoSTEP*** rule takes as input low-level events (generated by a system in runtime) or intermediate events (generated by rules to be fed into other rules) and generates as output architecture events or intermediate events. In the case of architecture events, they are fed to an Architecture Builder that incrementally builds the system architecture. A ***DiscoSTEP*** program has three parts: the declaration of input and output event types used in the rules, the definition of architectural rules and the declaration of rule compositions.

#### 2.2.3.1. Declaration of input and output event types

In a ***DiscoSTEP*** program we must declare which event types are of input and which ones of output, we can do it following this structure:

```
event {
      input{ input_event_type_names }
      output{ output_event_type_names }
}
```

Where *input_event_type_names* is a list of the input event type names and *output_event_type_names* is a list of the output event type names. Let's see an example:

```
event {
      input {
             init;
             call;
```

```
              string;
        }
        output {
              string;
              create_component;
              create_client;
              create_connector;
        }
  }
```

Every ***DiscoSTEP*** program must declare its input and output event types, this declaration is mandatory.

### 2.2.3.2. Architectural rules definition

A ***DiscoSTEP*** architectural rule has the following parts:

```
  rule rule_name {
        input { input_event_declarations }
        output { output_event_declarations }
        trigger {$ conditions $}
        action {$ assignments $}
  }
```

Where

- *input_event_declarations* are the declarations of the input events of a rule, each declaration is given by an input event type and the event name.

- *output_event_declarations* are the declarations of the output events of a rule, each declaration is given by an output event type and the event name. Output events can be architecture events or events to be consumed by other rules because of rule composition.

- *conditions* are a set of conditions on the input events of the rule. If all these conditions are true then the rule is applicable and its action block is fired. Predicates are written in XQuery language. If the trigger block is false then no input event is consumed.

- *assignments* are assignments from new output events to each output event declared in the output block of the rule. Each assignment is preceded by the word *let*. Other assignments can be done, for example, for temporary usage inside the action block. The right member of every assignment is written in XQuery language.

Note that conditions or assignments inside a trigger or an action block must be enclosed between the ‘{$’ and ‘$}’. Event names must be preceded with a ‘$’ symbol.

Input events are of a type such as *call*, *init*, *string*, etc. Events, generated by the system in runtime or by another rule, are bound, if the type matches, to the event names declared in the input block of a rule. When all of these input event names have a binding then the trigger block is evaluated. If it is true, the rule is applicable and the action block is carried out generating new output events which can be architecture events or intermediate events to feed to other rules in rule compositions. Let’s see an example of ***DiscoSTEP*** rules used to create a server component, connect a client and connect it to the server:

```
rule CreateServer {
    input { init $e; }
    output { string $server_id; create_component $create_server; }
    trigger {? contains($e/@constructor_name, "ServerSocket") ?}
       action = {?
        let $server_id := $e/@instance_id;
        let $create_server := <create_component name="{$server_id}"
        type="ServerT"/>;
    ?}
}


rule ConnectClient {
    input { call $e; string $server_id; }
    output {create_component $create_client; create_connector
    $create_cs_connection;
        string $client_id;
    }
    trigger {? contains($e/@method_name, "ServerSocket.accept")
                  and $e/@callee_id = $server_id
    ?}
    action = {?
        let $client_id := $e/@return_id;
        let $create_client := <create_client name="{$client_id}"
        type="ClientT" />;
        (:$concatedIds is an auxiliary variable, not an event. :)
        let $concatedIds := concat($client_id,"-",$server_id)
        let $create_cs_connection :=
                <create_connector name= "{$concatedIds}"
                type="CSConnectorT" end1="{$server_id}"
                end2="{$client_id}" />;
    ?}
}
```

The CreateServer rule creates a server component. The rule declares in its input block that it takes as input a runtime event $e of type init. The output block declares that it generates an architecture event of type create_component called $create_server and an intermediate event of type string called $serve_id. The trigger block has only one condition which states that the constructor_name property of the input event $e contains the string "ServerSocket". The action block has two assignments, first the intermediate event $server_id is assigned the instance_id property of the $e input event and second the architecture event used to create a component, a server in this case, is assigned to $create_server.

The ConnectClient rule creates a client and connects it a server identified by $server_id. The rule can be understood in the same way as the CreateServer rule. Inside the action block a comment is written. Comments are enclosed between '(:' and ':)', if more than one line is written then a ':' must be written at the beginning of the line. Below the comment the concatenation of two strings $client_id, a dash and $server_id is assigned to the $concatenatedIds variable. This variable is not declared in the output block of the rule, so it is not an event, it is just an auxiliary variable to be used inside the action block.

### 2.2.3.3. Composition of two rules

***DiscoSTEP*** rules can be composed, that is to say, intermediate events generated by a rule can be taken as input events by another rule. For instance, the

CreateServer and the ConnectClient rules can be composed via the $server_id event in the way showed below:

```
composition {
      CreateServer.$server_id -> ConnectClient.$server_id;
}
```

The unidirectional binding denoted by -> states that the output event $server_id generated by the CreateServer rule can be consumed by the ConnectClient rule as an input event.

The composition can also be bidirectional denoted by the bidirectional binding **<->.** Let´s see an example:

```
composition {
      CreateServer.$server_id <-> ConnectClient.$server_id;
}
```

In the example above the bidirectional binding states that the rule that takes as input the $server_id event can make use of it without consuming it. In our case the ConnectClient rule uses the $server_id but it does not consume it so this event can be consumed or used by another rule. Note that both rules could take the $server_id event as input but in our example the CreateServer rule does not take this event as input.

When the input block of a rule declares runtime input events and the rule is applicable then these runtime events are consumed by the rule. But when the input block of a rule declares non-runtime input events and the rule is applicable then these events can be consumed or just used by the rule without consuming them, it depends on the type of composition.

The concrete syntax can be found at Appendix B.

# 2.3. Model Checking

Model checking is a widely used formal method for the verification of concurrent programs. The problem with concurrent programs is that the number of possible computations is astronomical, so it seems that exhaustive checking is impractical as a method of gaining confidence in the correctness of the program. In the 1980s, Clarke, Emerson and Sifakis showed that it can be feasible to check all possible computations of a concurrent program. Their key insight was to note that both a concurrent program and its correctness property can be transformed into nondeterministic finite automata (NDFA) and “run” simultaneously. Given the NDFA corresponding to the program and the NDFA corresponding to the negation of the correctness property (expressed in temporal logic), a model checker searches for an “input string” accepted by both automata. If it finds one, the input represents a computation of the program that breaks the correctness claim; therefore, the program is not correct and the computation can be reported as a counterexample to the correctness claim.

Model checking is a model-based, automatic method that, given a finite-state model $M$ of a system and a property $p$, checks the validity of P in M, ie, $M|=p$.

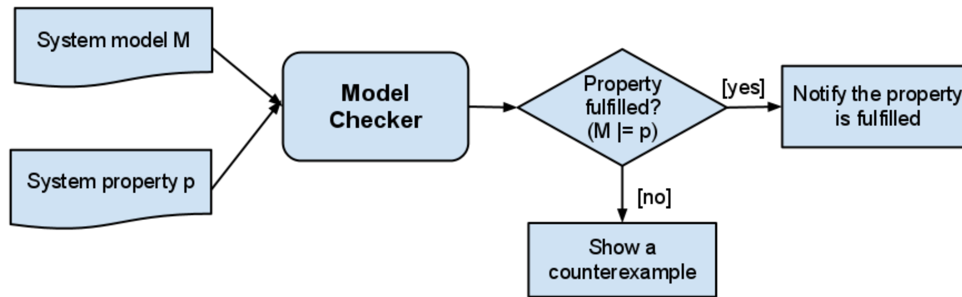



Figure 4

Typically it involves three steps:

1. Create a finite state model of the system design in a formal language.
2. Specify a system property in a formal language.
3. Model-check the model and the property together to verify if the model satisfies the property.

Some of the properties that can be verified with model checking are: deadlocks, race conditions, assertion violations, safety properties (system is never in a "bad" state), liveness properties (system eventually arrives in a "good" state), livelock, starvation, under-specification, over-specification, violations of constraints, etc.

An important ingredient of model checking is an expressive language that can be used for model description. Such a language must have a precise semantics, yet it must also be suitable for its application domain and easy to use.

### 2.3.1. The SPIN model checker

***SPIN*** (**S**imple **P**romela **In**terpreter) is a powerful widely used and proved automata-based model checker particularly for analysing the logical consistency of concurrent systems. ***SPIN*** is nowadays one of the foremost model checkers.

It was written by Gerard J. Holzmann who in 2001 received the ACM Software Systems Award for ***SPIN***. It was originally designed for verifying communications protocols and has evolved since then for more than twenty years. It has become one of the most widely used verification tools. ***SPIN*** is particularly suited for modelling concurrent and distributed systems that are based upon interleaving of atomic instructions.

***SPIN*** can be used as a full **LTL model checking** system, supporting all correctness requirements expressible in ***linear time temporal logic***, but it can also be used as an efficient on-the-fly verifier for more basic safety and liveness properties. Correctness properties can be specified as system or process invariants using assertions, as LTL requirements, or in other ways.

***SPIN*** is commonly used in simulation and verification modes. In simulation mode just one choice in the state-space is made non-deterministically. It allows rapid prototyping with random, guided, or interactive simulations. In the verification mode, full exhaustive validation is carried out using partial order reduction theory to optimize the search, and based on either depth-first or breadth-first search in the state-space.

Given a model and a property specified in Promela, the language understood by ***SPIN***, both are model-checked together. Once the model checker finds a trace that complies with the specified property, it generates a file containing the trail with the information of all the non-deterministic choices made by the model checker. This information could also be read later by the model checker to make simulations over the model.

For our work we also used ***iSpin***, a graphical user interface for ***SPIN***, from which we can check syntax correctness of Promela specifications, run simulations and verifications and other useful things.

### 2.3.1.1. Promela language

As stated before, ***SPIN*** supports a high level modeling language called ***Promela*** (***Pro***cess ***Me***ta ***La***nguage) which allow us to specify system models and properties. Promela also has the ability to embed C code blocks or include C code files in specifications.

Promela models mainly consist of 3 types of objects: processes, message channels, and variables. Processes are global objects. Message channels and variables can be declared either globally or locally within a process. Processes specify behavior, channels and global variables define the environment in which the processes run. Processes send and receive messages through channels and channels can store messages in different ways depending on its type, by default in FIFO order.

Process communication via message channels can be defined to be synchronous (rendezvous), or asynchronous (buffered); mixed specifications are supported. Processes can also communicate via shared memory.

More about Promela, including the complete language reference, can be found at [10].

## 2.4. Symbolic Execution

The idea of ***symbolic execution*** is born from the Symbolic Mathematics. These relate to the use of computers to manipulate mathematical equations and expressions in symbolic form, as opposed to manipulating the approximations of specific numerical quantities represented by those symbols. Such a system might be used for symbolic integration or differentiation, substitution of one expression into another, simplification of an expression, etc. It has uses in software testing under the title of ***symbolic execution*** where it can be used to analyse if and when errors in the code may occur. It can be used to predict what code statements do to specified inputs and outputs. It is also important for considering path traversal. Thus Model Checking techniques have used ***symbolic execution*** for more than 15 years.

Nowadays ***symbolic execution*** techniques are used in several fields. The techniques have evolved so much that there are model checkers like Zing which

allow specifying a program using Object Oriented Programming combined with ***symbolic execution***.

We based our work in a more primitive idea of ***symbolic execution***. An easy way to understand it is with the following pseudocode example.

```
Function IsMinor (age as Integer)
      If age < 21 Then
            Return True
      Else
            Return False
      End If
End Function
```

In this function we do not need to know the exact value of the age parameter. We just need to know in which of the intervals it is located. The intervals would be [0..20] and [21..∞].

If we now have the following piece of a program:

```
If IsMinor(KidAge) Then
      If KidAge > 21 Then
            testValue = True
      Else
            testValue = False
      End If
Else
      testValue = False
End If
```

Using ***symbolic execution*** we can know that `testValue` is always going to be equal to False.

We used this idea to create a model where each time there is a branch or an assignment an assumption is created with the condition stated. After a trace execution is finished the assumptions are evaluated to see if the set is satisfiable comparing all the assumptions made.

# 3. The Scenario Verifier

First we explain how ***DiscoTect*** [4] works in order to provide the base for understanding the technique defined in this work. ***DiscoTect*** takes as input a file containing ***mapping specification rules*** written in ***DiscoSTEP*** language. ***DiscoTect*** constructs a ***Colored Petri Net*** from these rules which consume runtime events and generate architectural ones as outputs. Architectural events are fed to an ***Architecture Builder*** that incrementally creates a model of the system architecture

In [4] a way of translating the mentioned ***mapping specification rules*** into a CPN is provided. We extend this translation and show a generic way of specifying CPN constructs in a language accepted by a model checker. We start by modeling these rules as a ***CPN model*** and then we specify it in Promela language. Given that our aim is to find a trace of runtime events, we need something to generate them, so we specify a ***stub*** in Promela which is able to produce all possible runtime events. An ***initial marking*** of a CPN sets a configuration of tokens in it determining the number of tokens to be positioned in every place. The ***stub*** is responsible for generating all the different ***initial markings***. An ***initial marking*** of the ***CPN*** models a determined sequence of runtime system events. Furthermore the ***stub*** establishes these ***initial markings*** in the modeled CPN for a fixed amount of tokens. For a same amount of tokens, different random ***initial markings*** are tried simulating different instrumentations, and consequent runtime events, that could be generated by any program. The desired trace must map to an ***architectural scenario*** which specifies a determined situation at high level. This scenario is also written in Promela language.

