HAL
archives-ouvertes.fr

# Integrated environment for verifying and running distributed components -Extended version

Ludovic Henrio, Oleksandra Kulankhina, Siqi Li, Eric Madelaine

▶ **To cite this version:**

Ludovic Henrio, Oleksandra Kulankhina, Siqi Li, Eric Madelaine. Integrated environment for verifying and running distributed components -Extended version. [Research Report] RR-8841, INRIA Sophia-Antipolis. 2015, pp.24. <hal-01252323>

**HAL Id: hal-01252323**
**https://hal.inria.fr/hal-01252323**

Submitted on 7 Jan 2016

**HAL** is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

# Integrated environment for verifying and running distributed components - Extended version

**Ludovic HENRIO , Oleksandra KULANKHINA , Siqi LI , Eric MADELAINE**

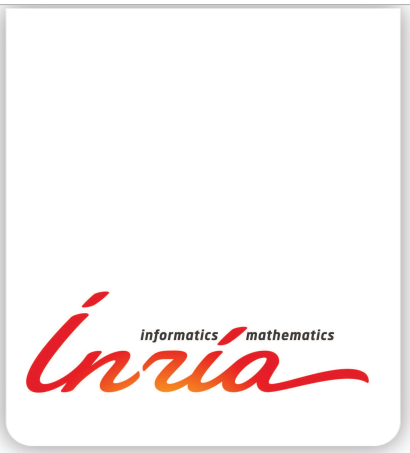
**RESEARCH REPORT**

**N° 8841**

December 2015

Common Project-Team Scale

ISSN 0249-6399 ISRN INRIA/RR--8841--FR+ENG




# Integrated environment for verifying and running distributed components - Extended version

Ludovic HENRIO *, Oleksandra KULANKHINA † *, Siqi LI ‡, Eric MADELAINE † *

Common Project-Team Scale

Research Report n° 8841 — December 2015 — 24 pages

**Abstract:** This paper targets the generation of distributed applications with safety guarantees. The proposed approach starts from graphical specification formalisms allowing the architectural and behavioral description of component systems. From this point, the user can automatically verify application properties using model-checking techniques. Finally, the specified and verified component model can be translated into executable Java code. We implement our approach in a tool suite distributed as an Eclipse plugin. This paper also illustrates our approach by modeling and verifying Peterson's leader election algorithm.

**Key-words:** Formal methods, Distributed systems, Behavioral semantics, Structured labelled transition systems

SCALE is a common team between the I3S Lab (Univ. of Nice Sophia-Antipolis and CNRS) and INRIA Sophia Antipolis Méditerranée

* University of Nice Sophia Antipolis, CNRS, UMR 7271, 06900 Sophia Antipolis, France
† INRIA Sophia Antipolis Méditerranée, BP 93, 06902 Sophia Antipolis, France
‡ MoE Engineering Research Center for Software/Hardware Co-design Technology and Application, East China Normal University, 200062, Shanghai, China

RESEARCH CENTRE
SOPHIA ANTIPOLIS – MÉDITERRANÉE

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

# Un environnement intégré pour la vérification et l'éxécution de composants distribués - Version étendue

**Résumé :** Ce travail concerne la génération d'applications distribuées avec garanties de bon fonctionnement. L'approche proposée commence avec des formalismes graphiques permettant la spécification architecturale et comportementale des systèmes de composants. Ensuite, l'utilisateur peut vérifier automatiquement les propriétés de l'application en utilisant des techniques de model-checking. Enfin, le modèle spécifié et vérifié peut être traduit en code exécutable Java. Nous mettons en oeuvre notre approche dans une suite d'outils Eclipse distribués sous forme de plugin. Ce rapport illustre notre approche par la modélisation et la vérification de l'algorithme d'élection distribué de Peterson.

**Mots-clés :** Méthodes formelles, Systèmes distribués, Sémantique comportementale, Systèmes de transition structurés,

# 1 Introduction

Component-oriented programming has become a popular approach for distributed application development. Components enforce a clear design and specification stage of the applications, and provide a solid basis for safe and modular development of complex systems. This work aims at including systematic verification of behavioral properties in the development process of component-based applications. For this purpose we would like to provide the developers of distributed component-based systems with a set of tools supporting rigorous design and implementation of safe applications. Our tools should guide the user through all crucial phases of component software development: from application design specification to verification of the designed architecture and behavior properties as well as automated code generation.

Applying static analysis on hand-coded programs is complex and often imprecise, especially for distributed systems. Instead we chose a Model-Driven Engineering and component-oriented approach in which the structure of the application is directly specified by the developer, and in which the final code is generated automaticaly, partialy or totaly.

VerCors[1] is a software platform which aims at supporting the creation of safe distributed component-based applications. VerCors[2] includes a set of graphical designers based on UML where the user can specify the architecture and the business logic of his application, and check the static correctness of the component architecture [1]. The specification is then automatically transformed into a behavior graph that can be model-checked to prove its correctness. We rely on model-checking for verification, but we want to hide as much as possible the complexity of the underlying formal techniques to make our tools accessible to non-experts in model-checking. VerCors uses parametrized networks of asynchronous automata (pNets) as an intermediate format for behavior modeling and relies on CADP [2] model-checker to verify temporal properties. Last, Java code of the modeled application can be automatically generated and executed. We rely on ProActive[3] and the Grid Component Model (GCM) [3]. We chose GCM/ProActive because it targets distributed systems and features a well-defined semantics. Because of the chosen verification methodology, the current platform can only verify finite-state systems, but infinite-space systems can already be specified and modeled as pNets.

This paper shows that our approach is suitable for applications involving complex interactions between processes but without too much computational complexity. For the case studies involving such a computational complexity the model-checking approach might be limited. However in that case we advocate the use of the VerCors platform to specify and verify the core of the application, abstracting away computational details. The user can still generate the executable skeleton of the verified core application. He can then extend it with computational details. While the application logic is unchanged, the behavioral properties will still be valid.

The VerCors platform has already undergone several major generations, with significant evolutions for the underlying semantic model, as well as the modeling platform and the specification formalisms. The original version was using UML component structures for describing the application architecture, but this was too far from GCM needs, hence a new DSL and graphical formalism were defined. At the same time, aiming at better support for maintenance and usability, VerCors was moved to an Eclipse-based environment [4]. A series of publications described the support for several features of distributed component-based systems, including group communications, first-class futures, and reconfiguration. At that time, the platform was only able to generate part of the behavioral model and it relied on several manual steps only realizable by experts in formal methods. No code generation was supported. Starting from that preliminary

[1] https://team.inria.fr/scale/software/vercors/vcev4-download/
[2] Not to be confused with http://fmt.cs.utwente.nl/research/projects/VerCors/
[3] https://team.inria.fr/scale/software/proactive/

work a new VerCors tool is presented in this paper. It includes the full set of modeling formalisms (architecture, types abstractions, and state-machines), the validation of static correctness, the full chain of tools for the generation of a pNet model for model-checking, as well as a new tool for automatic generation of executable GCM/ProActive code. More recently, theoretical papers defining the pNet model [5] and the behavioral semantics of GCM in terms of pNets [6] were published. They build a formal foundation for the VerCors tools.

First, Section 2 presents the background on GCM, the pNets formalism and our use-case (Peterson's leader election algorithm). In Section 3 we introduce a set of graphical formalisms to define abstractions of distributed component-based system architecture and behavior. In Section 4 we show how the specified models can be transformed into behavioral graphs accepted as input by a model-checker. We present in Section 5 the generation of executable code from the model specification. Finally, we discuss the related work in Section 6 and conclude in Section 7. We illustrate our contributions by modeling, verifying, and running Peterson's leader-election algorithm[4][7].

# 2 Background

## 2.1 Grid Component Model and ProActive platform

The Grid Component Model (GCM) [3] targets large-scale distributed component systems. Its reference implementation is GCM/ProActive.

**Architecture**. A GCM application consists of components, interfaces and bindings. Figure 2 illustrates an example of a GCM system. A component can be either *composite* (it consists of other subcomponents), e.g. `Application`, or *primitive* (a simple element encapsulating business code), e.g. `Comp1`. Components communicate through *interfaces* of two types: client and server (e.g. `C1` and `S1` correspondingly). A component sends requests and receives replies through *client* interfaces; a component receives requests and sends back results through *server* interfaces. The interfaces that communicate are connected with *bindings*.

ProActive is a Java library for distributed computing. Every component in GCM/ProActive is an active object made of a single applicative thread.

