Mapping Component Models on Distributed Architectures: Correctness Checking

Christian Attiogbé
LINA - UMR CNRS 6241 - University of Nantes
F-44322 Nantes Cedex, France
christian.attiogbe@univ-nantes.fr

***Abstract*—We propose a method to check if a distribution of components, proposed to implement a given assembly of components, is correct with respect to the targeted host distributed architecture. The method is based on the principle of checking that the accessibility relation induced by the proposed distribution of components, is consistent with the accessibility relation imposed by the designed component model. The benefit is twofold: either to refine the designed component model or to adapt the envisioned deployment. The study is illustrated with examples of component models. The Event-B framework is used to check correctness, and we have developed a simulation prototype tool to support experimentations.**

keywords: Abstract Component Model, Distributed Architecture, Verification, Deployment, Event-B

I. Introduction

Context. Component-based software development still poses many challenges among which the correctness of the deployment of a given component-based design on a targeted execution architecture. More specifically, distributed applications are made of components which services are linked together through a network, with a provide-require relationship which emphasizes a dependency or an accessibility feature.

In the current work we deal with one of the challenges: the correctness of the projection[1] of an assembly of components on a host distributed architecture. Indeed the interaction between the involved services of the components may be broken in case of an inappropriate distribution of the components on the nodes of the distributed architecture. We argue that the correctness of the whole development process from the design step to the deployment step can be assisted by rigorous and pragmatic methods and tools. This work follows a series where we experiment on formal analysis of component-based models, using especially the Event-B method [1], [2].

An assembly of components, is the specification of interacting component services; a service of a component will be running on a host node; accordingly an assembly of components and their services will be running on various nodes of a distributed architecture. The concern is as follows: *is a given projection or a deployment of the assembly on an existing targeted network of nodes is correct or not?* This aspect is not covered by the numerous works, proposals and industrial solutions[2] that already exist; indeed they mostly focus on low level and platform-specific features (versions, file systems, runtime details, etc.) for the deployments of components.

We are motivated by the need of such methods and tools to ensure that an abstract model of an assembly of components is well distributed on a host architecture which is viewed as a graph of nodes. We can then ensure that the deployment and its future evolution preserve the design time dependency requirements. Indeed highly distributed or mobile platforms demand very efficient component deployment techniques. Besides, in the case of maintenance or evolution of the host architecture, a redeployment of components may be performed for an already deployed component-based application; thus on-the-fly deployment of components, needs a correctness checking to ensure that the redeployment will be free of inconsistencies. A model-based correctness checking is likely to ensure confidence; this perspective is considered as preliminary step before unit deployment of components and their bindings.

Starting from an abstract model of an assembly of components and a proposed distribution related to a target architecture, we show how the proposed distribution can be checked. We define the rules that ensure the correctness of a distribution; incorrect distribution can then be detected. To build our solution we consider that one input parameter, an assembly of components, involves an accessibility graph of the services provided and required by the components used in the assembly. In the same way, the second input parameter, a given distributed architecture intended to host the execution of application assemblies, is viewed as a directed graph of nodes. Therefore the proposed method is based on the principle of checking that the accessibility relation induced by the proposed distribution of components, is consistent with the accessibility relation required by the designed assembly. We have already used

[1] the projection is an envisioned deployment

[2] For instance Java and Corba CCM; the DLLs; .NET framework, the OMG's DC approach

Event-B to check some properties on component-based models, we follow this approach by considering here Event-B and its tools to illustrate the proposed method.

Contribution. The contribution of the current work is a pragmatic method to check the correctness of a distribution of an assembly of components on an existing physical network architecture; *i)* we define the correctness and a set of rules to check it; *ii)* we define a systematic method and an accompanying generic architecture based on Event-B to check the correctness, by reusing the consistency obligation proof of Event-B. The Event-B method being designed for a correct-by-construction approach, it is not straightforward to use it to check such a property independently; *iii)* we experiment on several cases with the Event-B generic structure and the Rodin tool; we have developed an accompanying prototype in Python for simulation purpose to get user-friendly feedback.

Structure of the article. In Section II we give the material necessary for our method. We introduce a general abstract component method and the Event-B formalism. Section III is devoted to the proposed approach to check the correctness of a distribution. Section IV deals with the experimentation conducted with Event-B and the developed prototype. Finally Section V concludes the article and comments on related and future works.

II. Background: Component Models and Event-B

A. An Abstract Component Model Core

We consider the common features of various component models in order to have a general approach independent of any specific component model. The minimal common features are: component, required service, provided service and assembly. Some component models permit several provided services or behaviours (like Kmelia [3], Fractal[4]).

Whatever the formalism a component model has at least a *provided service*; a service is a functionality often modelled as an automata. A component model may have one or several *required services* (those needed to achieve the functionalities or services which are provided).

Components are assembled to build larger models. An *assembly* is built by linking required services and provided services of different components. A basic semantics of a link between a provided service p and a required one r, is that the link denotes a possible call to the provided service s from a service of the component that declares the required service r.