The three mentioned elements: the ***CPN model***, the ***stub*** and the ***architectural scenario*** are all of them model checked by the model checker ***SPIN*** [10]. During the model checking stage the ***stub*** first sets a random ***initial marking*** and then the CPN execution starts consuming and generating tokens from and into places. Tokens are consumed when mapping rules apply and this happens when certain conditions on runtime events, which means on tokens, are true. While the CPN "executes" these conditions are not evaluated, they are just supposed to be true and are accumulated to be evaluated later. This approach is partly taken from ***symbolic execution*** [12].

Given an initial marking of a CPN for a fixed amount of tokens, a CPN execution finishes when either all the architectural scenario properties are gathered or when all tokens are consumed before reaching the scenario.

In the latter case, the initial marking is discarded and the model checker continues trying with others initial markings (for the same fixed amount of tokens) and different sequences of token consumptions. But if this situation happens for all these different initial markings, then the model checking finishes without having found a trace of runtime events for the architectural scenario. So a new higher amount of tokens must be fixed for the stub and the model checking is restarted, repeating the whole process again and again till either a trace is finally found or all possible traces are inspected and none of them maps into the specified scenario.

Otherwise, in the first case, when the scenario is reached, the set of all the accumulated assumptions (those constraints collected when consuming tokens while executing the CPN) is verified by an ***Assumption Verifier*** which determines the satisfiability of this set. If it is satisfiable then a valid trace of runtime events

that maps into the architectural scenario has effectively been found. The ***Assumption Verifie***r is also model checked together with the ***CPN model***, the ***stub*** and the ***architectural scenario***.

The approach used to find the minimum trace is based on the fact that if a possible trace exists the model checker will find it. The size of the trace, when found, is the same as the amount of tokens of the last initial marking used by the ***stub***. A token consumption models a runtime event execution. So we start with one initial token and we apply model checking to find the trace. If a trace is found then it is minimal because it will have one event, if not we model check again but using one more token and so on. If, for example, a trace is found using five tokens (and so the trace will have five events) we know it is minimal because a trace with four tokens was not found. If the user has a good understanding of the ***CPN model*** and of the ***architectural scenario*** he may predict a more approximated amount of the minimum of necessary tokens. Then he may try with that amount of tokens and if a trace is found he can then start trying with fewer tokens as a way to discover the minimum trace.

As an example, we decided to model and implement the client-server architectural rules shown in Appendix A of [4]. These rules were modeled as a CPN and translated into Promela code in order to be able to model check this model with ***SPIN*** [10]. The architectural scenario was written as a constraint in Promela which must be verified during the whole model-checking. The stub is a Promela process which establishes the initial marking in the CPN

When modelchecking, the CPN is executed until all the tokens are consumed (so a blocking state is reached) or the architectural scenario properties are gathered. When the model checker finds an error, in the model, the searched trace is found. If the modelchecking process finishes without any error then no trace has been found for the initial marking of the CPN. The consumptions of tokens determine certain constraints over their properties which can be verified at that moment or later. As we mentioned before, we have adopted a ***symbolic execution*** approach based on [12] in which, while finding the desired trace, implied constraints (assumptions from now on) are accumulated instead of being verified at the moment of being found. We work with two kinds of assumptions:

- ***Value Assumption***: it constraints a token property to a constant value.
- ***Equality Assumption***: it establishes an equality relation between two properties of two different tokens, properties which may not be necessarily the same.

While modelchecking the CPN, value and equality assumptions are accumulated in two structures, one for each type. As we explained before, once the desired trace is found then all the accumulated assumptions are verified by an ***Assumption Verifier*** that determines the satisfiability of the assumptions set. If the set is satisfiable then a trace of runtime events, that satisfies the architectural scenario for the specified architectural rules, has effectively been found. Otherwise the model checker tries to find another trace and the whole verification process is repeated.

# 3.1. Architecture of the Scenario Verifier

Below a sketch of the C&C view of the architecture is shown:

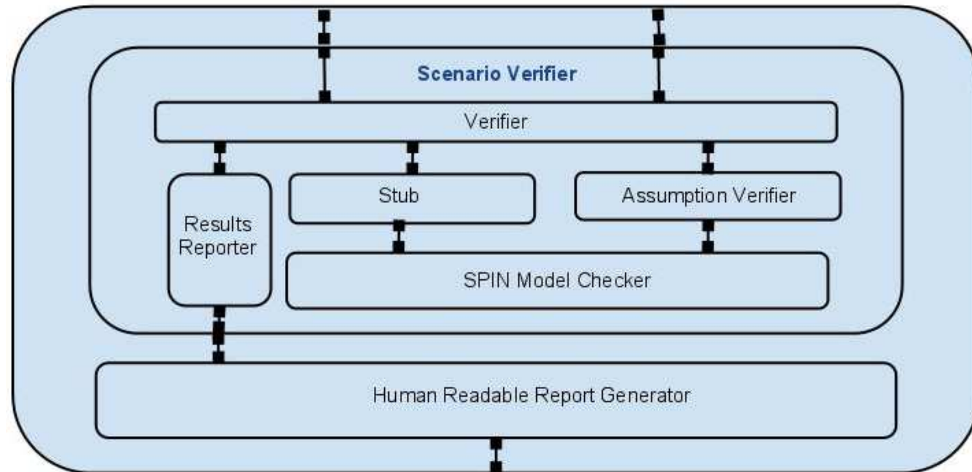



Figure 5

One input is the Promela and C specification of the architectural rules in ***DiscoSTEP*** and the other is the architectural scenario specified in Promela.

The output is a report that includes the trace of runtime events that fulfil the scenario and other results of the analysis.

# 4. The technique

In this section we will explain the full technique which is the base of our work. It combines manual and semi-automatic steps.

## 4.1. Steps of the technique

1. Select or create the set of ***DiscoSTEP*** architectural mapping rules corresponding to the architecture you wish to use.
2. Model the previous architectural mapping rules as a CPN.
3. Specify the CPN model in Promela and C code.
   3.1. Determine some CPN-related constants.
4. Specify an architectural scenario in Promela.
   4.1. Determine some scenario-related constants.
5. Model check the Promela specification.
   5.1. Determine verification parameters of the ***SPIN*** model checker.
   5.2. Start verification.
   5.3. Analyze results:
      5.3.1. If memory was insufficient then determine new values for the verification parameters of the ***SPIN*** model checker and restart model checking. Go to step 5.2.
      5.3.2. If the model checking process finishes without assertions then (if a trace exists) the initial amount of tokens is not enough so it is necessary to increment it in one:
         5.3.2.1. If this new initial amount of tokens is higher than the sum of all quantities chosen in the architectural scenario for the architectural events then it is not possible to find a trace of runtime events that maps the architectural scenario proposed in 4. Go to step 7.
         5.3.2.2. Otherwise go to 5.2 to restart model checking.
      5.3.3. If an assertion (which is not the last assert(false)) is risen then determine new values for the corresponding scenario-related constants. The assertion raised will provide clues to discover which constant has to be modified. Go to step 5.2.
      5.3.4. If the final assertion is raised then a trace has been found and it is minimal.
6. If a trace has been found, run the ***SPIN*** Ouput Analyzer to interpret the ***SPIN*** output trail and generate a human readable trace.
7. End.

In the following sections we explain the main steps involved in the technique. Some of the explanations are based on a set of ***DiscoSTEP*** mapping rules for client-server architecture, see appendix A.

# 4.2. Model a set of DiscoSTEP architectural mapping rules as a CPN

In this section we explain how a set of architectural ***DiscoSTEP*** mapping rules is modelled as a CPN. In [4] a way of translating these mapping rules into a CPN is provided; in this section we extend that explanation adding a detail process for doing this.

## 4.2.1. Translation rules

**(tr1).** Each ***DiscoSTEP*** rule is modeled as a CPN transition.

**(tr2).** Each event declared in the input block of a rule, ***eventType eventName,*** is modelled as a CPN place called ***eventName*** with an associated colorset derived from the event type ***eventType***. The place is then connected to the transition that models the rule with a directed arc from the place to the transition. Finally a declarative variable name is placed as an arc inscription and we declare that its colorset is the same as the colorset associated to the place.

**(tr3).** Each event declared in the output block of a rule, ***eventType eventName,*** is modelled as a CPN place called ***eventName*** with an associated colorset derived from the event type ***eventType***. The place is then connected to the transition that models the rule with a directed arc from the transition to the place. Finally a declarative variable name is placed as an arc inscription and we declare that its colorset is the same as the colorset associated to the place.

**(tr4).** The declaration of colorsets (those associated to places) for the CPN depends on the different ***DiscoSTEP*** event types. An event type can be simple, if it does not have any property, or compound if it has properties of possibly different colorsets or types. A colorset derived from a compound event type is declared as a record with as many fields as properties the event type has. The record field (or property) names remain the same as the properties of the event type and their colorsets or types are derived from the types of the properties of the event type.

**(tr5).** Each event ***e*** generated by the system (runtime event) or by another rule is modelled as a CPN token positioned in the CPN place that models the declared event (in an input or output block of a rule) to which the event ***e*** is bound. Note that the fact of binding the event ***e*** to one event declared, in the input or output parts of a rule, is modelled as the fact of binding a token to the corresponding arc inscription.

**(tr6).** Each ***DiscoSTEP*** trigger block is modelled as a CPN transition guard written in CPN ML[1].

[1] CPN ML is an acronym for Coloured Petri Net Meta Language.

**(tr7).** Each ***DiscoSTEP*** action block is modelled as a transition code segment written in CPN ML.

**(tr8).** The composition of rule output R1.O1 with rule input R2.I2, denoted as R1.O1 -> R2.I2, generates a merging of the place P1 that represent O1 and I2. If the rules are well defined the colorset of O1 and I2 must be the same.

**(tr9).** The dual composition of rule output R1.O1 with rule input R2.I2, denoted as R1.O1 <-> R2.I2, generates a merging of the place P1 that represent O1 and I2. If the rules are well defined the colorset of O1 and I2 must be the same. Then a transition is added from R2 to the new place. This implies that each time R2 is executed a token is consumed from P1 and after R2 execution a new token is inserted in P1. This new token has the same color that the token previously consumed.

**(tr10).** A rule application is modeled as the occurrence of its corresponding CPN transition.

Runtime events are always input events, their declarations appear only in the input block of a rule so they are always modelled as **input places**. Instead architecture events are always output events, their declarations appear only in the output block of a rule so they are always modelled as **output places**. Respect to intermediate events, they are generated by architecture rules to be fed to other rules. Their declarations always appear in the output and input blocks of composed rules so they are always modelled as **intermediate places**, that is to say, input and output places.

Now let's take, as an example, the CreateServer ***DiscoSTEP*** mapping rule:

```
event {
     input {
            init;
     }
     output {
            string;
            create_component;
     }
}

rule CreateServer {
   input { init $e; }
   output { string $server_id; create_component
   $create_server; }
   trigger {? contains($e/@constructor_name, "ServerSocket")
   ?}
      action = {?
      let $server_id := $e/@instance_id;
      let $create_server := <create_component
      name="{$server_id}" type="ServerT"/>;
   ?}
}
```

Let's apply the translation rules to this example and let's see how it is modelled with CPN constructs.

**(tr1)** The rule CreateServer is modeled as a transition called CreateServer.

**(tr2)** The input block of the rule has only one event declaration: ***init $e***, so the transition has an only input place named $e with the associated colorset ***Init*** derived from the event type ***init***. The variable name ***init_event*** is written as arc inscription surrounding the arc that goes from the place $e to the transition CreateServer. We must also state that the colorset of init_event variable is ***Init*** and we do it via this declaration:

```
var init_event :Init;
```

**(tr3)** The output block of the rule has two event declarations: ***string $server_id*** and ***create_component $create_server*** and so the transition has two output places called ***$server_id*** and ***$create_server*** with associated colorsets ***String*** and ***CreateComponent*** respectively. The variable names ***server_id*** and **create_component** are written as arc inscriptions surrounding the arcs that go from the CreateServer transition to the places ***$server_id*** and ***$create_component*** respectively. We now declare the colorsets of these variables with these declarations:

```
var server_id: String;
var create_server :CreateComponent;
```

By now, the CPN looks like this:

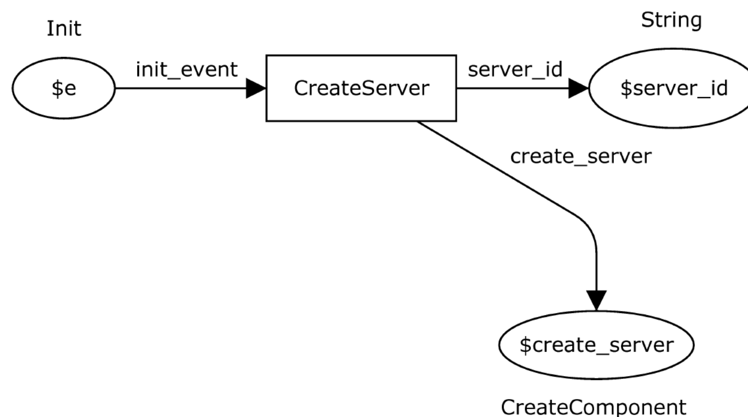



Figure 6

```
var init_event: Init;
var server_id : String;
var create_server: CreateComponent;
```

Note that we haven´t still declared the colorsets ***Init***, ***String*** and ***CreateComponent***, it will be done in the following translation rule.

**(tr4)** The event type string does not have any properties so its derived colorset ***String*** is simple.

```
color String = string;
```

Note that we are just renaming the ML type string.