**Informal semantics of ProActive components**. Figure 1.a illustrates treatment of requests by primitive components. Every primitive component has a FIFO request queue, a body and an active object that serves requests. All requests to the server interfaces are first dropped to the queue. The body takes the first request from the queue and triggers the execution of the corresponding method of the active object. To process a request the component may need additional services provided by the other components, using operations calls on its client interfaces. Once a request is served, the component sends back a reply consisting of the value returned by the method. Then, the next can be served.

Figure 1.b illustrates the behavior of a GCM/ProActive composite. A composite has a FIFO request queue, a body, an associated active object, and some subcomponents. The body takes requests from the queue and forwards them to the subcomponents that serve them. In order to serve a request, a subcomponent may need to call methods of other subcomponents or outside of the composite, using client interfaces. Once a request has been served by the subcomponent, the composite receives the reply and forwards it to the requester. Every request sent from a subcomponent towards the outside of a composite passes by the queue of the composite before being forwarded through the composite client interface.

[4] available at: https://github.com/Scale-VerCors/VCEv4/tree/master/Examples

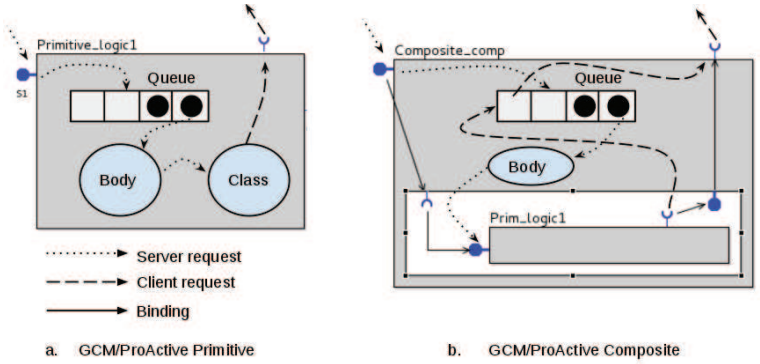



Figure 1: GCM/ProActive component behavior

GCM components communicate using futures. When a component sends a request to another component, the caller continues its execution as long as it does not need the result of the request. When the result is needed the caller blocks automatically. We call this behavior a "wait-by-necessity". In the meantime, an empty object called *future* represents the result of the request.

## 2.2 pNets

Parametrized networks of asynchronous automata (pNets) have been formalized in [5]. pNets are composition of labeled transition systems with parameters; they are used as an intermediate model for encoding behavior of GCM-based applications. The behavioural semantics of GCM has been formalized in [8, 6]. A pNet is a hierarchical structure where leaves are pLTSs. A pLTS is a labelled transition system with variables, where labels are of the form $\langle \alpha,\ e_b,\ (x_j := e_j)^{j \in J} \rangle$, where $e_b$ is a guard, the variables $x_j \in P$ are assigned when the transition is triggered, finally $\alpha$ is a parametrized action that has a label and a set of arguments, some of them are input variables, others are output expressions. By convention, we annotate actions with "!" and "?" depending on the information flow. We assume that the information goes from $!\alpha$ to $?\alpha$. A pNet is either a pLTS or the composition of several pNets; in the second case, the possible interaction between sub-entities are specified by *synchronisation vectors*: $pNet \triangleq pLTS \mid \langle\!\langle L, pNet_i^{i \in I}, SV_k^{k \in K} \rangle\!\rangle$ where $L$ is the set of global actions, $pNet_i^{i \in I}$ is the family of sub-pNets. $SV_k^{k \in K}$ is a set of synchronization vectors. $SV_k = \alpha_j^{j \in J_k} \rightarrow \alpha'_k$ means that each of the sub-pNets in the set $J_k$ can perform synchronously an internal action $\alpha_j$; this results in a global action $\alpha'_k$. Elements not taking part in the synchronization are denoted $-$ as in: $< -, -, \alpha, - > \rightarrow \alpha$.

## 2.3 Peterson's leader election algorithm

Distributed processes often need to select a unique leader. Peterson's election algorithm [7] can be used for this purpose in a unidirectional ring of asynchronous processes. Every process participating in the elections has a FIFO queue and the order of sent messages is preserved by the communication channels. Each process can be either in active mode if the process participates in the election, or in passive mode if it only forwards messages. Initially, every process stores a unique number that will be modified during the election. The processes exchange two rounds of messages so that every active process learns the numbers stored by the two nearest active processes preceding it. If the maximum of the two previous values and the value held by the current process is the value received from the nearest predecessor of the process, then the active process takes this value as its own value; otherwise the process becomes passive. The rounds

of messages and local decision steps are repeated until a process receives its own number, this process is the leader.

In details, every process $P$ stores variables $max(P)$ and $left(P)$. $Max(P)$ is the number stored by $P$. $Left(P)$ is the number of the active process on the left of $P$. Processes exchange messages of the form $M(step, value)$ where *step* is the phase of the algorithm. At the *preliminary phase*, each process $P_i$ sends $M(1, max(P_i))$ to its neighbor. Then, if an active process $P_i$ receives a message $M(1, x)$ and $x$ is equal to its own number, the process is the leader, otherwise it assigns $x$ to $left(P_i)$ and sends $M(2, x)$ to its neighbor. When an active process $P_i$ receives $M(2, x)$ it compares $left(P_i)$ to $x$ and $max(P_i)$. If $left(P_i)$ is greater than both values, $P_i$ assigns $left(P_i)$ to $max(P_i)$ and sends $M(1, max(P_i))$; otherwise $P_i$ becomes passive.

# 3 Graphical designer

VerCors includes a graphical designer for modeling component-based system architecture and behavior. These models must be precise enough to be translated into both input for validation and for executable code. The graphical specification part of VerCors is based on EclipseIDE; it was implemented using Sirius[5]. The VerCors platform includes graphical designers for four types of diagrams: Components, UML Class, UML State Machine, and Type diagrams. This section describes the four editors and the way they are integrated.

## 3.1 Architecture specification

Component diagrams are used for the specification of a distributed application architecture. A component diagram includes primitives (grey boxes), and composites (white rectangles with grey border). Interfaces are attached to the borders of their containers. An interface has a set of characteristics, e.g. whether an interface is server or client. The icon representing an interface changes depending on the characteristics. Bindings are shown as arrows between interfaces.

UML Class diagrams are used to specify the list of attributes stored by components and the list of operations a component offers. The user can attach a UML class to a primitive component and a UML interface to client and server interfaces. If a class is attached to a component, it means that the attributes of the class are stored by the component and the operations of the class define the business logic of the component. A UML interface attached to a client or a server GCM interface stores the list of operations that can be called and served with this interface. Each operation defined in a class either has a reference to the operation of the interface it implements (or redefines in UML terms), or is a local method of the component.

The types of operations, attributes, and variables can be declared using Type diagrams. Enumerations, integer intervals, records (C-like structs) and infinite integers can be specified, while boolean and void types are created by default.

**Use-case example** The Component diagram representing the architecture of our use-case model is shown on Figure 2. The UML Class diagram of our example is given in Appendix A.

`Application` is a composite; it includes four primitives that participate in the leader election process. The primitives are connected in a ring topology and have similar structure. The entry point of the system is the *runPeterson()* operation of `Application` server interface `S1`. This request is forwarded to `Comp4` that triggers the election process. During the election, components invoke method *message* on their client interfaces `C1`. As defined in Section 2.3,

[5]Sirius is an open-source Eclipse project for development of graphical modeling environment based on EMF and GMF: http://www.eclipse.org/sirius/

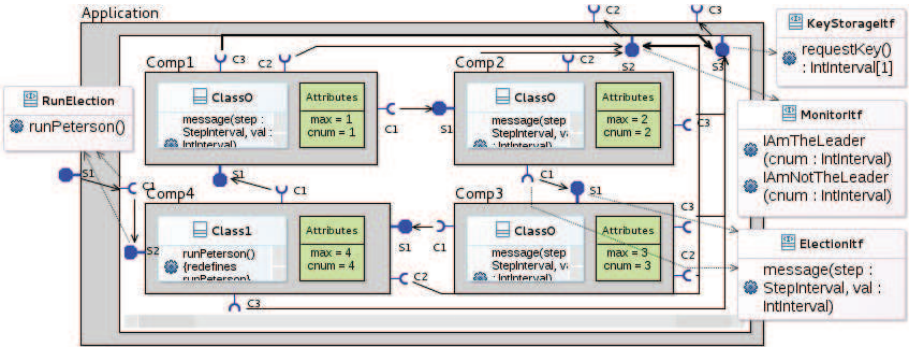



Figure 2: Components diagram

each message transmits two parameters: *step* and *val*. The message is transmitted to the server interface `S1` of the called component. The signature of *message* is specified in a UML interface `ElectionItf`. If a component decides to become a leader or a non-leader, it reports its decision to the environment by invoking an *IAmTheLeader(cnum)* or an *IAmNotTheLeader(cnum)* method on its client interface `C2`. These operations take the identifier of the component as a parameter.

All four components have the same set of attributes. They have the *message(...)* method implementing the leader election algorithm and a set of methods to access local attributes. `Comp4` implements an additional operation *runPeterson()*. `Comp1`, `Comp2`, and `Comp3` are implemented by `Class0` while `Comp4` uses `Class1` that extends `Class0` with *runPeterson()* operation. Initially, the components should have different default values of attribute `max` and `cnum`. `cnum` is a static unique identifier of a component. To specify the values of those attributes for every component individually, we define them in the **Attributes** field represented as a green box in every primitive definition.

In our model we define two integer interval types on Type diagram (see Appendix B) : *StepInterval = 0..2* for the parameter *step* of messages and *IntInterval = 1..4* for the component unique identifier.

## 3.2 Behavior specification

UML State Machine diagrams are used for behavior specification in VerCors. Each State Machine defines the behavior of an operation of a UML Class.

A State Machine has a set of states connected by transitions. A state stores its name, while logic code is specified on transitions. To enable behavioral analysis we specify the syntax of UML transitions: a transition has a label of the form `[guard]/action1....actionN` where **Guard** is a boolean expression and an **action** is an assignment or a method call (to a local operation or a client interface). This set of actions is sufficient to encode any behaviour of distributed objects; control structures have to be encoded as guards on transitions.

The VerCors UML-based editors are based on Obeo UML Designer[6]. In particular, we integrated the State Machines graphical designer of Obeo UML Designer into VerCors, adding local variable declarations. A State Machine has access to its own local variables, to the client interfaces and to local methods of the component which behavior the State Machine describes. A State Machine can access the attributes of the component but only through getters and setters.

Figure 3 illustrates the State Machine of the *message* method of Peterson's leader election

[6]http://www.umldesigner.org/

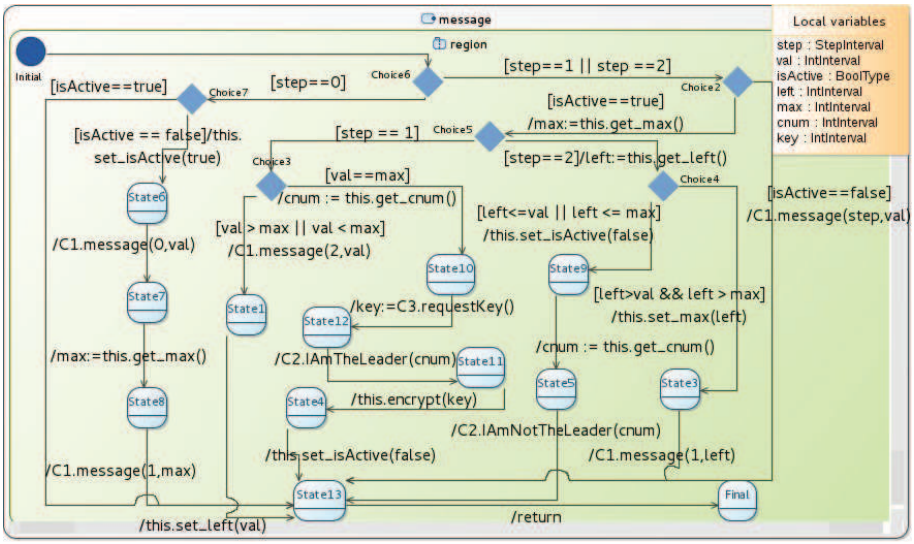



Figure 3: Message State Machine

algorithm. It uses seven variables where *step* and *val* are input parameters of the method. The initial state is illustrated with a blue circle. First, `Choice6` checks the phase of the election algorithm. If the algorithm is in the preliminary (zero) phase either the component is active – it already participates in the election – or the component triggers the election process on its neighbor and performs the preliminary phase described in Section 2.3. If it is not the preliminary phase, either the component is passive and the message is forwarded to the neighbor `[isActive==false]/C1.message(step,val)`, or the actions of the State Machine correspond to the two cases $M(1, x)$ or $M(2, x)$ depending on the value of *step* (see Section 2.3).

To illustrate future-based communications in VerCors, we extend our use-case as follows. If a component decides to become the leader, it sends a *requestKey()* invocation on its client interface (see the transition from `State10` to `State12`). The request is forwarded to outside of `Application`. Then, the component claims itself as the leader by sending an *IamTheLeader(cnum)* request. Finally, the component calls its local method *encrypt(key)* using the result of *requestKey()* as a parameter. The component should be able to claim itself as the leader before it receives the result of *requestKey()*. However, it cannot execute *encrypt(key)* if the *key* is not obtained. The VerCors user does not need to explicitly model future-based communications. Whenever a State Machine has a non-void client method invocation, it is interpreted as a future-based one.

To conclude, four integrated diagram editors are implemented in VerCors. Component diagrams correspond to architecture specification, Class diagrams represent attributes and method signatures of components, State Machine diagrams are used for behavior specification, and Type diagrams define type abstractions. They allow the user to easily describe his/her application and provide sufficient input both for model-checking and for code generation.

# 4 Behavior verification

From user-defined architecture and behavior models VerCors produces input data for the CADP [2] model-checker following a chain of transformations presented in this section. First, we analyze

input models and generate a corresponding pNet structure. Second, we generate a finite graph given as an input to CADP, together with auxiliary scripts for managing state-space explosion. Finally, the user can specify the properties that he wants to check on the generated graph and run CADP. While the specified system and the pNet model rely on parameterized state-machines potentially featuring infinite state-space, the model-checking phase can handle finite state-space only. As a consequence, the correctness of the finite abstraction should be checked by abstract interpretation techniques. From another point of view, the pNet model could also be checked by a different tool that handles infinite state-space.

## 4.1 From application design to pNets

We present here the generation of pNets specifying the application behavior [6].

A pNet of a **primitive** assembles pLTSs of two types: the generic ones whose structure is identical for all primitives (e.g. queue, body) and the pLTSs generated from the user-defined State Machines (server and local methods behavior). Appendix C gives examples of generated pLTSs for our use-case. Figure 4 shows the pNet generated for `Comp1` of our use-case. An **Attribute controller** pLTS is generated for each attribute of a primitive; it allows storing and modifying the value of this attribute. The list of component attributes can be derived from the UML Class of the component. **Proxy** and **ProxyManager** pLTSs are generated for every client operation having a non-void result. They model the implementation of the futures mechanism. A pLTS is generated for each **server and local method**. For this purpose we translate UML State Machines specifying methods behavior into pLTSs. To translate a State Machine into a pLTS we first map each state of a State Machine into a pLTS state and each transition to one or several pLTS transition (potentially adding intermediate states). For example, a State Machine transition [isActive==true]/max:=this.get_max() involves one guard condition and two actions: a call to a local function *get_max* and a return of its result. A pLTS transition can perform at most one action, hence, the result of the translation will consist in two sequential transitions.

The behavior of the components is modeled by synchronization vectors, expressing the synchronization and the data flow between pLTSs. As an example, the Body and the Queue pLTSs of a primitive are synchronized using:

$<!Serve_message(...), ?Serve_message(...), -, -, -, -, - >\rightarrow Serve_message(...)$

in which, the subnets occur in the following order:

$< Queue, Body, message, max_ac, cnum_ac, left_as, isActive_ac >$ .

Synchronization of the Queue with the environment under reception of a request is expressed by: $<?Q_message(...), -, -, -, -, -, - >\rightarrow ?Q_message(...)$ , meaning that this action is exposed at the next level of pNet to synchronize with another pNet. The other vectors synchronize the following entities: the Body and a server method pLTS ($Call_message(...)$); a server method pLTS and other local methods, or client method of the environment; the server method, the Body and the environment to return the result ($R_message(...)$); the environment and the Queue when the Queue is saturated, raising an $Error_queue$ event.

The pNet of a **composite** assembles pLTSs for queue, body and sub-entities enabling futures mechanism with pNets of the subcomponents. The request reception mechanism is similar to the one of a primitive. The only difference is that the body is synchronized with subcomponent pNets in order to forward them the requests. pNets of subcomponents are synchronized with each other under internal method invocation (*e.g.* $Comp4_Comp1_message(...)$) and result reception. If a subcomponent invokes an operation outside of the composite, it synchronizes with the composite queue. Then, the queue synchronizes with the environment and forwards the request to outside of the composite. More detailed description of the synchronization vectors generated from VerCors can be found in Appendix D.