Events of type init are runtime events. The XML schema of this type is the following:

```
<element name="init">
    <complexType>
        <attribute name="constructor_name" type="string" />
        <attribute name="instance_id" type="string" />
    </complexType>
</element>
```

The colorset ***Init*** derived from the event type init is compound. So it is declared as a record with two fields, constructor_name and instance_id, whose colorsets are String. Let's see the declaration:

```
color Init = record constructor_name :String * instance_id
:String;
```

Events of type create_component are architecture events. The XML schema of this event type is the following.

```
<element name="create_component">
    <complexType>
        <attribute name="name" type="string" />
        <attribute name="type" type="string" />
    </complexType>
</element>
```

So the derived colorset ***CreateComponent*** is declared as a record with two fields, name and type with colorsets String. Let's see the declaration:

```
color CreateComponent = record name :String * type :String;
```

Let's see all the declarations of the CPN together:

```
color String = String;
color Init = record constructor_name: String *
instance_id: String;
color CreateComponent = record name: String * type
:String;

var init_event: Init;
var server_id: String;
var create_server: CreateComponent;
```

Let's continue translating.

**(tr6)** The trigger block of the rule:

**trigger** {? contains($e/@constructor_name, "ServerSocket") ?}

has a condition that states that the property constructor_name of the $e event contains the value of "ServerSocket". Without any loss of generality, this condition is modeled as an equality expression between the constructor_name property of the token and the string value "ServerSocket". The expression is enclosed between brackets and positioned next to the transition. In this rule we make a simplification

in our implementation, supporting a small amount of operations instead of the full XPath stack. This is only due to simplify our work for this thesis and more operations could be added in future work.

**(tr7)** The action block of the rule:

```
action = {?
let $server_id := $e/@instance_id;
let $create_server := <create_component
name="{$server_id}" type="ServerT"/>;
?}
```

carry out the following assignments:

- The **instance_id** property of the **$e** event is assigned to the output event **$server_id.**

- The output event **$create_server** is set as a compound element of type **create_component** with two properties: **name** and **type**. The **name** property is set as the **$server_id** which is in fact **instance_id** property of the **$e** event and the **type** property is set as the string value "**ServerT**".

These assignments are modeled inside the transition code segment as bindings between the values retuned by the action clause and the variables listed in the output clause.

Below the resulting CPN is shown.

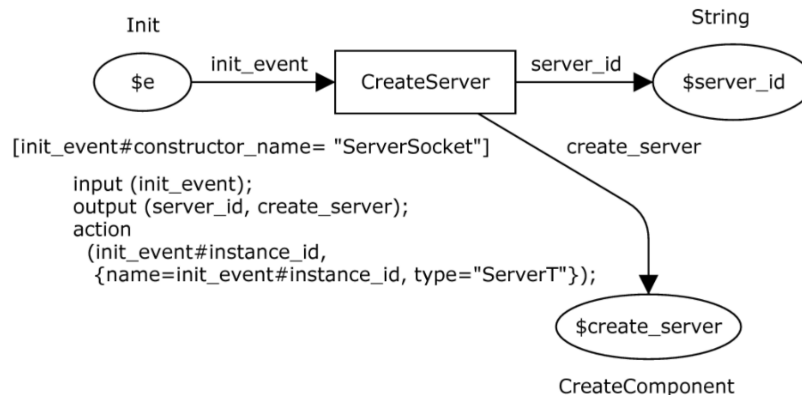



Figure 7

```
color String = string;
color Init = record constructor_name :String *
instance_id :String;
color CreateComponent = record name :String * type
:String;

var init_event :Init;
var server_id: String;
var create_server :CreateComponent;
```

## 4.2.2. Rules composition

Now we explain how to compose two rules. We take as example the CreateServer and ConnectClient rules. The CreateServer ***DiscoSTEP*** rule and its CPN model were already shown. Let's see the ConnectClient ***DiscoSTEP*** rule.

```
rule ConnectClient {
   input { call $e; string $server_id; }
   output {create_component $create_client; create_connector
   $create_cs_connection; string $client_id;}
   trigger {? contains($e/@method_name,
   "ServerSocket.accept")and $e/@callee_id = $server_id
   ?}
   action = {?
      let $client_id := $e/@return_id;
      let $create_client := <create_client
      name="{$client_id}" type="ClientT" />;
      let $create_cs_connection :=
            <create_connector name= concat($client_id,"-
            ",$server_id)
              type="CSConnectorT" end1="{$server_id}"
            end2="{$client_id}" />;
   ?}
}
```

Note that the string event type should also be included in the input block of the declarations of input and output event types for the ***DiscoSTEP*** program.

Below the CPN model of the ConnectClient rule is shown.

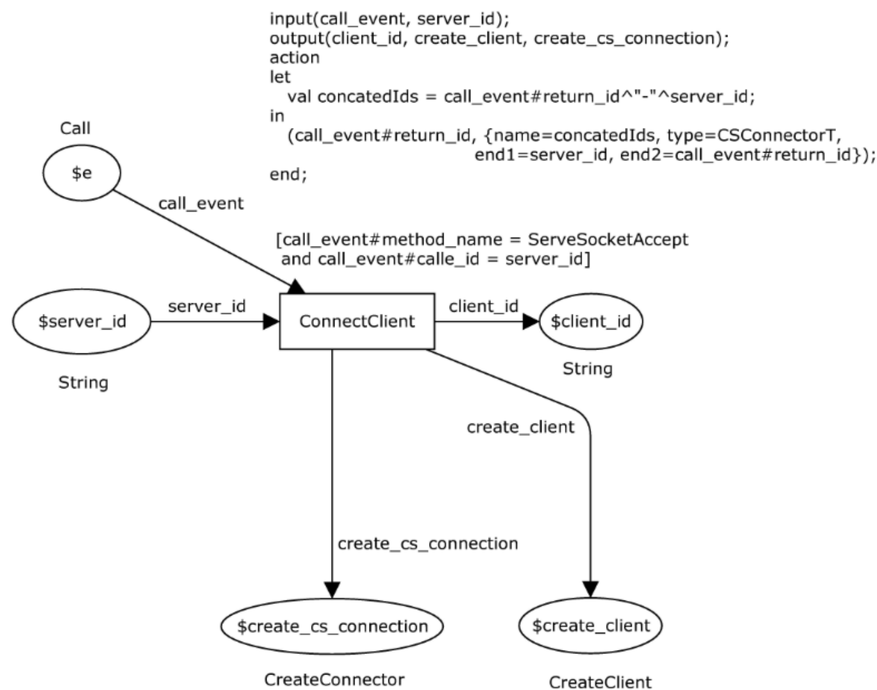



Figure 8

```
color String = string;
color Call = record method_name :String * callee :String
* return_id :String;
color CreateClient = record name :String * type :String;
color CreateConnector = record name :String * type
:String * end1 :String *
             end2 :String;

var call_event :Call;
var server_id, client_id: String;
var create_connector :CreateConnector;
var create_client :CreateClient;
```

The following piece of ***DiscoSTEP*** code states that the CreateServer and ConnectClient rules are composed via the $server_id event. The unidirectional binding denoted by **->** states that the output event $server_id of the CreateServer rule can be consumed by the ConnectClient rule as an input event.

```
composition {
    CreateServer.$server_id -> ConnectClient.$server_id;
}
```

Now we model this composition by applying the translation rule 8 **(tr8)**. The place $server_id is common to both transitions: CreateServer on the left in the diagram and ConnectClient on the right. Note that we together the declarations of both CPNs. Below we show the resulting CPN.

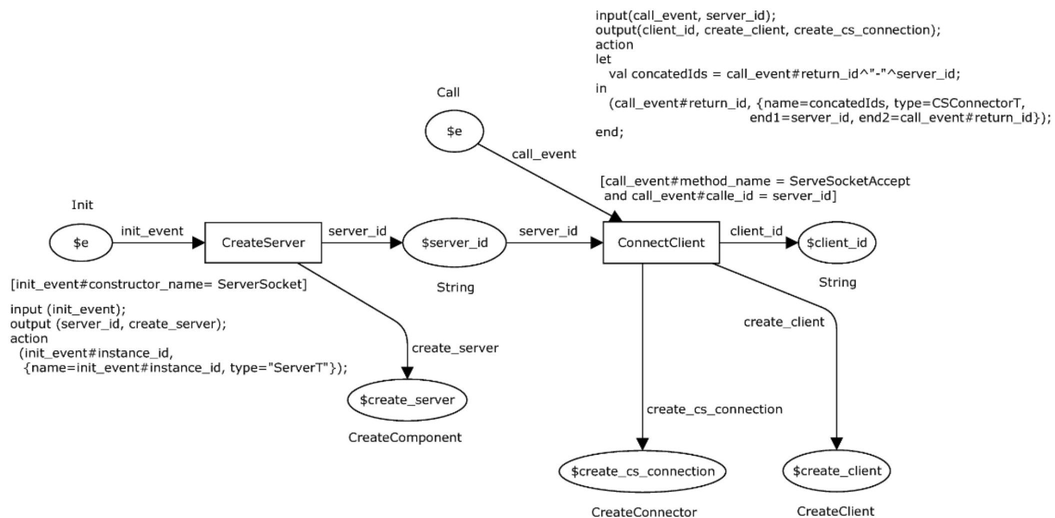



Figure 9

```
color String = string;
color Init = record constructor_name :String * instance_id
:String;
color Call = record method_name :String * callee :String *
return_id :String;
color CreateComponent = record name :String * type :String;
color CreateClient = record name :String * type :String;
color CreateConnector = record name :String * type :String *
end1 :String * end2 :String;
```

```
var init_event :Init;
var call_event :Call;
var server_id, client_id: String;
var create_server :CreateComponent;
var create_connector :CreateConnector;
var create_client :CreateClient;
```

The composition can also be bidirectional denoted by the bidirectional binding **<->.** It states that the rule that takes as input the $server_id event can make use of it without consuming it.

```
composition {
    CreateServer.$server_id <->ConnectClient.$server_id;
}
```

This bidirectional binding is modeled as bidirectional arcs between the $server_id place and the CreateServer and ConnectClient transitions. But, if we look at the input block of the CreateServer rule we see that it does not consume a $server_id event so the only bidirectional arc is the one between the $server_id place and the ConnectClient transition. Note that bidirectional bindings and arcs are shorthand for two bindings and arcs respectively with opposite directions. Below the resulting CPN is shown.

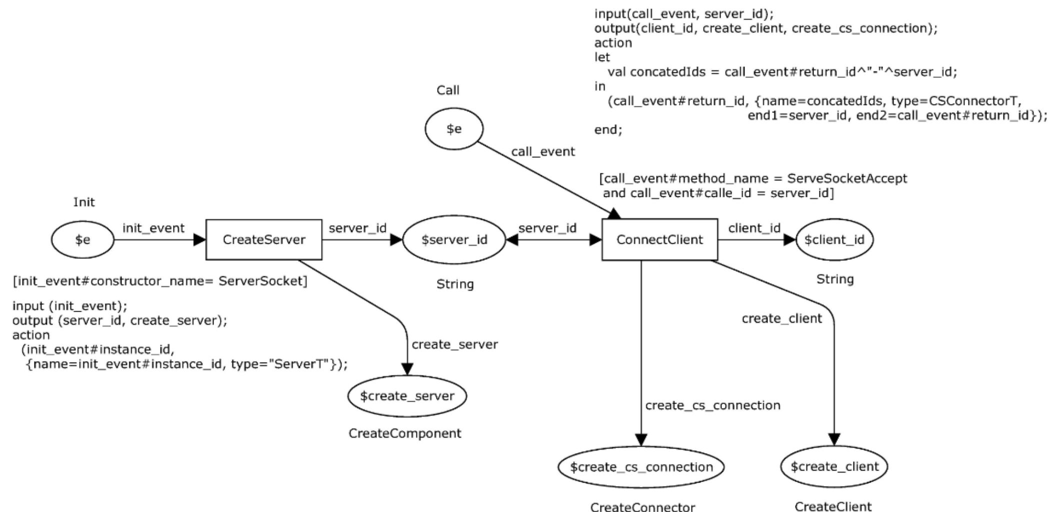



Figure 10

CPN declarations remain the same.