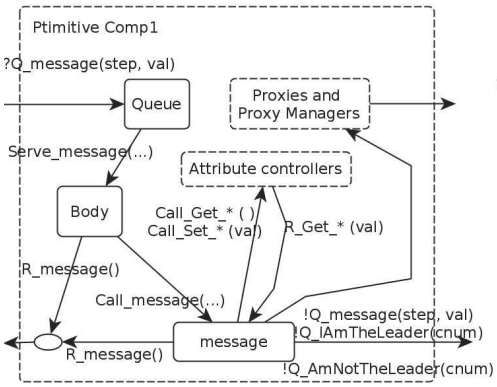



Figure 4: pNet of Comp1

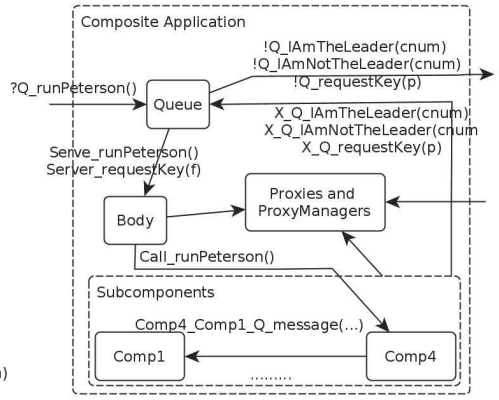



Figure 5: pNet of Application

**Scenario**. The user can specify a Scenario State Machine, encoding the legal sequences of actions performed by the environment, accessing only the server interfaces of the root component. The scenario of our use-case calls the *runPeterson* method on interface `S1` of `Application` once. The scenario State Machine is translated into a pLTS and synchronized with the queue of the root component. This leads to a much smaller and meaningful behavior model.

## 4.2 From pNets to Model-Checking

**Generation of verification input.** As the next step, VerCors translates the pLTSs into the Fiacre format [9] and the synchronization vectors into EXP [10]. Then, the FLAC compiler translates the Fiacre specification into Lotos code. Finally the CADP front-end generates a labelled transition system in a format that can be used by the CADP model-checker. We generate a set of scripts for managing the execution of all steps: communication hiding, minimization, and hierarchical product using EXP files. In order to limit the state-space explosion phenomenon inherent to explicit-state model-checkers, the user should:

- use a scenario to limit acceptable inputs of the modeled system,
- specify the internal actions that he does not want to observe during model-checking (we generate a script transforming them into *internal* actions),
- limit the size of the data domains using the Types diagram.

All generated transition systems are minimized using branching bisimulation.

We have used the VerCors model-generation function to produce Fiacre, EXP and auxiliary scripts for our use-case. Table 1 presents size information for some of the intermediate behavior graphs. The last line is for the hierarchical construction of the full model of the application (including the Scenario), and the time includes the whole model-generation workflow. The time needed to generate Fiacre, EXP files and scripts from VerCors is neglectible.

**Model-checking.** We use the Model Checking Language (MCL [11]) to express the behavioral properties we want to prove on our system. MCL is a very expressive logic including first order predicates for the data part, and the alternation free $\mu$-calculus for branching time logics. On top of MCL, we use *Specification Patterns* [12] for easier expression of some usual temporal logic properties, as in the examples below. We recall that in our example the properties are evaluated in the context of the scenario where the election algorithm is triggered.

First, we check that after a call to *runPeterson()*, it is inevitable (under fairness hypothesis) that either the leader is elected or one of the queues is saturated. The model-checker answers

Table 1: Behavior graph files (all with Queue size of 3)

| Graph | States | Transitions | Computation time |
|---|---|---|---|
| Behaviour of `Comp4` | 3.217.983 | 45.055.266 | 2m48.520s |
| Comp4 (internal communication hidden, minimized by branching simulation) | 90.821 | 1.306.138 | 5m23.030s |
| full application | 296 | 661 | 47m1.673s |

`true`: the election terminates. We also proved that with adequate queue size, they never saturate.

['Call_RunPeterson'] Inev ('Q_IamTheLeader.*' or 'ErrorQueue.*')

Then, we prove that the event `Q_IamTheLeader` is emitted only once:

Absence_Before ('Q_IamTheLeader.*', 'Q_IamTheLeader.*')"

In order to check that the communications in the generated graph are indeed implementing futures properly, we verify the following formula which states that a key is always received before IamTheLeader() is invoked:

Existence_Between('R_RequestKey.*', 'Q_requestKey.*', 'Q_IamTheLeader.*')

The model-checker answers `false` and provides an example of system behavior where *IamTheLeader()* method is invoked before the key is received. This proves that a component is not blocked if the key is not needed.

To summarize, from the graphical models provided by the user we automatically generate a behavior description in the form of pNets, and translate these into an input for CADP verification tools. We tested our approach on our use-case and proved by model-checking the correctness of the application, including its safety, termination, and functional correctness.

# 5 Code generation and execution

## 5.1 Executable code generation

From the specified architecture and behavior we automatically generate executable code. We produce an ADL (XML) file defining architecture, and Java interfaces and classes files for the implementation of the methods specified by State Machines. This code can be run using the GCM/ProActive Java library.

We generate a Java interface for every UML interface and a Java class for every UML class. An excerpt of our use-case ADL file is provided in Appendix E. We translate each State Machine attached to a method into Java code. To do this we use a Java enumeration representing the state machine steps, a local variable `curState` holds the current state of the state machine and actions are taken depending on this state. Listing 1 shows a skeleton of the encoding of the *message* operation from Figure 3. Note that if-else statements are used for states with more than one outgoing transition. For example in Choice 2, the guard label `[isActive==false]` is translated as an if-else statement in line 12; depending on the result, a *message* invocation is emitted (corresponding to `C1.message(...)`, line 13) and the value of `curState` is updated (line 14). A drawback of this approach is that such code may not be very convenient for the programmer since *do-while, for, while* constructs cannot be written as such in the state machine, but will rather be encoded within the state structure, separated by case instructions. We also generate skeleton code for getter and setter methods, which have no associated state machine.

Listing 1: Generated Java code of message

```
1  Boolean isActive = null;
2  Integer left = null, max = null, cnum = null;
3  State curState = State.Initial;
4  while(true) {
5   switch (curState) {
6   ...
7   case Choice2:
8           if(isActive == true) {
9             max=this.get_max();
10            curState = State.Choice5;
11            break; }
12          else if(isActive == false) {
13            C1.message(step, val);
14            curState = State.State13;
15            break; }... };}
```

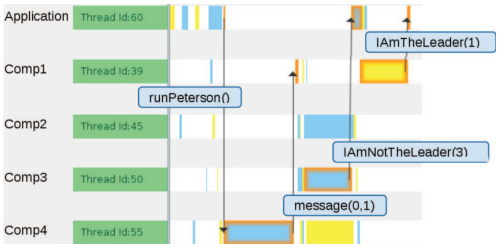



Figure 6: Code execution

The Java code generated by VerCors relies on futures. To implement their generation, we analyze the State Machines and mark the variables that store remote method invocation results. This information is used to generate the types of those variables and to access their values. For example, the `key` variable from our use-case State Machine will be generated with an *IntWrapper* type[7]. Then the statement `this.encrypt(key)` requires the value of `key` and it will be translated to the following Java code: `this.encrypt(key.intValue())`.

## 5.2 Code execution

We generated ProActive/Java code of our use-case example; the resulting execution is shown in Figure 6[8]. Black arrows represent request emissions (the figure only shows some of them). Yellow and blue rectangles show request processing. For example, we can see how the call to *runPeterson* of Application is transmitted to Comp4 and at the end of the *runPeterson* request processing Comp4 triggers the elections on Comp1 by calling *message(0,1)*. At the end of the algorithm execution we can see how Comp3 reports to the Application that it is not the leader and Comp1 claims to be the leader.

To sum up, from the specification provided by the user VerCors automatically produces executable ProActive/Java code. We generated and executed code of our use-case model and we observed expected behavior of the produced system. The generated code is guaranteed to verify the temporal properties proven on the model. It can either be used as it is or serve as code skeleton if the programmer wants to add computational steps that he did not include in the model.

[7] basic types need to be wrapped to enable future-based commnuications

[8] We use a dedicated tool for the visualization of ProActive program execution: `https://github.com/scale-proactive/A-viewer-tool-for-multiactive-objects.git`

# 6 Related Work

There exist a number of languages, formalisms, and tools aiming at verification and safe code generation, we focus here on the ones that are dedicated to distributed systems and composition of distributed systems.

BIP (Behavior Interaction Priority) [13] allows rigorous design of complex component-based systems. BIP is supported by a toolset including translators of various source models to BIP, code generators, and verification mechanisms. BIP focuses on the design of systems based on the notion of interacting entities whereas our approach takes the point of view of the software developer, using classical UML-based descriptions augmented only by our graphical DSL for architecture, relying on notions the user knows well. Our approach is closely tied to the notion of distributed components interacting by requests and replies; while this reduces the field of applicability of our work, it allows us to generate the component interaction automatically, without additional input from the user.

Cadena[14] is a platform for the development of component-based applications, initially targeted for the Corba Component Model (CCM), and more recently extended to support OpenCCM, EJBs, and sensor networks specified with the nesC language. Cadena allows the user to specify component types, define and analyze inter-component dependencies, specify and model-check correctness properties, generate code in the various component formalisms, and even specify new user-defined component models. Unlike VerCors, it does not manage hierarchical components, so it could not be used for Fractal or GCM.

Palladio [15] is a tool for design, analysis and generation of hierarchical large-scale component-based systems. Palladio has less restrictions on types and allows more expressive modeling than VerCors. However, while Palladio has strong emphasis on simulation and system performance prediction, our approach benefits from the use of formal methods for validation.

Creol [16] is an object-oriented programming language based on concurrent objects that communicate asynchronously. Creol is supported by the Credo [17] toolset. In Credo the application description relies on Reo [18]. Credo provides an abstract but executable model of the application. Then, a test specification is derived to check compatibility between the two models. Creol is supported by a type-checker, a simulation and model-checking platform based on Maude. In VerCors we rely on UML-based formalisms, better known by the programmers than Reo. We also directly generate efficient code that can be executed on large-scale distributed infrastructures.

SOFA 2 [19] is a framework for distributed hierarchical component-based systems development. SOFA 2 is supported by a tool set comprising graphical designers and behavior validation instruments. SOFA 2 supports dynamic architectures, multiple communication styles and transparent distribution with the help of software connectors. Validation in SOFA 2 relies on behavioral protocols that are easy to understand for the programmer. This provides developers with validation capacities that require no expertise in any general logical formalism, though the expressivity may be lower than with temporal logic.

JHelena is a framework for modeling and generation of executable code of highly dynamic ensembles of autonomic distributed components that are modeled using Helena [20] technique. Our approach allows modeling systems with several levels of hierarchy while to our knowledge in Helena approach the composition only occurs at one level.

ABS [21] is a formal executable component modeling language supported by a deductive verification system Key-ABS. ABS is a powerful language for concurrent object-oriented programming, however it does not support any architectural description. The verification pattern is also quite different. Different tools for ABS either focus on specific properties (absence of deadlock for example) or use KeY to specify invariants of the program and verify them. Our approach allows us to target a wide range of properties while not asking the programmer to have

the expertise necessary to write program invariants.

Concerning actor systems, the related work the closest to ours is Rebeca [22] that handles both functional and real-time verification. The first main difference between Rebeca and Vercors is the programming model: Rebeca has no *future* and no synchronisation operation, which makes the generation of behavioural model easier. The second one is that the Rebeca toolset does not provide a design tool or an execution platform as efficient as Vercors+ProActive. On the other side, Rebeca has strong results concerning the scalability of the approach, and the range of systems and of properties handled.

Several verification tools focus on "real-time aspects" allowing to reason on the time-sensitive properties. In this section we have focused on the tools that explicitly handle asynchrony and we have not cited works on real-time systems in general.

# 7 Discussion and Perspectives

In this paper we presented our integrated environment for designing and implementing safe component-based systems. Our approach includes three main aspects. First, we provide graphical formalisms for the application architecture and the behavior specification, as well as type abstractions. The formalism extensively uses UML models that makes it easy to learn and use for the programmer. Second, we ensure behavioral correctness, by running a model-checker on the specified model. In practice, we transform graphical models into input for the CADP model-checker. As a result, the user can verify correctness properties of the modeled system even if he does not have a strong expertise in formal methods. Finally, we transform the models into executable application code. We implemented our approach in the VerCors platform and we tested it by modeling, verifying, and executing Peterson's leader election algorithm. Our approach was illustrated by generating GCM/ProActive code but it would be easy to generate code for any actor or active-object based language, or more generally any programming model made of components interacting by asynchronous requests and replies. Beyond the academic example of this paper, we have also published a study of a fault-tolerant protocol [23], showing how to handle scalability issues in the model-checking activities. In another paper, we showed an industrial-inspired study [24] in which we handle large state-spaces modeling an application with dynamic reconfiguration of components.

This paper raises the question of the relation between the semantics of the handled models: state-machines, pNets, finite-state models, and distributed Java programs. Previous usecases show that many applications and protocols can be encoded faithfully and executed correctly. It is not in the scope of this paper to study the semantic gap between these models or to formally prove that the behavioral model has the same semantics as the generated code. However, the formal semantics of ProActive [25], the semantics of pNets [5], and the formal definition of the translation from GCM to pNets [6] allowed us to check carefully that the semantics correspond faithfully. Considering the complexity of the system, an exhaustive formal proof of bisimulation between the semantics would require several years.

While creating the VerCors platform we tackled a number of challenges. First, the choice of the underlying technology was not trivial: we experimented with the Topcased platform, UML profiles, Eclipse Papyrus, before finding a usable environment with Sirius. Second, finding an expressive and easy to learn graphical formalism was a challenging task. We wanted to reuse UML notions as much as possible, but soon we realized the need for our own graphical ADL formalism, and we had to find a way to map a large part of GCM specifications into UML models. Finally, the integration of all languages, models and formalisms involved in modeling, execution and verification was not trivial. For example, we had to restrict the syntax of State Machine to

be able to translate them into Fiacre.

We are currently working on extensions of the VerCors platform that would address more features of distributed component-based applications. In particular, we want to address separation between functional code and application management and verify the correct interaction between those two aspects. Another challenge that we plan to address is the expression of the application properties using a higher level specification language. This should also include the translation from the model-checker diagnostics back to the user-level formalism, that is not implemented in the current version. This would make our approach even more attractive for users non-expert in model-checking.

# References

[1] Henrio, L., Kulankhina, O., Liu, D., Madelaine, E.: Verifying the correct composition of distributed components: Formalisation and Tool. In: FOCLASA, Rome, Italy (September 2014)

[2] Garavel, H., Lang, F., Mateescu, R., Serve, W.: Cadp 2010: A toolbox for the construction and analysis of distributed processes. In: TACAS'11. Volume 6605 of LNCS., Saarbrücken, Germany, Springer, Heidelberg (2011)

[3] Baude, F., Caromel, D., Dalmasso, C., Danelutto, M., Getov, V., Henrio, L., Pérez, C.: GCM: A Grid Extension to Fractal for Autonomous Distributed Components. Annals of Telecommunications **64**(1) (2009) 5–24

[4] Cansado, A., Madelaine, E.: Specification and verification for grid component-based applications: from models to tools. In de Boer, F.S., Bonsangue, M.M., Madelaine, E., eds.: FMCO 2008. Number 5751 in LNCS, Berlin Heidelberg, Springer-Verlag (2009) 180–203

[5] Henrio, L., Madelaine, E., Zhang, M.: pnets: An expressive model for parameterised networks of processes. In: 23rd Euromicro International Conference on Parallel, Distributed, and Network-Based Processing, PDP 2015, Turku, Finland, March 4-6, 2015. (2015) 492–496

[6] Ameur-Boulifa, R., Henrio, L., Madelaine, E., Savu, A.: Behavioural Semantics for Asynchronous Components. Rapport de recherche RR-8167, INRIA (December 2012)

[7] Dolev, D., Klawe, M., Rodeh, M.: An o(n log n) unidirectional distributed algorithm for extrema finding in a circle (1982)

[8] Barros, T., Ameur-Boulifa, R., Cansado, A., Henrio, L., Madelaine, E.: Behavioural models for distributed fractal components. Annals of Telecommunications **64**(1-2) (2009) 25–43

[9] Berthomieu, B., Bodeveix, J., Filali, M., Garavel, H., Lang, F., Peres, F., Saad, R., Stoecker, J., Vernadat, F.: The syntax and semantics of Fiacre. (March 2009)

[10] Lang, F.: Exp.Open 2.0: A flexible tool integrating partial order, compositional, and on-the-fly verification methods. In: Integrated Formal Methods, 5th International Conference, IFM 2005, Eindhoven, The Netherlands, 2005. (2005) 70–88