The complete translation of the Client-Server example to a CPN would be:

```
color String = string;
color Init = record constructor_name :String * instance_id
:String;
color Call = record method_name :String * callee :String *
return_id :String;
color CreateComponent = record name :String * type :String;
color CreateClient = record name :String * type :String;
color CreateConnector = record name :String * type :String *
end1 :String * end2 :String;
```

```
color UpdateComponent = record name :String * property:String
* value :String;
color UpdateConnector = record name :String * property:String
* value :String;

var init_event :Init;
var call_event :Call;
var server_id, client_id, io_id, activity_type : String;
var create_server :CreateComponent;
var create_client :CreateClient;
var create_connector :CreateConnector;
var update_component :UpdateComponent;
var update_connector :UpdateConnector;
```

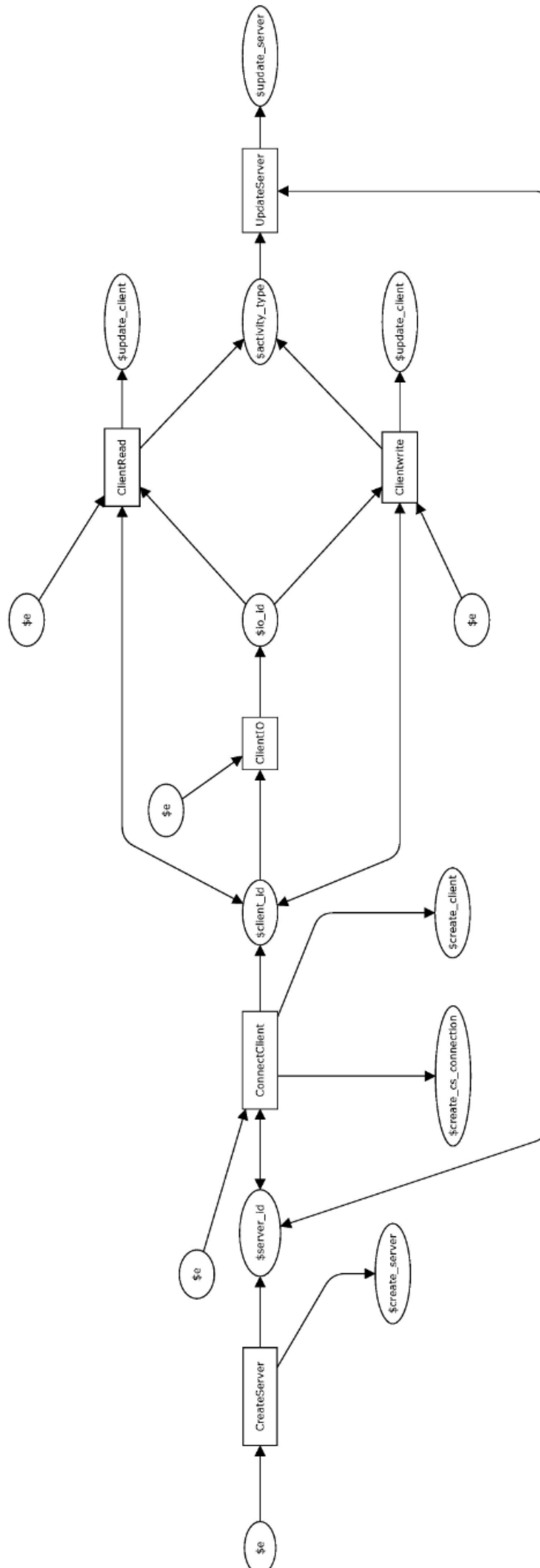



Figure 11

## 4.3. Specify the CPN model in Promela and C code

In this section it is explained how to specify a CPN, that models a set of architectural ***DiscoSTEP*** mapping rules, into a Promela specification. As a teaching example, we show how to specify the parts of the CPN shown in the previous section which models a set of architectural ***DiscoSTEP*** mapping rules for a Client-Server architecture.

The whole specification is written in Promela, including the stub; nevertheless, there are some embedded pieces of C code to manipulate accumulations of value and equality assumptions because of the adopted ***symbolic execution*** approach.

In the following subsections it is explained how CPN constructs are specified in Promela language.

The specification was modularized in different files grouped by the CPN specification, the architecture scenario, the Assumption Verifier and general files such as the main one and support files. A detailed explanation of the contents of each file may be found in Appendix C.

### 4.3.1. Places

Places are modeled as Promela channels. Channels are used to transfer messages between active Promela processes. Channel declarations are preceded by the reserved word **chan** followed by channel names. By default, channels store messages in FIFO order. Messages can have many fields of different types. Below we show all channel declarations and definitions done for the Client-Server architecture:

```
/*
CPN places are specified as channels.

Prefixes:
------------------------
CS:     ClientServer
CC:     ConnectClient
CIO:    ClienrIO
CR:     ClientRead
CW:     ClientWrite
------------------------
*/

/* Input places. */
chan CS_ePlace  = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan CC_ePlace  = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan CIO_ePlace = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan CR_ePlace  = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan CW_ePlace  = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};

/* Intermediate (input and output) places. */
chan server_idPlace      = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan client_idPlace      = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan io_idPlace          = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan activity_typePlace = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};

/* Output places. */
chan create_serverPlace          = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan create_clientPlace          = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan create_cs_connectionPlace   = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan CR_update_clientPlace       = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan CW_update_clientPlace       = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
chan update_serverPlace          = [maximumAmountOfTokensPerPlace] of {tokenIdInPlace};
```

Let's take as an example the first Promela sentence in the above piece of code; it declares a channel called **CS_ePlace,** defined with a maximum channel capacity of **maximumAmountOfTokensPerPlace** messages which only have one field of

type **tokenIdInPlace**. This sentence models a place that can contain at most **maximumAmountOfTokensPerPlace** tokens of type **tokenIdInPlace**.

In some parts of our Promela specification channels are identified by numbers denoted by a constants. Below we show these constants:

```
/* Input places Ids. */
#define CS_ePlaceId                   0
#define CC_ePlaceId                   1
#define CIO_ePlaceId                  2
#define CR_ePlaceId                   3
#define CW_ePlaceId                   4

/* Input and output places Ids. */
#define server_idPlaceId              5
#define client_idPlaceId              6
#define io_idPlaceId                  7
#define activity_typePlaceId          8

/* Output places Ids. */
#define create_serverPlaceId          9
#define create_clientPlaceId          10
#define create_cs_connectionPlaceId   11
#define CR_update_clientPlaceId       12
#define CW_update_clientPlaceId       13
#define update_serverPlaceId          14
```

Note that constant names have the channel name as prefix and the word “Id” as suffix.

## 4.3.2. Tokens

A token may carry one or many property values depending on whether its colorset is simple or compound. Due to our symbolic execution approach it is not necessary for a token to carry values in its properties when the CPN is being executed. Instead we do need to identify each token because conditions on token properties (assumptions) are collected when executing the CPN

Tokens identifiers are of type **tokenIdInPlace** which is a compound type as we can see below:

```
/* A symbolic token representation */
typedef tokenIdInPlace
{
        short locTId;
        byte pId;
}
```

The **locTId** property represents a local token identifier, and the **pId** property is the place identifier. When a new token is created in a place, it is identified using the place identifier and a local consecutive number for the token in the place. In this way a token is univocally identified in a place. So we defined a macro that does it whenever a token is created, as we can see below:

```
/* Token id initialization. */
inline newTokenIdInPlace(newToken, newPlaceId)
{
    d_step
    {
        newToken.pId = newPlaceId;
        newToken.locTId = nextTokenIdForPlace[newPlaceId];
        nextTokenIdForPlace[newPlaceId] = nextTokenIdForPlace[newPlaceId] + 1;
    }
    assert(maximumAmountOfTokensPerPlace > newToken.locTId);
}
```

Where **nextTokenIdForPlace[newPlaceId]** stores, as its names says, the identifier for the next token to be created in the place **newPlaceId** which is used as index for the array.

We decided to use this identification structure with two numbers (instead of using only one identifying number for each token) as a way to optimize the model checking process. Model checkers detect the state space, and its changes, of the specification while it is being analyzed. The state space contains the information which identifies the status of the model. This means that the order in which tokens are created affects the state space because for the model checker the number used as an identifier has a meaning. For example if we have a state space where there are two tokens in a place which the model checker identifies with the numbers 1 and 2,then there is another state space which also has two tokens in the same place but which are identified 2 and 1. Logically the equivalence of these two state spaces will depend on the properties of the tokens. For the model checker the state spaces will not be equivalent because the identifiers are different. The comparison made by the model checker between the two state spaces will see the channels (places) as queues, and will compare the messages in the order they have been inserted into the channel. For us the only meaning for the numbers is the ability to identify each token, but not the number itself. We decided to identify each token in this way due to how the chosen model checker ***SPIN*** works. This option allows ***SPIN*** to make a better identification of two equivalent state spaces. This is easily seen in an example where there are two places, each with one token. If only one number is used, and this number is created with consequent numbers in order to be sure that the numbers will not be repeated, two possible state spaces may arise depending which token is inserted first. Instead, using the chosen structure the order will not make any difference and only one state space is going to be created for this example.

If we look at channel declarations in the previous section, we see that token identifiers are specified as Promela messages for all channels. Since now on we will talk about tokens instead of token identifiers. This is because despite of dealing with token identifiers for our particular specification, we are in fact specifying tokens of a CPN.

Tokens can be removed or added from and into places, in the same way, Promela messages can be removed or added from and into Promela channels. For instance, when a token identifier message is consumed from a channel, we are specifying that a token is consumed from a place.

### 4.3.3. Colorsets

As explained in the previous section, in our Promela specification we deal with tokens identifiers, instead of tokens. All tokens identifiers are of type **tokenIdInPlace**. The question is then ¿How do we store the property values of tokens of different colorsets then? Well in a way we do not. We store assumptions of the possible values of these properties. For this we use a constant number to denote a token property and use this number as an index in an array where the property value is stored.

Below we can see the constant numbers for every token property of every colorset:

```
/* Colorsets */
c_decl
{
        /* String colorset. */
        unsigned char self             = 0;

        /* Init colorset. */
        unsigned char  constructor_name = 0;
        unsigned char  instance_id      = 1;

        /* Call colorset. */
        unsigned char  method_name      = 0;
        unsigned char  callee_id        = 1;
        unsigned char  return_id        = 2;

        /* CreateComponent and CreateClient colorsets. */
        unsigned char  name             = 0;
        unsigned char  type             = 1;

        /* CreateConnector colorset. */
        /* Use name and type constants for 0 and 1 respectively */
        unsigned char  end1             = 2;
        unsigned char  end2             = 3;

        /* UpdateComponent and UpdateConnector colorsets. */
        /* Use name constant for 0 */
        unsigned char  property         = 1;
        unsigned char  value            = 2;
}
```

Note that constant names are the same as the fields (or properties) of records in the declaration of the CPN colorsets:

```
color String = string;
color Init = record constructor_name :String *
instance_id :String;
color Call = record method_name :String * callee :String
* return_id :String;
color CreateComponent = record name :String * type
:String;
color CreateClient = record name :String * type :String;
color CreateConnector = record name :String * type
:String * end1 :String * end2 :String;
color   UpdateComponent   =   record   name   :String   *
property:String * value :String;
color   UpdateConnector   =   record   name   :String   *
property:String * value :String;
```

The String colorset is simple so it just has one value which is indexed with the **self** constant.

### 4.3.4. Transitions, transition guards, code segments and arc inscriptions

Each CPN transition is specified as a Promela **proctype** with some embedded pieces of C code to manipulate accumulations of assumptions. In our Promela specification we create a proctype for every transition in the CPN. These are: createServer(), connectClient(), clientIO, clientRead(), clientWrite() and updateServer(). The execution of each of these **proctypes** models the occurrences of a CPN transitions

.

We will explain how a CPN transition is specified in Promela language. To ease its understanding we will base our explanation in the CreateServer transition which will be used as example. Let's remember the transition:

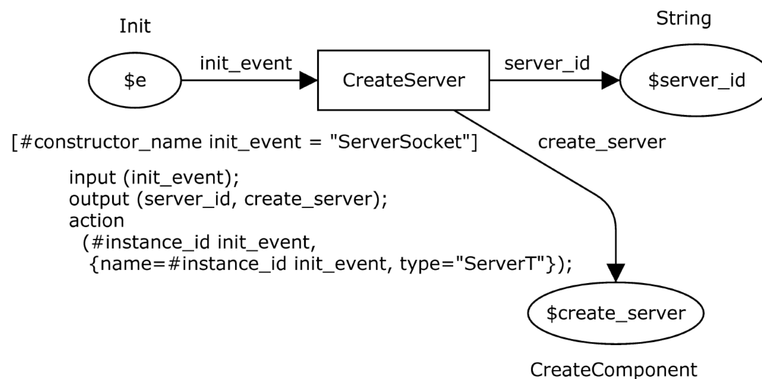



Figure 12

And let's see its Promela specification.

```
/* It models occurrences of the CreateServer transition. */
proctype createServer() {

      tokenIdInPlace init_eventTokenId;
      tokenIdInPlace server_idTokenId;
      tokenIdInPlace create_serverTokenId;
      byte isValidAss = 0;

      xr CS_ePlace;
      xs create_serverPlace;

start:
      atomic {
            /* Token consumptions from input places. */
            CS_ePlace ?? init_eventTokenId;
            c_code { runTimeEventCallDetector(PcreateServer->init_eventTokenId); };


            /* Transition guard: assumption accumulations. */

            /* init_event */
            c_code { PcreateServer->isValidAss = addValueAssumption(PcreateServer->init_eventTokenId, constructor_name, cServerSocket); };
            (isValidAss);


            /* Transition code segment: assumption accumulations. */

            /* create_server */
            newTokenIdInPlace(create_serverTokenId, create_serverPlaceId);
            c_code { PcreateServer->isValidAss = addValueAssumption(PcreateServer->create_serverTokenId, type, cServerT); };
            (isValidAss);
            c_code { addEqualityAssumption(PcreateServer->init_eventTokenId, instance_id, PcreateServer->create_serverTokenId, name); };

            /* server_id */
            newTokenIdInPlace(server_idTokenId, server_idPlaceId);
            c_code { addEqualityAssumption(PcreateServer->init_eventTokenId, instance_id, PcreateServer->server_idTokenId, self); };

            /* Token additions into output places. */
            create_serverPlace ! create_serverTokenId;
            server_idPlace ! server_idTokenId;
      }
      goto start;
}
```

Now we explain how this CPN transition is specified in Promela showing the mappings between parts of the CPN transition and parts of the **proctype**.

### 4.3.4.1. Proctype declarations

The incoming arc from the **$e** place into the **CreateServer** transition indicates it removes tokens from this place while the outgoing arcs to the **$create_server** and **$server_id** places indicate that the transition adds tokens to these places. For every token consumed from **$e**, a token is added into the **$create_server** and **$server_id** places. These three tokens are declared at the beginning of the **proctype**.

```
tokenIdInPlace init_eventTokenId;
tokenIdInPlace server_idTokenId;
tokenIdInPlace create_serverTokenId;
```

In fact inside the **proctype** we deal with token identifiers but we will refer to them as tokens. Then a variable called **isValidAss** is declared and initialized:

```
byte isValidAss = 0;
```

This variable is used to know the result of some later validations.

The CreateServer transition is the only one to consume tokens from the **$e** place and to add tokens into the $**create_server** place and it is specified with two assertions:

```
xr CS_ePlace;
xs create_serverPlace;
```

The first assertion states that the **createServer proctype** is the only process in the system that can consume messages from the **CS_ePlace** channel. While the second assertion states that the **createServer proctype** is the only one allowed to send messages to the **CS_createServer** channel.

### 4.3.4.2. Transition occurrence

All the actions involved in a transition occurrence happen all together instantaneously without interleaving with any other execution of the CPN. Inside the **createServer proctype** a loop is implemented with a label named **start** and a **goto** statement at the end of the **proctype**. A loop iteration specifies an occurrence of the CPN transition. The body of the loop is a Promela atomic block which allows executing the enveloped code indivisibly. Note that the loop does not have any guard, it's because the transition occurs whenever possible. To complete a transition occurrence, it is required that all the input places have at least one token. If this is not the case, the transition gets blocked until all the input tokens can be consumed. A blocking breaks the atomicity of the execution. This may seem as a problem because it may generate a scenario where a deadlock may appear when two or more **proctypes** have consumed at least one token and they need to consume more tokens which have been consumed by the others waiting **proctypes**. Fortunately, given that we decided to use a model checker, we do not have to

worry about this issue, the model checker will discard all the blocked scenarios until it finds the right order of consumption where no **proctype** gets blocked.

Note that there are two constructs in PROMELA for specifying that a sequence of statements must be executed atomically: **d_step** and **atomic**. We decided to use atomic because **d_step** has the limitations that except for the first statement in the sequence (the guard), statements cannot block, and as we see before it may happen.

All the constraints found in the CPN execution are stored as assumptions because of the ***symbolic execution*** approach adopted by us. Each assumption is validated twice, first when it is created and secondly when a trace that complies with the chosen scenario is found. The first validation helps the model checker to discard invalid solutions faster.

In the following subsections we explain in detail the inside of the atomic block.

### 4.3.4.3. Token consumptions from input places of the transition

When a transition occurs it first consumes tokens from its incoming places so inside the atomic block the token **init_eventTokenId** is first removed from the **$CS_ePlace** place. In Promela it is specified like this:

```
/* Token consumptions from input places. */
CS_ePlace ?? init_eventTokenId;
```

In this sentence a message is consumed from the **$CS_ePlace** channel and copied into the **init_eventTokenId** variable. The **??** operator determines that the election of the message to be consumed is carried out randomly.

If there is no message to be consumed from the channel then the statement is blocked and in consequence the execution of the **proctype** is blocked at this point. As explained before when this happen the model checker continues executing the other **proctypes**. The blocked sentence can be passed when the ***SPIN*** process scheduler chooses this **proctype** to execute again and the sentence can effectively be executed, it means, there is a message in the channel to be consumed.

A **c_code** block is used to write C code in it allowing, for example, calls to C functions defined in other files. The C function **runTimeEventCallDetector** is called inside a **c_code** block:

```
c_code {
        runTimeEventCallDetector(PcreateServer->init_eventTokenId);
};
```

This function keeps count of the number of consumptions of tokens (that model runtime events) that have been consumed by every transition in the CPN. In this case runtime events are of type **Call** or **Init**. It also registers the order in which each input token has been consumed. This is necessary to know, once a trace has been found, which runtime events must be executed, and in what order, to have the desired scenario. We register the global order number of consumption to be able to recognize token consumptions backtracked by ***SPIN*** so that they are not taken into account in the final analysis.

#### 4.3.4.4. Transition guard

If we look at the guard of the CPN transition, we see it has a condition on the **constructor_name** property of the **init_event** CPN variable which states its value to be the string "**ServerSocket**". In our Promela specification this condition is not evaluated at the time of the occurrence of the transition, instead, it is assumed to be true and accumulated to be verified later. The accumulation of this value assumption is carried out inside the second **c_code** block in the atomic block:

```
/* Transition guard: assumption accumulations. */

/* init_event */
c_code {
    PcreateServer->isValidAss = addValueAssumption(PcreateServer->init_eventTokenId, constructor_name, cServerSocket);
};
(isValidAss);
```

Inside the c_block the C function **addValueAssumption** is called with three arguments. The first one is the token **init_eventTokenId**, the second one is the name of the property **constructor_name** and the third argument is the value **cServerSocket** to which the **constructor_name** property is constrained to be bound. Note that it was decided to precede constant names with a lowercase letter c. In our example the **addValueAssumption** function accumulates a value assumption that states that the property **constructor_name** of the token **init_eventTokenId** has the value **cSeverSocket**. Before accumulating this assumption, the function evaluates if the token property has already been set and if its current value is different from the new one to be set. If this is the case then a contradiction is found and a false answer is returned, otherwise the function returns true. This answer is assigned to the **isValidAss** variable of the **createServer proctype**. Note that inside a **c_block** a proctype name must be preceded with an uppercase letter P, for example, **PcreateServer**, in order to access the objects defined in the **proctype** scope. An arrow is also used after the proctype name to refer to its local variable **isValidAss**. This variable is then evaluated by the model checker via this sentence.

```
(isValidAss);
```

This evaluation is always done after calling the **addValueAssumption** C function. If the value is false (represented by 0) then ***SPIN*** blocks this execution thread. With this blocking ***SPIN*** will not finish this transition which will eliminate any chance of finding the desired trace. Then the model checker engine is going to backtrack in order to try to find another trace which is not blocked.

#### 4.3.4.5. Transition code segment

The code segment of the CreateServer transition takes as input the **init_event** CPN variable (which is used as arc inscription for the incoming arc of the transition) and returns as output the variables **server_id** and the **create_server** CPN variables (which are used as arc inscriptions for the outgoing arcs form the transition). By the time the CPN transition occurs, the **init_event** variable has already been bound to a token from the **$e** place so we refer to this variable as the token itself. Inside the action clause of the code segment the **server_id** variable is bound to the **instance_id** property of the **init_event** token. After this binding or assignment both, the variable and the property, are equal. In our Promela specification this equality is stored as an equality assumption.

```
/* server_id */

newTokenIdInPlace(server_idTokenId, server_idPlaceId);

c_code {
    addEqualityAssumption(PcreateServer->init_eventTokenId, instance_id, PcreateServer->server_idTokenId, self);
};
```

Note that first the **server_idTokenId** token is initialized by calling the inline definition **newTokenIdInPlace.** The first parameter is the token itself and the second is the identifier of the place **server_idPlace** where this token will be added. Then the equality assumption is effectively added by calling the **addEqualityAssumption** C function inside a **c_block**. It accumulates an assumption that states that the **instance_id** property of the **init_eventTokenId** token is equal to the **server_idTokenId**.

Inside the action clause the CPN variable **create_server** is also bound to a record of the same colorset. In the record its **name** property is bound to the **instance_id** property of the **init_event** token and its **type** property is bound to the string value "ServerT." These bindings apply for the properties of the **create_server** variable given that the record is bound to it. We refer to **create_server** variable as a token. In our Promela specification these two bindings are specified as the accumulation of two assumptions:

```
/* create_server */

newTokenIdInPlace(create_serverTokenId, create_serverPlaceId);

c_code {
    PcreateServer->isValidAss = addValueAssumption(PcreateServer->create_serverTokenId, type, cServerT);
};
(isValidAss);

c_code {
    addEqualityAssumption(PcreateServer->init_eventTokenId, instance_id, PcreateServer->create_serverTokenId, name);
};
```

First the **create_serverdTokenId** token is initialized by calling the inline definition **newTokenIdInPlace** having as arguments the token itself and the identifier of the place **create_serverIdPlace** where this token will be added. Then the assumptions are stored. First the value assumption is added by calling the C function **addValueAssumption** inside a **c_block**. It accumulates an assumption which states that the value of the type property of the **create_serverTokenId** token is **cServeT**. Then the equality assumption is accumulated by calling the **addEqualityAssumption** C function inside another **c_block**. It accumulates an assumption that states that the name property of the **create_serverTokenId** token is the same as the **instance_id** property of the **init_event** token.

Note that we first accumulate all value assumptions and then equality ones because we want we want to avoid unnecessary accumulations of any equality assumption. If after the accumulation of a value assumption the **isValidAss** statement evaluates to false (a contradiction was found) then the model checker will backtrack and no unnecessary equality assumption accumulation will be carried out.

### 4.3.4.6. Token additions into output places of the transition

Finally the CPN tokens **server_id** and **create_server** are added into the places **$server_id** ans **$create_server** respectively. Remember that by this time the CPN

variables **server_id** and **create_server** have already been bound after the execution of the code segment, so we refer to them as tokens. In our Promela specification these additions are specified like this:

```
/* Token additions into output places of the transition. */

create_serverPlace ! create_serverTokenId;
server_idPlace ! server_idTokenId;
```

In these sentences the messages **create_serverTokenId** and **server_idTokenId** are sent to the channels **create_serverPlace** and **server_idPlace** respectively.

We have finished explaining the CreateServer transition and all its related CPN constructs.

## 4.3.4.7. Double-oriented arcs

In this subsection we explain how double-oriented CPN arcs are specified in Promela. We take as base example the ConnectClient transition:

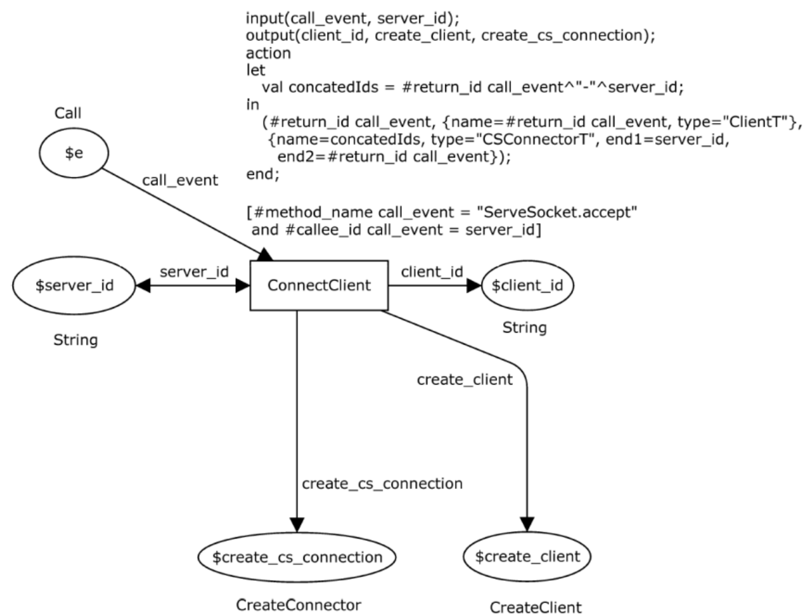


Figure 13

Remember that a double-oriented arc is shorthand for two arcs with the same arc inscriptions but opposite directions.

The Promela specification for this transition would be:

```
/* Represents occurences of a CPN transition. */
proctype connectClient() {
    /* c_code { log("connectClient"); }; */

    tokenIdInPlace call_eventTokenId;
    tokenIdInPlace server_idTokenId;
    tokenIdInPlace client_idTokenId;
    tokenIdInPlace create_clientTokenId;
    tokenIdInPlace create_cs_connectionTokenId;
    byte isValidAss = 0;

    xr CC_ePlace;
    xs create_clientPlace;
    xs create_cs_connectionPlace;

start:
    atomic {
        /* Token consumptions from input places. */

        CC_ePlace ?? call_eventTokenId;
        server_idPlace ?? <server_idTokenId>;

        c_code
        {
            runTimeEventCallDetector(PconnectClient->call_eventTokenId);
        };

        /* Transition guard: assumption accumulations. */
        .......

        /* Transition code segment: assumption accumulations. */

        /* create_client */
        .......

        /* create_cs_connection */
        .......

        /* client_id */

        .......

        /* Token additions into output places. */

        client_idPlace ! client_idTokenId;
        create_clientPlace ! create_clientTokenId;
        create_cs_connectionPlace ! create_cs_connectionTokenId;
        /*server_idPlace ! server_idTokenId;*/
    }
    goto start;
}
/* ---------------------------------------------------------------------------------- */
```

Note that a message is received from the **server_idPlace** channel and copied, not consumed, into the **server_idTokenId** variable. This is because of the double-oriented arc; instead of having the token first consumed and then immediately sent to the same channel we decided to have it copied directly to a local variable, which is semantically the same, in order to have a small performance improvement when model checking. We specify it in Promela in this way:

```
server_idPlace ?? <server_idTokenId>;
```

Note that the variable name is enclosed between angle brackets. This implies that the message will be copied and not consumed. If instead of using a variable we would have use a constant the angle bracket operator would have found a message using pattern matching.

## 4.3.5. CPN to Promela Conversion Algorithm

Below we present an algorithm to build a Promela specification from a CPN that models a set of architectural rules. The specifications shown in the previous sections are used.

#### 4.3.5.1. Places

1. For every CPN place:

    1.1. Create a channel with a capacity of **maximumAmountOfTokensPerPlace** messages which only have one field of type **tokenIdInPlace**.

    1.2. Define a constant number to identify the place. The constant name is the name of the channel that specifies the place with the suffix "**Id**". Each place must have a different constant number. The numbers must be consecutives starting from 0.

#### 4.3.5.2. Tokens

2. Tokens are identified as specified before. All token identifiers are of type **tokenIdInPlace.** A token is identified via a local id and the id of the place where it will be added:

```
/* A symbolic token representation */
typedef tokenIdInPlace
{
        short locTId;
        byte pId;
}
```

This token identifier specification is the same in all the Promela specifications of any CPN model.

#### 4.3.5.3. Colorsets

3. For every CPN colorset declared:

    3.1. If the colorset is simple, define a constant called **self** with value 0. Use the same constant for all declarations of simple colorsets.