[11] Mateescu, R., Thivolle, D.: A Model Checking Language for Concurrent Value-Passing Systems. In Cuellar, J., Maibaum, T., eds.: FM 2008. Volume 5014., Turku, Finland, Springer Verlag (May 2008) 148–164

[12] Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: 21st International Conference on Software Engineering. (May 1999)

[13] Basu, A., Bensalem, B., Bozga, M., Combaz, J., Jaber, M., Nguyen, T., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3) (May 2011) 41–48

[14] Childs, A., Greenwald, J., Jung, G., Hoosier, M., Hatcliff, J.: CALM and Cadena: Metamodeling for Component-Based Product-Line Development. IEEE Computer **39**(2) (2006) 42–50

[15] Reussner, R., Becker, S., Burger, E., Happe, J., Hauck, M., Koziolek, A., Koziolek, H., Krogmann, K., Kuperberg, M.: The Palladio component model. Technical report, Karlsruhe Institute of Technology (march 2011)

[16] Leister, W., Bjork, J., Schlatte, R., Griesmayer, A.: Verifying distributed algorithms with executable Creol models. (January 2011)

[17] Grabe, I., Jaghoori, M.M., Aichernig, B.K., Baier, C., Blechmann, T., de Boer, F.S., Griesmayer, A., Johnsen, E.B., Klein, J., Klüppelholz, S., Kyas, M., Leister, W., Schlatte, R., Stam, A., Steffen, M., Tschirner, S., Xuedong, L., Yi, W.: Credo methodology: Modeling and analyzing A peer-to-peer system in Credo. Electr. Notes Theor. Comput. Sci. **266** (2010) 33–48

[18] Arbab, F.: A behavioral model for composition of software components. L'OBJET **12**(1) (2006) 33–76

[19] Hnětynka, P., Plášil, F.: Dynamic reconfiguration and access to services in hierarchical component models. In: Proceedings of the 9th int. conference on Component-Based Software Engineering. CBSE'06, Springer-Verlag (2006)

[20] Klarl, A., Hennicker, R.: Design and Implementation of dynamically evolving ensembles with the HELENA framework. In: Proceedings of the 23rd Australasian Software Engineering Conference, IEEE (2014) 15–24

[21] Hähnle, R., Helvensteijn, M., Johnsen, E.B., Lienhardt, M., Sangiorgi, D., Schaefer, I., Wong, P.Y.: HATS Abstract Behavioral Specification: The architectural view. In: Proc. 10th International Symposium on Formal Methods for Components and Objects (FMCO 2011). LNCS 7542, Springer-Verlag (2013) 109–132

[22] Sirjani, M., Movaghar, A., Shali, A., de Boer, F.S.: Modeling and verification of reactive systems using rebeca. Fundam. Inform. **63**(4) (2004) 385–410

[23] Ameur-Boulifa, R., Halalai, R., Henrio, L., Madelaine, E.: Verifying safety of fault-tolerant distributed components. In: International Workshop on Formal Aspects of Component Software (FACS'11), Oslo (Sept 2011)

[24] Gaspar, N., Henrio, L., Madelaine, E.: Formally reasoning on a reconfigurable component-based system - a case study for the industrial world. In: International Symposium on Formal Aspects of Component Software (FACS 2013). Lecture Notes in Computer Science, Nanchang, China, Springer (Oct 2013)

[25] Caromel, D., Henrio, L.: A Theory of Distributed Objects. Springer-Verlag (2005) ISBN 3-540-20866-6.

# A Class diagram

The appendix includes the UML Class diagram modeled in VerCors for an application consisting of four components that implement leader election algorithm.

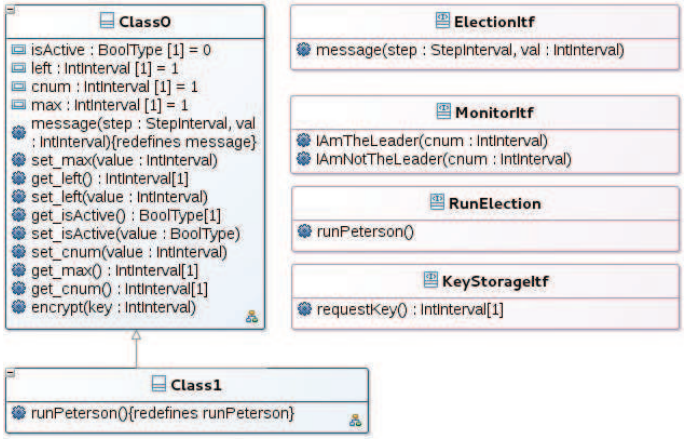



Figure 7: UML Class diagram of the use-case

# B Types diagram

The appendix includes the Type diagram modeled in VerCors for an application consisting of four components that implement leader election algorithm. The diagram includes specification of two integer intervals: StepInterval and IntInterval. The former is used as a type of leader election algorithm's phase. The letter is used for the unique identifiers of the components

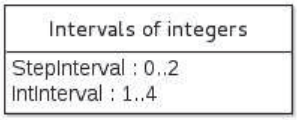



Figure 8: UML Class diagram of the use-case

# C Examples of pLTSs

The appendix provides examples of pLTSs generated from VerCors tool for Peterson's election algorithm use-case. Two examples are presented: the first one illustrates a pLTS produced for a fragment of a user-defined State Machine, the second example demonstrates a generic pLTS following a pattern of body pLTS generation.

Figure 9 illustrates a simplified State Machine of *message* operation behavior corresponding to the actions upon *message(1, value)* call, i.e. first phase of the election algorithm. Figure 10 illustrates a pLTS generated for the State Machine fragment.

Figure 11 shows a pLTS of `Comp4` body. The pLTS has three actions for every server operation of the component. *?Serve_ operation(...)* is performed when *operation* call is forwarded from

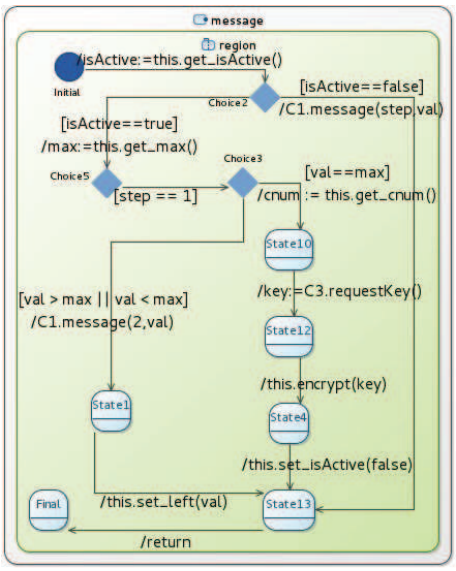



Figure 9: State Machine of the 1st phase of election algorithm

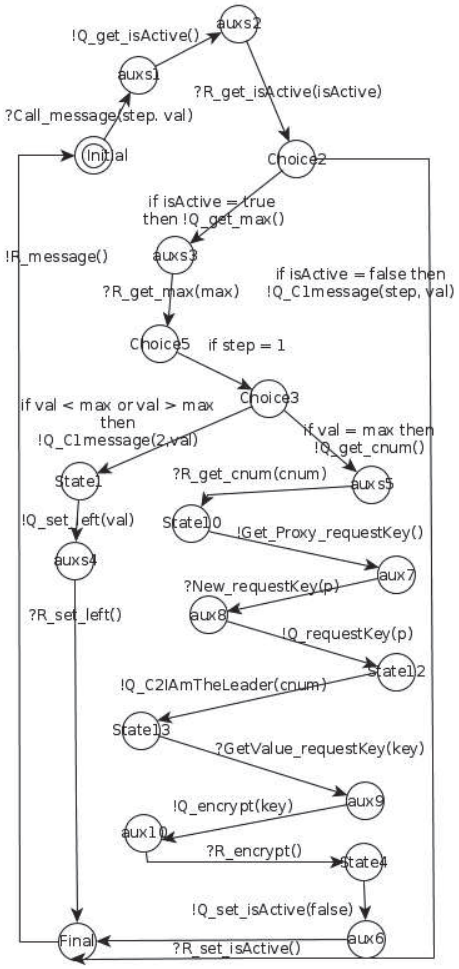



Figure 10: pLTS of the 1st phase of election algorithm

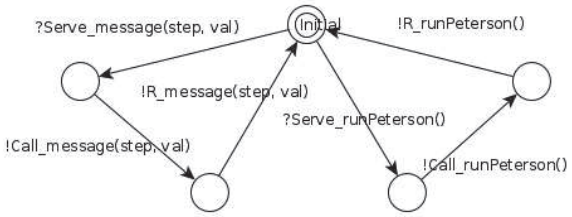



Figure 11: pLTS of Comp4 Body

queue to the body. *!Call_ operation(...)* calls the *operation* execution on the serving pLTS. Finally, *!R_ operation()* is used to synchronise the end of *operation* execution and the body.