    3.2. If the colorset is compound, a record in our case, then define constants named as the record property (or field) names and with different values in order to be able to uniquely identify a property of the record. Note that when different CPN colorsets have properties with the same names their corresponding constants could be reused.

#### 4.3.5.4. Transitions

4. For every CPN transition:

    4.1. Create a **proctype** and inside it write an infinite loop with an atomic block as body.

    **Arc inscription variables of incoming arcs**

4.2. For every CPN arc inscription of an incoming arc of the transition declare a variable of type **tokenIdInPlace** named as the name of the arc inscription variable with the suffix "**TokenId**".

These declarations are written at the beginning of the **proctype** and outside the loop. Inside the **proctype** these variables are treated as if they specified tokens.

**Input and output places of the transition**

4.3. For each input and output place of the transition:

4.3.1. If the transition is the only one to consume tokens from an input place then assert that this **proctpype** is the only one to receive messages from the corresponding channel.

4.3.2. If the transition is the only one to add tokens into an output place then assert that this **proctpype** is the only one to send messages to the corresponding channel.

Assertions are written at the beginning of the **proctype** and outside the loop.

**Transition occurrence**

**Token consumption from input places of the transition**

4.4. For every input place of the transition:

4.4.1. If it is only an input place of the transition then consume a message from the channel that specifies the place.

**Double oriented arcs**

4.4.2. If there is a double-oriented arc between the place and the transition, that is to say, the place is also an output place of the transition then receive a message from the channel that specifies the place but do not consume it.

In both cases the message must be received in an already declared variable (in 4.2) derived from the CPN arc inscription variable that surrounds the arc between the input place and the transition. Message receptions are carried out at the beginning of the atomic block of the loop.

**Transition guard**

4.5. For every condition in the transition guard accumulate a value or equality assumption. Write all accumulations of value assumptions first and then all the accumulations of equality assumptions. For every value assumption call the **addValueAssumption** C function inside a separate c_block and after every block evaluate the value returned by the function using a blocking statement. Reuse the **isValidAss** variable declared at the beginning of the **proctype**. The function

receives three parameters, the desired token, the constant which represent the property of the token to be bind and the binding value. For every equality assumption call the **addEqualityAssumption** C function. The function receives four parameters, the first token, the constant that represent the property of the first token to bind, the second token and the constant that represent the property of the second token to bind. Write all these calls inside the same c_block.

**Arc inscription variables of outgoing arcs**

4.6. For every CPN arc inscription variable of an outgoing arc of the transition:

4.6.1. Initialize the Promela variable declared in 4.2 for this arc inscription variable. This initialization consists in stating that this variable specifies the token to be added into the output place of the transition which is pointed by the arc that is surrounded by the arc inscription in matter. The initialization is carried out by calling the inline definition **newTokenIdInPlace** whose arguments are the mentioned Promela variable and the identifier of the mentioned output place of the transition.

**Transition code segment**

4.6.2. For every binding for the CPN arc variable or for every record property of it done in the action clause of the transition code segment accumulate a value or equality assumption. Write all accumulations of value assumptions first and then all the accumulations of equality assumptions. For every value assumption call the **addValueAssumption** C function inside a separate c_block and after every block evaluate the value returned by the function using a blocking statement. Reuse the **isValidAss** variable declared at the beginning of the **proctype**. The function receives three parameters, the desired token, the constant which represent the property of the token to be bind and the binding value. For every equality assumption call the **addEqualityAssumption** C function. The function receives four parameters, the first token, the constant that represent the property of the first token to bind, the second token and the constant that represent the property of the second token to bind. Write all these calls inside the same c_block.

**Token additions into output places of the transition**

4.7. For every output place of the transition send a message into the channel that specifies the place. This message must be sent from the variable derived from the CPN arc inscription variable that surrounds the arc between the output place and the transition. Messages are sent at the end of the atomic block of the loop.

## 4.4. Specify an architectural scenario in Promela

Scenarios are specified in Promela. In scenarios we can state facts such as the number of component creations, connector creations and connections between components including the number of occurrences of certain activities in components, ie, component executions. Below, a sample scenario is shown.

```
(len(create_serverPlace) == 1 && len(create_clientPlace) == 2 && len(create_cs_connectionPlace) == 2 );
```

It specifies that one server and two clients were created and they were connected with each other.

## 4.5. Model check the Promela specification of the CPN

This section explains general behavior of the Promela specification and C code and how to write the stub which depends on the architecture.

### 4.5.1. The init process

The **init proctype** is is the main process and is the first to be executed. Let´s see its specification:

```
init {
    byte isSatisfacible = 0;

    /* Initilize structrures */
    c_code {
        initAssStructures();
    };

    /* Initial CPN marking is built non-deterministically. */
    run stub();
    (stubFinishedPlacing == 1);

    /* CPN transitions able to occur. */
    atomic {
        #include "client-server CPN transitions to occur.pml"
    }

    /* The assertion for detection program finalization is defined. */
    #include "a client-server scenario.pml"

    /* Verify accumulated assumptions. */
    assert(maximumAmountOfEqualityAssumptionsPerProperty > maxEqAssMembersAmount);

    c_code {
        Pinit->isSatisfacible = assumptionVerifier();
    };
    (isSatisfacible);

    /* Printing. */
    c_code {
        printRunTimeEventCall(0);
        printValueAssumptions(0);
        printEqualityAssumptions(0);
    };
    assert(false);
}
```

The **isSatisfiable** variable is used to know if the trace found is valid. The structure that stores the assumptions to be accumulated when executing the CPN, is initialized. Then the stub is run to generate an initial marking in the CPN. Then a file which contains all the transitions of the CPN is included. They were specified as proctypes and they may occur non-deterministically. After executing the CPN we want to know if the architecture scenario is fulfilled. If this is the case then the Assumption Verifier is executed to evaluate all the assumptions accumulated when executing the CPN. In case they are valid then the trace of runtime events that made the scenario to be fulfilled, are printed exactly in the order they happened. This print is followed by the value and equality assumptions verified about the found trace.

## 4.5.2. Stub

Our aim is to find a trace of runtime events and we need something to generate them, so we specify a ***stub*** in Promela which is able to produce all possible traces of runtime events. An ***initial marking*** of a CPN sets a configuration of tokens in it, determining the tokens inside each place. A determined sequence of runtime events is conceptually equivalent to an initial marking in the CPN model derived from the mapping rules. This initial marking only configures the amount of tokens in the input places of the CPN because those are the tokens that represent the runtime events. Due to our ***symbolic execution*** approach we do not need to define any property of the tokens placed in the initial marking. Furthermore the stub establishes this initial marking in the modeled CPN for a bounded amount of tokens. For a same amount of tokens, different random initial markings are tried simulating different instrumentations, and consequent runtime events, that could be generated by any program.

The stub is specified by the a **proctype** called **stub()** which executes before all the processes that represent the actions. Let's see its specification for the Client-Server example:

```
proctype stub()
{
        atomic
        {
                tokenIdInPlace auxTokenId;
                short tokenAmountPlaced = 0;
                do
                :: (tokenAmountPlaced  < initialTokenAmount) ->
                        if
                        :: newTokenIdInPlace(auxTokenId, 0); CS_e ! auxTokenId
                        :: newTokenIdInPlace(auxTokenId, 1); CC_e ! auxTokenId
                        :: newTokenIdInPlace(auxTokenId, 2); CIO_e ! auxTokenId
                        :: newTokenIdInPlace(auxTokenId, 3); CR_e ! auxTokenId
                        :: newTokenIdInPlace(auxTokenId, 4); CW_e ! auxTokenId
                        fi;
                        tokenAmountPlaced++
                :: else -> break
                od;
                c_code
                {
                        initRunTimeEventCallStructure();
                };
                stubFinishedPlacing = 1;
        }
}
```

Inside the atomic block we see a loop; it iterates while the number of positioned tokens is lower than the chosen initial amount of tokens denoted by the constant **initialTokenAmount**. In every iteration only one token is initialized and

added into a place that models a runtime event, that is to say, a place that does not have incoming arcs. This place is chosen non-deterministically using an **if** construct where the decision between the inner statements is taken by the model checker. These statements must be modified depending on the model to be analyzed. There must be one line for each input place. It's important to note that an input place represents a type of runtime event.

## 4.6. The Assumption Verifier

In this section we are going to explain an overview of how the assumption verifier works. As we mentioned in previous sections, the Assumption Verifier verifies that all the accumulated value and equality assumptions has been satisfied.

The accumulated assumptions are collected in two structures called **EqualityAssumptions** and V**alueAssumptions** as we can see bellow.

```
/* VALUE ASSUMPTIONS ------------------------------------------------------------------ */
typedef struct TokenValueAssumptions
{
        unsigned char properties[maximumAmountOfPropertiesPerToken];
        unsigned isEmpty : 1;
} TokenValueAssumptions;

typedef struct PlaceValueAssumptions
{
        TokenValueAssumptions tokens[maximumAmountOfTokensPerPlace];
        unsigned isEmpty : 1;
} PlaceValueAssumptions;

typedef struct ValueAssumptions
{
        PlaceValueAssumptions places[placesAmount];
} ValueAssumptions;


/* EQUALITY ASSUMPTIONS --------------------------------------------------------------- */

typedef struct EqualityAssumptionMember
{
        short locTId;
        unsigned char pId;
        unsigned char propIdx;
} EqualityAssumptionMember;

typedef struct PropertyEqualityAssumptions
{
        EqualityAssumptionMember eqAssMembers[maximumAmountOfEqualityAssumptionsPerProperty];
        short eqAssMembersAmount;
} PropertyEqualityAssumptions;

typedef struct TokenEqualityAssumptions
{
        PropertyEqualityAssumptions properties[maximumAmountOfPropertiesPerToken];
} TokenEqualityAssumptions;

typedef struct PlaceEqualityAssumptions
{
        TokenEqualityAssumptions tokens[maximumAmountOfTokensPerPlace];
} PlaceEqualityAssumptions;

typedef struct EqualityAssumptions
{
        PlaceEqualityAssumptions places[placesAmount];
} EqualityAssumptions;
```

Both structures are inside a **c_decl** block, which means that they are written in C and that they declare C types.

```
c_decl
{
        typedef struct tokenIdInPlace TokenIdInPlace;

  ...

 }
```

We choose these structures in order to optimize the verification algorithm. The chosen structures provide a complexity order O(1) to insertion operations, O(1) to value assumption access operation and O(N) to equality assumption access operation; where N equals the maximum amount of equality assumptions per property constant which is configure by the user for each model. The complexity O(1) is given by using C direct access in arrays. The complexity O(N) is given by having to search inside the **EqualityAssumptionMember** array. These structures have the disadvantage of requiring much more memory space than the straight approach where you use a list to store the assumptions.

The algorithm is implemented completely in C. Let's see it:

```
uchar assumptionVerifier(){

    uchar actualPlace = 0;
    short actualToken = 0;
    uchar actualProperty = 0;
    uchar actualValue;
    uchar aValueAssumptionIsValid = 1;

    while (actualPlace < placesAmount && aValueAssumptionIsValid)
    {
        actualToken = 0;
        if (!now.valAss.places[actualPlace].isEmpty)
        {
            while (actualToken < maximumAmountOfTokensPerPlace && aValueAssumptionIsValid)
            {
                actualProperty = 0;
                if (!now.valAss.places[actualPlace].tokens[actualToken].isEmpty)
                {
                    while (actualProperty < maximumAmountOfPropertiesPerToken && aValueAssumptionIsValid)
                    {
                        actualValue = now.valAss.places[actualPlace].tokens[actualToken].properties[actualProperty];
                        if (actualValue != cNull)
                        {
                            aValueAssumptionIsValid = isValidAssumption(actualPlace, actualToken, actualProperty, actualValue);
                        }
                        actualProperty++;
                    }
                }
                actualToken++;
            }
        }
        actualPlace++;
    }
    return aValueAssumptionIsValid;
}
```

The algorithm verifies the absence of contradictions by calculating a transitive closure of the assumptions set. Let's see an example of a contradiction:

*token1.property1=1*
*token1.property1= token3.property4*
*token3.property4= token2.property3*
*token2.property3= 5*

As we can see, by transitivity, 1 = 5 which is a contradiction.

The algorithm navigates all the value assumptions and for each one checks if there is a contradiction. This is done by navigating the equality assumption tree, expanding the tree each time an equality assumption is found. Each node would represent a token property and the node value would be a value assumption. If no value assumption exist the node value is null. Once the whole tree has been navigated all the node values should have been null or the initial value. If not a contradiction has been found.

# 4.7. Summary of the key challenges raised when specifying and verifying the CPN

As we could see from sections 4.3 to 4.6 we had to cope with different types of challenges related to how to specify some CPN constructs and its operational aspects in Promela and how it affects the verification carried out by SPIN. The explanations of the solution to these challenges show the correction of the specification and the assertiveness of the taken decision. Below we sum up some of these challenges:

We found an appropriate and comfortable way to specify places as channels, transitions as processes declared as **proctypes** and tokens as messages. Channels may hold messages as places may hold tokens, and messages may carry data values as tokens do. **Proctypes** may access channels to allow communication between processes and they are shared, as a place may access transitions. In **proctypes** messages can be removed, added or just only accessed from and into channels; as transitions may remove and add tokens from and into places.

We needed to find an appropriate way to identify tokens as explained in 4.3.2. We found a way that make an optimization to the state space and consequently avoid unnecessary possibilities.

Having specified transitions as **proctypes** allowed us to simulate the nondeterministic occurrence of CPN transitions given that processes are run non-deterministically by SPIN.

Transitions occurrence is indivisible so execution of its corresponding **proctype** should be too. So we embraced its body in an atomic block. A transition is enabled only when all its incoming input places have tokens, otherwise it is not. We simulate it by blocking the process execution when a message is attempted to be received from a channel that specifies an incoming place. Fortunately the management of blocking and unblocking is carried out by SPIN when the conditions are given. Note that there are two constructs in PROMELA for specifying that a sequence of statements must be executed atomically: **d_step** and **atomic**. We decided to use **atomic** because **d_step** has the limitations that except for the first statement in the sequence (the guard), statements cannot block, and as we see before it may happen.

When executing the CPN and accumulating assumptions we did not want to accumulate unnecessary assumptions. So when detecting an invalid assumption the model checker cut that execution, backtracks and continues with other and we would not accumulate this assumption. Furthermore, in the case of value assumptions, we could immediately evaluate them whenever found.

After having found a way to specify a CPN, we wondered how we would manage its transition occurrences when SPIN verifies the CPN. As we explained in previous sections, while model checking, we may find a set of constraints related to all the transitions occurrences involved in a trace and so there may be more than one possible configuration for the trace. To cope with it, we decided to use a symbolic execution approach, so we accumulate all these constraints. We used this idea to create a model where each time there is a condition, or an assignment, an assumption is created with the condition stated. After a trace execution is finished

the assumptions are evaluated to see if the set is satisfiable comparing all the assumptions made.

Finally we had to put it all together. We had the CPN, the Stub, the Assumption Verifier and architecture scenario. We decided that the CPN execution would be carried out non-deterministically. The decision of using a symbolic approach obliged us to have at least an algorithm of two big steps: first execute the CPN and then evaluate the assumptions accumulated. It was clear that Stub should run at the beginning of the algorithm. But when should we verify the scenario? The scenario predicates on output places given that they model the architectural events generated by the DiscoSTEP mapping rules. So before evaluating all the conditions accumulated in the trace that leads to a CPN marking we first want to know if that marking is the desired one, that is to say, if the scenario is fulfilled. So we evaluate the architecture scenario after executing the CPN.

# 4.8. Run the SPIN Output Analyzer

We developed a tool that interprets the output trail generated by SPIN into a human readable trace, in this section we explain how we do this translation.

Analyzer to interpret the SPIN output trail and generate a human readable trace.

Translate the SPIN trace into a human readable trace of runtime events

## 4.8.1. The report generator

Once the model checker finds a trace that complies with the scenario specified a file is generated by ***SPIN***. This file contains the information of all the non-deterministic choices made by the model checker. This information may be read by the model checker to make simulations over the model, but is not easy to be read by an external program or by the human. Another file is generated with information of the model checking process itself. We decided to add information of the trace found to this file. So when a trace is found a C method is called which prints all the important information related with the trace found. This information will look similar to the following example:

```
RUN TIME EVENTS CALL ORDER - START
0 0 1
3 0 7
3 1 8
3 2 9
4 0 4
4 1 5
4 2 6
RUN TIME EVENTS CALL ORDER - END
VALUE ASSUMPTIONS - START
0 0 0 1
0 1 0 1
0 2 0 1
1 0 0 3
1 1 0 3
1 2 0 3
```

```
2 0 0 6
2 1 0 6
2 2 0 6
3 0 0 8
3 1 0 8
3 2 0 8
VALUE ASSUMPTIONS - END
EQUALITY ASSUMPTIONS - START
0 0 1 6 0 0
0 0 1 5 0 0
0 1 1 6 1 0
0 1 1 5 1 0
0 2 1 6 2 0
0 2 1 5 2 0
1 0 1 6 0 0
1 0 1 8 0 1
1 0 2 9 0 0
1 0 2 7 0 0
1 0 2 8 0 0
1 1 1 6 1 0
1 1 1 8 1 1
1 1 2 9 1 0
1 1 2 7 1 0
1 1 2 8 1 0
1 2 1 6 1 0
1 2 1 8 2 1
1 2 2 9 2 0
1 2 2 7 2 0
1 2 2 8 2 0
2 0 1 9 0 0
2 0 2 10 0 0
2 1 1 9 0 0
2 1 2 10 1 0
2 2 1 9 0 0
2 2 2 10 2 0
3 0 1 10 0 0
3 1 1 10 0 0
3 2 1 10 0 0
4 0 1 10 0 0
4 1 1 10 0 0
4 2 1 10 0 0
EQUALITY ASSUMPTIONS - END
```

Is easy to see that the information is divided in three sections:

1. **Run Time Events Call Order:** This section contains information regarding the events that were executed for the trace found and the order in which they were executed. Each line represents an event. Each event is represented by 3 numbers, the first is the place id, the second is the local token id and the third is the order number.
2. **Value Assumptions:** This section contains information regarding the values that the parameters of the events must have in order to have a complying trace. Each line represents the binding of a variable to a value. Each binding is represented by 4 numbers, the first is the place, the second is the local token id, the third is the property index and the fourth is the constant id.
3. **Equality Assumptions:** This section contains information regarding an equality relation between some of the events parameters. Each line represents a binding between two properties. Each binding is represented by 6 numbers, the first three numbers represent one property and the rest the other one. Each property is represented by three numbers, the first is the place, the second is the local token id and the third is the property index.

For a human it is very difficult and time consuming to analyze all these numbers. So we created a tool to automatically analyze this information. The input of this tool is the information explained earlier plus two sets of XML files. The first set has a XML file for each constant used in the model. Each XML contains the id and the name of a constant. The second set has a XML file for each place used in the model. Each XML contains the place id, the place name, if the consumption of a token represent a run time event or not, and the data type of the events. The data type has a name and the name of each property defined for the type. For simple or primitive data types as for example “String” a property named “value” is added for consistency with more complex types. This XML files have to be specified according to the model and scenario specified in Promela.

After executing the tool a report is generated with the information presented in a way easily understandable for humans. The following is an example of such a report:

```
CreateServer.e (constructor_name 'ServerSocket', instace_id A)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id A, return_id C)
ClientIO.e (method_name 'SocketGetInputStream', calee_id C, return_id D)
ClientWrite.e (method_name 'OutputStreamWrite', calee_id D, return_id-4-0)
ClientWrite.e (method_name 'OutputStreamWrite', calee_id D, return_id-4-1)
ClientWrite.e (method_name 'OutputStreamWrite', calee_id D, return_id-4-2)
ClientRead.e (method_name 'InputStreamRead', calee_id D, return_id-3-0)
ClientRead.e (method_name 'InputStreamRead', calee_id D, return_id-3-1)
ClientRead.e (method_name 'InputStreamRead', calee_id D, return_id-3-2)
ClientIO.e (method_name 'SocketGetInputStream', calee_id C, return_id-2-1)
ClientIO.e (method_name 'SocketGetInputStream', calee_id C, return_id-2-2)
CreateServer.e (constructor_name 'ServerSocket', instace_id B)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id B, return_id-1-1)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id B, return_id-1-2)
CreateServer.e (constructor_name 'ServerSocket', instace_id-0-2)
A = {instace_id-0-0, calee_id-1-0}
B = {instace_id-0-1, calee_id-1-1, calee_id-1-2}
C = {return_id-1-0, calee_id-2-0, calee_id-2-1, calee_id-2-2}
D = {return_id-2-0, calee_id-3-0, calee_id-3-1, calee_id-3-2, calee_id-4-0,
calee_id-4-1, calee_id-4-2}
```

The report has two different sections. The first is the resulting trace of the whole process. All the events appear in the order in which they should be executed. For each event you can see its name and the parameters for that specific execution. For each parameter you can either see its type name and value, type name and equality class or type name with unique instance identification. The second section shows the definition for each equality class. Each class is composed by two or more instances.

# 5. Conclusions

## 5.1. Concluding remarks

Having chosen the symbolic execution approach was an asserted key decision given that otherwise we would not have been able to inspect the whole state-space when model-checking.

Working with ***SPIN*** conducted us to write the specification in Promela which turned out to be a very intuitive language to specify a CPN. Furthermore ***SPIN*** has the flexibility that allowed us to easily add custom C code, and it was fast enough to model check big scenarios.

As we could see we had to cope with some issues when specifying the CPN in Promela, mainly related to the semantic and operational aspects of the CPN. Fortunately we could find neat and clear ways to do it with Promela and Spin, and the theoretical background of SPIN help us to verify the correctness of some key parts of this work. In our opinion, SPIN is great model checker and Promela is a very comfortable language to specify concurrency and synchronization.

We first started our work with the idea of just providing a way to verify if some important architecture scenarios were still valid in an already implemented system at runtime. Fortunately we noted that our technique approach was flexible enough to be used at an early stage of a system development, during its codification and testing, and even after finishing the development. It took us to conclude that our work could be useful during the whole development process and that it seems to be worth extending our technique and developing tools to automate steps of it, tasks that are left as future work.

When we modeled the ***DiscoSTEP*** mapping rules as a CPN, we detected some errors in these rules. So we discovered that our technique could also be used as a way for healing the mapping rules, especially if the modeling is carried out automatically.

Nowadays distributed architectures are more common than ever thanks to the new Cloud services, with its natural elasticity that helps to scale horizontally in ways that were never thought before. What´s more new and complex architectures like CQRS ES or EDA are being commonly used in the industry. Even though the advantages are clear, the architecture complexity gets bigger every day, even in the most basic web applications. The importance of our work grows alongside this complexity making the possibility of leaving the academic sphere to enter the industry a reality.

## 5.2. Potencial applications

Below we list some of the possible uses of this:

### System architecture testing

It could be used to test that an already developed system complies with the key architecture scenarios in terms of components, connectors and their relationships.

### Software development guidance

This technique can be used to help take some decisions when coding and at design time being able to test how the system may behavior under certain architectural scenarios. It provides a tool to minimize the risk of developing an application that does not comply with an already defined architecture. It provides a guide on how to realize a given specified architectural scenario.

### Code generation

The presented technique may also be used as a tool to generate source code of the system skeleton so that when correctly used by an application, it would assure that the architecture is being complied. In this way we could effectively state that the generated system complies with the architecture scenario.

Let´s see an example based on the client-server DiscoStep mapping rules sample of this thesis and let´s suppose we want to generate Java code.

DiscoStep mapping rules could be specified without having written any piece of code, just knowing the programming language syntax and considering some design aspects given that we have to choose class names, method names, object relations, etc. which impact directly in the code to be generated as a consequence of the application of a mapping rule.

Let´s suppose that after verifying an architecture scenario our SPIN Output Analyzer outputs this trace:

```
CreateServer.e (constructor_name 'ServerSocket', instace_id A)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id A, return_id C)
ClientIO.e (method_name 'SocketGetInputStream', calee_id C, return_id D)
ClientWrite.e (method_name 'OutputStreamWrite', calee_id D, return_id-4-0)
ClientWrite.e (method_name 'OutputStreamWrite', calee_id D, return_id-4-1)
ClientWrite.e (method_name 'OutputStreamWrite', calee_id D, return_id-4-2)
ClientRead.e (method_name 'InputStreamRead', calee_id D, return_id-3-0)
ClientRead.e (method_name 'InputStreamRead', calee_id D, return_id-3-1)
ClientRead.e (method_name 'InputStreamRead', calee_id D, return_id-3-2)
ClientIO.e (method_name 'SocketGetInputStream', calee_id C, return_id-2-1)
ClientIO.e (method_name 'SocketGetInputStream', calee_id C, return_id-2-2)
CreateServer.e (constructor_name 'ServerSocket', instace_id B)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id B, return_id-1-1)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id B, return_id-1-2)
CreateServer.e (constructor_name 'ServerSocket', instace_id-0-2)
A = {instace_id-0-0, calee_id-1-0}
```

B = {instace_id-0-1, calee_id-1-1, calee_id-1-2}
C = {return_id-1-0, calee_id-2-0, calee_id-2-1, calee_id-2-2}
D = {return_id-2-0, calee_id-3-0, calee_id-3-1, calee_id-3-2, calee_id-4-0, calee_id-4-1, calee_id-4-2}

And if we look, for instance, at the definition of the first involved rule, the CreateServer one:

```
rule CreateServer {
    input { init $e; }
    output { string $server_id; create_component    $create_server;
    }
    trigger {? contains($e/@constructor_name, "ServerSocket") ?}
       action = {?
        let $server_id := $e/@instance_id;
        let $create_server := <create_component name="{$server_id}"
        type="ServerT"/>;
        ?}
}
```

we could immediately infer that we there must be a class called `ServerT`.

Then if we look at the second rule, ConnectClient,:

```
rule ConnectClient {
    input { call $e; string $server_id; }
    output {create_component $create_client; create_connector
    $create_cs_connection;
           string $client_id;}
    trigger {? contains($e/@method_name, "ServerSocket.accept") and
            $e/@callee_id = $server_id ?}
    action = {?
        let $client_id := $e/@return_id;
        let $create_client := <create_client name="{$client_id}"
        type="ClientT" />;
        let $create_cs_connection :=
              <create_connector name= concat($client_id,"-
              ",$server_id)
              type="CSConnectorT" end1="{$server_id}"
              end2="{$client_id}" />;
    ?}
}
```

We could infer that there must be a class called `ClientT`. If we observe the let part we see that there is a `create_connector` high level event which represents that there exist a static relationship between the server and the client. The server is identified because its id is received as a parameter in the input part and both rule applications:

CreateServer.e (constructor_name 'ServerSocket', instace_id A)
ConnectClient.e (method_name 'ServerSocketAccept', calee_id A, return_id C)

are related because of the `A` argument. Finally we know that the server references or contains the client as a consequence of the rules composition:

```
composition System {
    CreateServer.$server_id<-> ConnectClient.$server_id;
    …
}
```

At first sight we can see that some types and some static relationships can be inferred with just a simple observation. A more deep analysis may show other useful characteristics.