# D Synchronization vectors

The appendix presents examples of synchronization vectors generated by VerCors for our use-case consisting of four components that implement leader election algorithm.

We present below the synchronization vectors generated for `Comp1` from Figure 2. The pNet of `Comp1` (Figure 4) is constructed of the pLTSs modeling the behavior of the queue, body, server method *message(...)*, local method *encrypt()*. four attribute controllers, proxy manager and proxy for *requestKey()* client method. Proxy manager and proxy pLTSs are created for every non-void client method in order to implement futures mechanism. More precisely, proxy manager allocates new proxies. A proxy receives the result of the remote method invocation. This result can be then accessed by the server (or local) method of the component.

The sub-pLTSs in our example occur in the following order:
$< Queue, Body, message, encrypt, max_ac, cnum_ac, left_as, isActive_ac,$
$Comp1_requestKey_Proxy_Manager, Comp1_requestKey_Proxy >$.

- **Queue and environment**. The vectors allow the queue to receive requests and to report about an error if a queue is saturated.
  $<?Q_message(...), -, -, -, -, -, -, -, -, - >\rightarrow ?Q_message(...)$
  $<!Error_Queue(...), -, -, -, -, -, -, -, -, - >\rightarrow !Error_Queue(...)$
- **Queue and body**.Queue and body are synchronized when the body takes a request that needs to be served from the queue.
  $<!Serve_message(...), ?Serve_message(...), -, -, -, -, -, -, -, - >\rightarrow Serve_message(...)$
- **Body and server methods**. Synchronization occurs when the body forwards the request to the server method pLTS that should serve it and when the result of the request is provided. Even if a server method does not return any result, the method pLTS synchronizes with the body at the end of the execution in order to impose mono-threaded processing of requests.
  $< -, !Call_message(...), ?Call_message(...), -, -, -, -, -, -, - >\rightarrow Call_message(...)$
  $< -, !R_message(...), !R_message(...), -, -, -, -, -, -, - >\rightarrow R_message(...)$
- **Methods and attribute controllers**. If a server or a local method calls another local method (in particular attribute controller get and set methods) and the latter returns the result, their pLTSs are synchronized. The information about which pLTSs should be synchronized can be retrieved from the state machine of the methods.
  $< -, -, !Call_Get_max(), -, ?Call_Get_max(), -, -, -, -, - >\rightarrow Call_Get_max()$
  $< -, -, ?R_Get_max(), -, !R_Get_max(), -, -, -, -, - >\rightarrow R_Get_max()$

- **Methods and environment**. Invocation of a void client method does not need futures mechanism, hence, it can be forwarded directly from the pLTS of the caller to outside of the component.
  $< -, -, !Q_IAmTheLeader(...), -, -, -, -, -, -, - > \rightarrow !Q_IAmTheLeader(...)$
- **Methods and Proxy Managers**. Whenever a non-void client method is invoked by a server or a local method, the pLTS of the caller is synchronized with the pLTS of the proxy manager that controls the futures mechanism for the corresponding client method. The caller's pLTS asks the proxy manager to allocate a new proxy.
  $< -, -, !Get_Proxy_requestKey, -, -, -, -, -, ?Get_Proxy_requestKey, - > \rightarrow$
  $Get_Proxy_requestKey$
- **Methods, Proxy Managers and Proxies**. When a new proxy is allocated, it synchronizes with the pLTS of the server or local method that will use this proxy (i.e. the method that invoked the client operation). $< -, -, ?New_requestKey, -, -, -, -, -, !New_Proxy_requestKey, ?New_Proxy_requestKey > \rightarrow New_Proxy_requestKey$
- **Proxy and environment**. The result of a client method invocation is first returned to the proxy allocated for this method invocation. In order to receive the result, the proxy pLTS synchronizes with the environment.
  $< -, -, -, -, -, -, -, -, -, ?R_requestKey > \rightarrow R_requestKey$
- **Methods and Proxies**. When a method needs the result of a client method invocation, its pLTS synchronizes with the pLTS of the proxy allocated for the remote method invocation.
  $< -, -, !Get_Value_requestKey, -, -, -, -, -, -, ?Get_Value_requestKey > \rightarrow$
  $Get_Value_requestKey$
  $< -, -, ?R_Get_Value_requestKey, -, -, -, -, -, -,$
  $!R_Get_Value_requestKey > \rightarrow R_Get_Value_requestKey$

The vectors of a **composite** synchronize queue and environment, queue and body under server method invocation and queue saturation in the same way as for the primitive components. We discuss below the rest of synchronization vectors and provide examples for the `Application` component pNet of Figure 5. The pNet includes the pLTSs of queue and body; three pLTSs controling futures mechanism for *requestKey()* client method: delegate, proxy manager and proxy; and pNets of four subcomponents. Delegate pLTS is generated for every non-void client method in order to delegate the method invocations to the corresponding proxy manager.

The subnets are given in the following order:
$< Queue, Body, requestKey_Delegate, requestKey_Proxy_Manager,$
$requestKey_Proxy, Comp1, Comp2, Comp3, Comp4 >$

- **Body and Subcomponents**. For every server method of a composite we synchronize the action where the body forwards the request to the pNet of the subcomponent that should serve the request.
  $< -, !Call_runPeterson(), -, -, -, -, -, -, ?Q_runPeterson() > \rightarrow !Call_runPeterson()$
- **Between Subcomponents**. If two subcomponents are connected by a binding, for each method of the client interface, we synchronize the method call and return of the result if there should be one.
  $< -, -, -, -, -, -, ?Q_message(...), -, -, !Q_message(...) > \rightarrow Comp4_Comp1_message(...)$
- **Queue and Subcomponents**. If a subcomponent invokes a client method, and the invocation should be forwarded to outside of the composite, the client request is first dropped to the queue of the composite. For this purpose the pNets of the subcomponent and the pLTS of the composite's queue are synchronized.
  $<?Q_IAmTheLeader(...), -, -, -, -, -, -, -, !Q_IAmTheLeader(...) > \rightarrow$
  $Comp4_Q_IAmTheLeader(...)$
- **Queue and environment**. Since the void client method invocations do not involve futures

mechanism, they can be forwarded from the queue directly to outside or the composite. For this purpose, the queue needs to synchronize with the environment
$<!Q_IAmTheLeader(...), -, -, -, -, -, -, -, - >\rightarrow !Q_IAmTheLeader(...)$

- **Queue and Body**. Except server requests, body also takes non-void client requests from the queue in order to forward them to the pLTSs implementing futures mechanism
$<!Serve_requestKey(), ?Serve_requestKey(), -, -, -, -, -, -, - >\rightarrow Serve_requestKey()$
- **Body and Delegate**. The body forwards a client method invocation to the delegate pLTS of the corresponding method.
$< -, !Call_requestKey(), ?Call_requestKey(), -, -, -, -, -, - >\rightarrow Call_requestKey()$
- **Delegate and Proxy Manager**. The delegate pLTS asks the proxy manager pLTS to allocate a new proxy.
$< -, -, !Get_Proxy_requestKey(), ?Get_Proxy_requestKey(), -, -, -, -, - >\rightarrow Get_Proxy_requestKey()$
- **Delegate, Proxy Manager and Proxy**. When a new proxy is allocated, the delegate pLTS that asked for the allocation is notified.
$< -, -, ?New_Proxy_requestKey(), !New_Proxy_requestKey(),$
$?New_Proxy_requestKey(), -, -, -, - >\rightarrow New_Proxy_requestKey()$
- **Delegate and environment**. Delegate pLTS synchronizes with the environment in order to send the client method invocation. The body of the composite should be notified about it, because from this moment it can take the next request from the queue.
$< -, !R_requestKey(), !Q_requestKey(), -, -, -, -, -, - >\rightarrow !Q_requestKey()$
- **Proxy Manager, Proxy and Subcomponents**. Once a remote request is served, its result is returned to the corresponding proxy and to the subcomponent that invoked the client method. After that, the proxy is not needed anymore and it can be recycled by the Proxy Manager.
$< -, -, -, !Recycle_requestKey(), ?R_requestKey(), -, -, -, ?R_requestKey() >\rightarrow ?R_requestKey()$

# E ADL example

The appendix illustrates an example of ADL specification generated by VerCors for our use-case model. A part of the file is given in Listing 1.3. A composite **Application** is represented in lines 1-20. A primitive **Comp1** is described in lines 5-15. Line 9 establishes a reference to the class of Comp1. Lines 11, 12 specify the attributes whose values are assigned during the application construction by the Factory. An example of a client interface specification is given in lines 6, 7. Line 17 describes a binding between two components inside a content.