### DiscoSTEP mapping rules healing

Given that the way of modeling the mapping rules as a CPN is clearly defined it is a good opportunity to verify the correctness of them. When we were modeling the CPN for the set of ***DiscoSTEP*** mapping rules of a client-server architecture, we detected some bugs and errors in these rules as a consequence of following the translations rules (from mapping rules to CPN) exposed in this thesis. So we discovered that our technique could also be used as a way for healing the mapping rules, especially if the modeling is carried out automatically.

## 5.3. Future work

Below we list some tentative improvements or extensions for our work:

1. In this thesis we presented the detailed steps to model whatever set of ***DiscoSTEP*** mapping rules as a CPN and how to specify this CPN in Promela. It would be useful to automate these two steps so that the ***DiscoSTEP*** mapping rules are taken as input and the Promela specification of its corresponding CPN is returned as output.

2. We specify an architecture scenario as a Promela assertion about the places of the CPN that models the ***DiscoSTEP*** mapping rules of the components and connectors of the architectures. We think that developing a visual SDL (Scenario Description Language) and a tool to specify a scenario directly in terms of these last elements (and then translated into a Promela assertion about the corresponding CPN) would facilitate the task of writing scenarios.

3. The source code generator mentioned in a section before could be developed to generate the skeleton of the system from a set of ***DiscoSTEP*** mapping rules.

The three above items could extend the current architecture of our work as shown below.

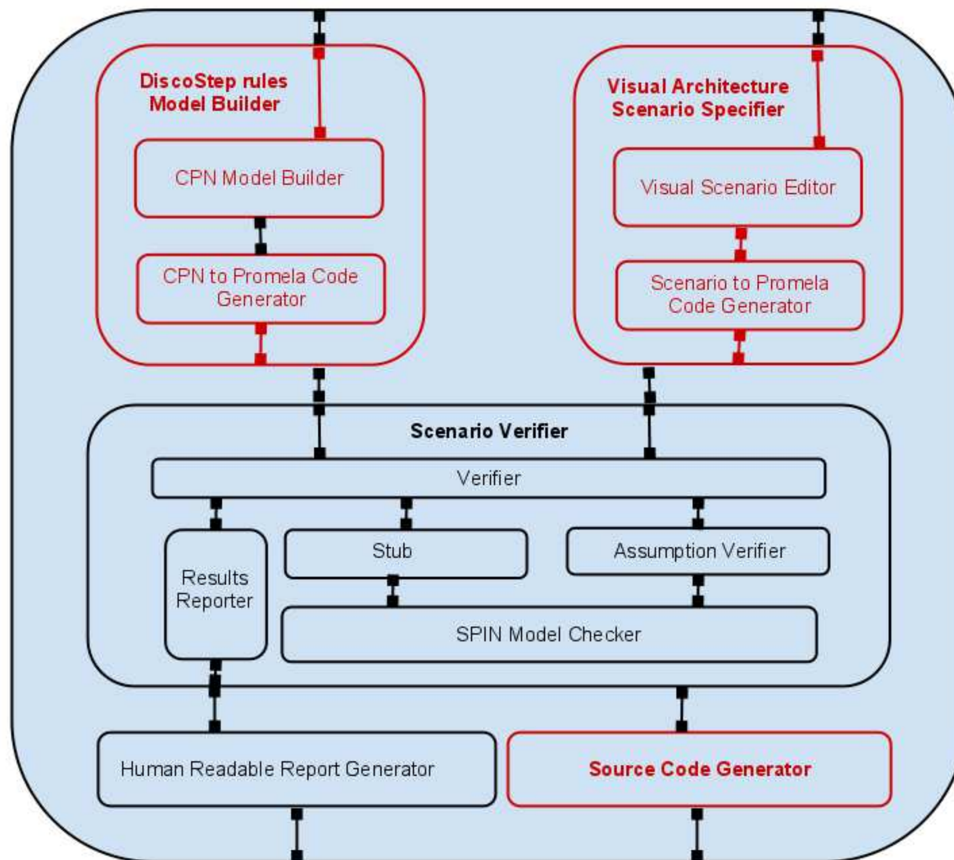



Figure 14

Note that the responsibility of the new added components were clearly described above.

4. We support architecture scenarios about the number of component creations, connector creations and connections between components including the number of occurrences of certain activities in components, ie, component executions. Scenarios could be extended to support, for example, the specification of ***order*** between the creations, connections and activities.

5. In our implementation we simplify the translation of XPath statements defined in [4] into simple equality statements between two variables or a variable and a constant. We could extend our work to support the full power of XPath statements.

6. A way of making our work more accessible and more popular could be to develop an Eclipse plugin to carry out all the steps involved in our technique including all the extensions improvements and mentioned above. The plugin could allow us, for example, to visualize the resulting CPN, to execute it, to ran Spin verifications, visualize traces and reports.

7. We only implemented completely the Client-Server architecture rules defined by Garlan. We tried some small variants but in the future it would be good to implement more architectures.

# 6. References


## About colored Petri nets

[1] Kurt Jensen; **An Introduction to the Theoretical Aspects of Coloured Petri Nets**. PB-476.

[2] Kurt Jensen; **A Brief Introduction to Coloured Petri Nets**.

[3] Kurt Jensen; **An Introduction to the Practical use of Coloured Petri Nets**.

## About architecture discovery

[4] David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman, Hong Yan; **DiscoTect, A System for Discovering the Architectures of Running Programs Using Colored Petri Nets**. Marzo del 2006. CMU-CS-06-109.

[5] David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman, Hong Yan; **DiscoTect: A System for Discovering Architectures from Running Systems**.

[6] David Garlan, Bradley Schmerl, Hong Yan; **Dynamically Discovering Architectures with DiscoTect**.

[7] David Garlan, Bradley Schmerl, Jonathan Aldrich, Rick Kazman, Hong Yan; **Discovering Architectures from Running Systems, Lessons Learned**.

## About model checking

[8] Gerald C. Cannon and Sunil Gupta; **An Automated Tool for Analyzing Petri Nets Using SPIN**.

[9] C. Pajault y J.-F. Pradat-Peyre; **Static Reductions for Promela Specifications**. CEDRIC 1005.

[10] Gerard J. Holzman; **SPIN Model Checker, The Primer and Reference Manual**. Septiembre del 2003. Editorial Addison Wesley

[11] Gerarld C. Gannod y Sunil Gupta; **An automated tool for analyzing Petri Nets using SPIN**. Noviembre del 2001. 16th International Conference on Automated Software Engineering, pp 404 – 407, IEEE.

## About symbolic execution

[12] Stephen F. Siegel, Anastasia Mironova, George S. Avrunin y Lori A. Clarke; **Using Model Checking with Symbolic Execution to Verify Parallel Numerical Programs**. 2005. UM-CS-2005-15.

[13] Michael Baldamus and Jochen Schr¨oder–Babo; **p2b: A Translation Utility for Linking Promela and Symbolic Model Checking (Tool Paper)**.

## Others

[14] Victor Braberman, Nicolas Kicillof, and Alfredo Olivero; **A Scenario-Matching Approach to the Description and Model Checking of Real-Time Properties**.

[15] Wolfgang Grieskamp and Nicolas Kicillof; **A Schema Language for Coordinating Construction and Composition of Partial Behavior Descriptions**.

[16] Wolfgang Grieskamp, Nicolas Kicillof and Nikolai Tillmann; **Action Machines, a Framework for Encoding and Composing Partial Behaviors**.

# Appendix A

## DiscoSTEP client-server mapping rules

Below a ***DiscoSTEP*** program for mapping runtime events into a client-server architecture is shown. It was taken from [4] but some details were corrected.

```
event {
     input {
            init;
            call;
            string;
     }
     output {
            string;
            create_component;
            create_client;
           create_connector;
            update_component
            update_connector
     }
}

rule CreateServer {
   input { init $e; }
   output { string $server_id; create_component $create_server; }
   trigger {? contains($e/@constructor_name, "ServerSocket") ?}
     action = {?
      let $server_id := $e/@instance_id;
      let $create_server := <create_component name="{$server_id}"
      type="ServerT"/>;
   ?}
}


rule ConnectClient {
   input { call $e; string $server_id; }
   output {create_component $create_client; create_connector
$create_cs_connection;
           string $client_id;}
   trigger {? contains($e/@method_name, "ServerSocket.accept") and
            $e/@callee_id = $server_id ?}
   action = {?
       let $client_id := $e/@return_id;
       let $create_client := <create_client name="{$client_id}"
       type="ClientT" />;
       let $create_cs_connection :=
              <create_connector name= concat($client_id,"-
              ",$server_id)
              type="CSConnectorT" end1="{$server_id}"
              end2="{$client_id}" />;
   ?}
}


rule ClientIO {
   input { call $e; string $client_id; }
   output { string $io_id; }
   trigger {? (contains($e/@method_name, "Socket.getInputStream")
   or contains($e/@method_name, "Socket.getOutputStream")) and
            $e/@callee_id = $client_id ?}
   action {? let $client_id := $e/@return_id; ?}
}
```

```
rule ClientRead {
    input { $e : call; $io_id : string; $client_id : string; }
    output { $update_client : update_component; $activity_type :
    string;}
    trigger {? (contains($e/@method_name, "InputStream.read") and
    $e/@callee_id = $io_id ?}
    action = {?
        let $update_client := <update_component name=$client_id
    property="Read"
                            value="true" />;
        let $activity_type := "Read";
    ?}
}


rule ClientWrite {
    input { $e : call; $io_id : string; $client_id : string; }
    output { $update_client : update_component; $activity_type :
    string; }
    trigger {? (contains($e/@method_name, "OutputStream.write") and
            $e/@callee_id = $io_id ?}
    action = {?
        let $update_client := <update_component name=$client_id
        property="Write"
                            value="true" />;
        let $activity_type := "Write";
    ?}
}


rule UpdateServer {
    input { string $server_id; string $activity_type; }
    output { update_component $update_server; }
    trigger {? ($activity_type = "Read") or ($activity_type =
    "Write") ?}
    action = {?
        let $ update_server := < update_componnet name=$server_id
    property="Activity"
                              value=$activity_type />;
    ?}
}


composition System {
    CreateServer.$server_id<-> ConnectClient.$server_id;
    ConnectClient.$client_id -> ClientIO.$client_id;
    ConnectClient.$client_id <-> ClientRead.$client_id;
    ClientIO.$io_id <-> ClientRead.$io_id;
    ConnectClient.$client_id <-> ClientWrite.$client_id;
    ClientIO.$io_id <-> ClientWrite.$io_id;
    ClientWrite.$activity_id -> UpdateServer.$activity_id;
    CreateServer.$server_id <-> UpdateServer.$server_id;
}
```

# Appendix B

## Concrete syntax of DiscoSTEP Language

We present the concrete syntax of ***DiscoSTEP*** [2].

```
PROGRAM ::=
    IMPORT*; EVENT; (COMPOSITION | RULE) *

IMPORT  ::=
  import quoted file name

EVENT::=                                              event type declarations:
      'event' '{'
        'input' '{' (ID ';')* '}'
        'output' '{' (ID ';')* '}'
      '}'

RULE  ::=                                             rule declarations:
      'rule' ID '{' RULEPARTS '}'

RULEPARTS2 ::=                                        rule property declarations:
      'input' '{' (ID VARID ';')* '}'
      'output' '{' (ID VARID ';')* '}'
      'trigger' '{$' XPRED '$}'
      'action' '{$' XQUERY '$}'

COMPOSITION ::=                                       composition declarations:
      'composition' ID '{' COMPOSITIONPART* '}'

COMPOSITIONPART::=                                    composition property declarations:
      MEMBER '->' MEMBER
      MEMBER '<->' MEMBER

MEMBER ::=
      ID '.' VARID                      |
      ID '.' MEMBER

ID ::= [a-zA-Z][a-zA-Z0-9_]*

VARID ::= [$][a-zA-Z0-9_]*|
```

Note that the productions XPRED and XQUERY in the language refer to XQuery Predicates and XQuery FLWOR expressions, respectively. The grammar for these is defined in http://www.w3.org/TR/xquery/

[2] The concrete syntax was taken from the Appendix A of the paper: “DiscoTect: A System for Discovering the Architectures of Running Programs using Colored Petri Nets”

# Appendix C

# A client-server architecture example

## The CPN specification

The table below lists the files used for the Promela specification of the CPN model of the ***DiscoSTEP*** mapping rules for the client-server architecture.

| File name | Description |
|---|---|
| `client-server architectural rules CPN.decl` | This file specifies the declaration and definition of the places of the CPN, its colorsets, the token representation, the structure that stores the tokens in the places and some CPN constants. |
| `client-server architectural rules CPN.pml` | This file specifies all the CPN transition occurrences and a stub for this architecture which determines a random initial marking for the input places of the CPN |
| `client-server CPN transitions to occur.pml` | In this file the CPN transitions are executed. |

## The architecture scenario specification

The specification is divided into these files:

| File name | Description |
|---|---|
| `a client-server scenario.pml` | This file contains a Promela assertion about the CPN places |
| `constants related to a client-server scenario.decl` | This file contains related constants to the scenario. |

## The Assumption Verifier

The implementation is divided into these files:

| File name | Description |
|---|---|
| `assumption verifier.decl` | This file contains the declaration and definition the structures that contain value and equality assumptions (conditions). |
| `assumption verifier.c` | This file contains functions and procedures to initialize the structures that contain the assumptions, to populate this structure with value an equality assumptions and to evaluate these assumptions. |

# General files

| File name | Description |
|---|---|
| support.decl | This file contains support structures and variables. |
| support.c | This file contains support procedures to accumulate and print out the runtime events and to print out the value and equality assumptions. |
| client-server_Main.pml | This file contains the entry point for the Spin verification. This file includes all the other files. I also contains a macro that adds a token in a place. Although the file is generic for whatever architecture, it must include the files related to the specific architecture and scenario. |