Listing 2: ADL file

```
1  <definition name="Application">
2      <interface name="S1" role="server" signature=
3          "leaderelection.interfaces.RunElection"/>
4      ... other interfaces...
5      <component name="Comp1">
6          <interface name="C1" role="client" signature=
7                  "leaderelection.interfaces.ElectionItf" />
8          ... other interfaces...
9          <content class="leaderelect.classes.Class0"/>
10         <attributes signature="leaderelect.interfaces.Class0AC">
11             <attribute name="_max" value="1"/>
12             <attribute name="_cnum" value="1"/>
13         </attributes>
14         <controller desc="primitive"/>
```

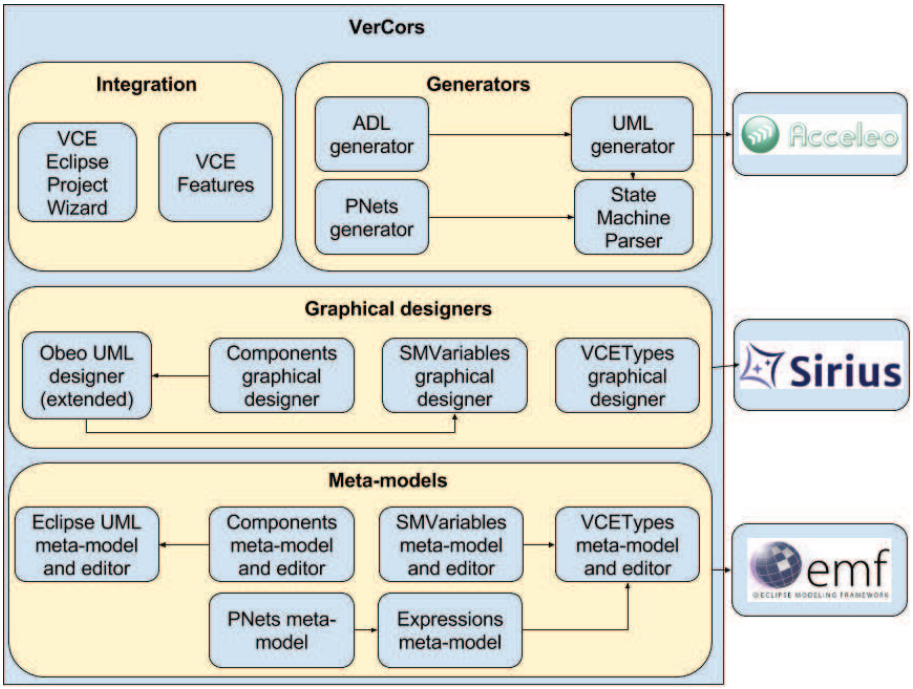



Figure 12: Architecture of VerCors

```
15      </component>
16    ... other components...
17      <binding client="Comp1.C1" server="Comp2.S1"/>
18    ... other bindings....
19      <controller desc="composite"/>
20  </definition>
```

# F Architecture of VerCors

VerCors is implemented as a set of plug-ins for Eclipse; its architecture is illustrated at Figure 12. The sub-modules of VerCors can be divided into four categories based on their functionality: meta-models and their editors, graphical designers, generators and integration plugins.

VerCors relies on six meta-models based on EMF Ecore[9]. **Components** meta-model is used for component-based system architecture specification. It structure reflects GCM Components structure. Components meta-model references **Eclipse UML**[10] meta-model for Classes, Interfaces and components behavior specification. Additionally, we implemented **SMVariables** meta-model for variables declaration on UML State Machines. The types used by State Machine variables and UML Operations signatures are based on **VCETypes** meta-model. Its root element *VCEType* extends UML *Type* which allows using VCETypes by UML elements. **PNets** meta-model is used for pNets construction. Finally, pLTS' labels are partially based on **Expressions** meta-model. Its structure reflects the grammar of UML State Machine labels. We

[9]https://eclipse.org/modeling/emf/
[10]org.eclipse.uml2.uml

generated EMF editors for those structures that can be edited by the users of VerCors (i.e. Components, SMVariables and VCETypes). An EMF editor for UML models is included in the UML Eclipse plugin.

The four graphical designers provided by VerCors are completely based on Sirius platform and rely on meta-models described above. **Components** graphical designer where the user can graphically specify the architecture of his/her application. **Obeo UML** graphical designer is integrated into VerCors and can be used to define Classes that implement components, Interfaces that define signatures of methods of components' interfaces and State Machines that illustrate components' behavior. Obeo UML designer includes a number of other UML diagram editors (e.g. Use-case and Activity diagrams). They can be also used by the users of VerCors, but they will not be involved in code generation and model-checking. We extended Obeo UML State Machine diagram editor with tools and graphical representation for **Variables declaration**. Finally, **VCETypes** designer can be user for specification of such types as integer intervals, enumerations, records and arrays of fixed size.

The core part of VerCors platform is ADL+Java and pNets generators.

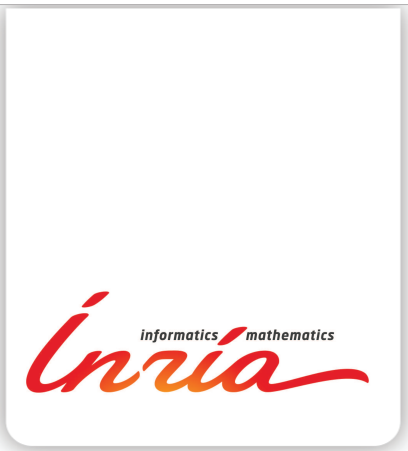
**ADL generator** takes Components diagram and a package name as input and produces an XML-based (ADL) file with the given architecture. The package name corresponds to the package where Java classes and interfaces will be produced. Then, ADL generator invokes **UML generator** that analyzes UML, VCETypes and Components models and uses Acceleo templates in order to produce Java classes and interfaces.

For every set/get method of a UML class, UML generator produces the corresponding Java code. For every method which behavior is illustrated by a State Machine diagram, UML generator invokes **State Machine parser** that parses labels of state machine transitions in the context of a given Components architecture. The context is used to establish references to the signatures of methods invoked by the state machine. The parsed state machine is then translated into Java code of the method.

**PNets generator** takes thae following input: components architecture and referenced UML elements, scenario state machine if there is any, queue size for each component, interactions that are not observed during model-checking. Then, it processes the input as follows:

1. **Pre-procession**. PNets generator analyses each composite component and gathers auxiliary information. For every server interface it finds the subcomponent that will process the requests. For every client interface it finds subcomponents that could send the request. Then, a State Machine Parser is invoked to parse labels of all state machines and gather information about local and remote methods invoked by each state machine.
2. **PNets generation**. Starting from the root component, pNets generator recursively produces pNets for every component. More precisely, for a composite, it generates pLTSs of internal processes (body, attribute controllers, etc), a set of synchronization vectors and triggers pNet generation for each subcomponent. For a primitive, it produces pLTSs of internal processes and a set of synchronization vectors. Scenario state machine is also translated into a pLTS. Synchronization vectors of the root component include synchronization with the scenario.
3. **Fiacre generation**. Every generated pLTS is translated into a Fiacre file.
4. **EXP generation**. A set of synchronization vectors of every pNet is translated into an EXP file.
5. **Auxiliary scripts generation**. For every pNets we generate a script assembling its subnets into a common structure with respect to the synchronization given in the corresponding EXP file. The scripts also hide communications that should not be observed during model-checking.

Finally, integration modules are used to integrate VerCors in Eclipse. **VCEWizard** implements a wizard creating a VerCors project with a Components, UML and VCETypes model files and one diagram illustrating each model. The user can then add other models and diagrams. **VCE Features** module makes VerCors installation/update accessible via standard Eclipse plug-in installation/update wizard.




**RESEARCH CENTRE**
**SOPHIA ANTIPOLIS – MÉDITERRANÉE**

2004 route des Lucioles - BP 93
06902 Sophia Antipolis Cedex

Publisher
Inria
Domaine de Voluceau - Rocquencourt
BP 105 - 78153 Le Chesnay Cedex
inria.fr

ISSN 0249-6399