A dependency relationship can be computed from any assembly of components, by considering that a component depends on another, if the last one provides at least one service to the former. Given an assembly A, extracting a dependency relationship between the components of A is straightforward: a component C_r depends on a component C_p if at least one service of C_r requires at least one service of C_p; that is C_r requires at least one service provided by C_p.

In this study, we restrict the description of components to their main features: a component is modelled by a set of provided services and a set of required services; an assembly is modelled by a set of components together with the links between some required services and provided services of different components. We do not need the other features of a component-based model. We do not consider the behavioural aspect of the services or component; but note that a behaviour of an abstract service is often described by a transition system. An assembly supports the interaction between linked services; this interaction is modelled by the composition of the labelled transition systems which describe the behaviour of the services.

A formal model of a core abstract component assembly

Formally we define an abstract model of an assembly of components as a directed graph of components. We call *a link* the connection between two components via a service providing relationship, it is a dependency relationship. Let $I_p(c)$ be the set of the services provided by the component c and $I_r(c)$ the disjoint set of the services required by c. The union of $I_p(c)$ and $I_r(c)$ is the interface of c. A link between a component c_i and a component c_j exists when there exists a service of c_i which requests at least one service provided by c_j. Let $Component$ be a set of components.

$$componentModel : Component \leftrightarrow Component$$
$$\forall (c_i, c_j) \in componentModel \;.\; I_p(c_i) \cap I_r(c_j) \neq \varnothing$$

The relation[3] *componentModel* describes not only the assembly of the components, but also the accessibility relationship between the components. If (c_i, c_j) is a member of *componentModel*, then c_i can access c_j, that means c_j provides a public service accessed (or called) by a service of c_i.

This core abstract model can be easily extended with the other features (detailed signatures, properties, service behaviours, etc) of a component model without breaking the forthcoming correctness checking method.

B. An Overview of the Event-B Method

Event-B [5], [6] is a modelling and development method where abstract machines are constructed and refined into concrete machines. An *abstract machine* describes a mathematical model of a system behaviour[4]. In an Event-B modelling process, abstract *Machines* constitute the dynamic part whereas *Contexts* are used to describe the static part. A *Context* is seen by machines. It is made of carrier sets and constants. It may contain properties (defined on the sets and constants), axioms and theorems.

[3]The symbol $\leftrightarrow$ denotes a relation; dom (resp. ran) denotes the domain (resp. codomain) of a relation.

[4]A system behaviour is a discrete transition system

A machine is made of variables, invariants and several *event* descriptions.

a) State Space of a Machine: The variables constrained by the invariants describe the state space of a machine. The change from one state to the other is due to the effect of the events of the machine. Specific properties required by the model may be included in the invariants. $I(x)$ where x is the state variables, denotes the invariants of the machine.

b) Events of an Abstract Machine: Within Event-B, an event is considered as the observation of a transition. Events are spontaneous and show the way a system evolves. An event e is modelled as a *guarded substitution*: $e \widehat{=} Guard \Longrightarrow Body$ where $Guard$ is a predicate which describes the conditions in which the event may occur and $Body$ is the action which is achieved when the event occurs.

An event may occur only when its guard holds. The action of an event describes with simultaneous generalized substitutions, how the system state evolves when this event occurs: disjoint state variables are updated simultaneously.

c) Rodin Tool: The Rodin[5] tool is an open tool dedicated to building Event-B models and to formal reasoning on them. It is made of several modules (plugins) to work with Event-B models or to interact with related tools.

III. The Proposed Distribution Checking Approach

In this section we define a systematic method to check, giving a component-based model, a target distributed host architecture and a proposed distribution of the components on the host architecture, if the proposed distribution is correct or not. We define for this purpose the structure of a host architecture and what is a correct distribution.

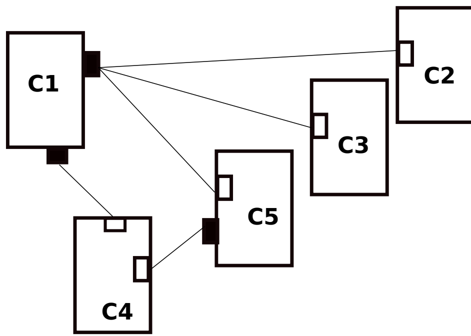


Fig. 1. An abstract component model

A distribution of components on a host physical architecture

The physical host architecture is a set of nodes that will host the components (as depicted in Fig.2). Remember that a *directed graph* is a set of nodes, connected by a set of edges which have a direction. The physical architecture is modelled as a directed graph of nodes representing the host machines. Here the nodes are the machines that really exist and are physically connected on a network; they can access each other following the direction of the edges. We use a function[6] *supportNode* to model the fact that a component is (proposed to be) deployed on one node; consequently several components can be deployed on the same node, but a component cannot be deployed on several nodes.

$physArchitecture : Node \leftrightarrow Node$
$supportNode : Component \rightarrow Node$

The relation *physArchitecture* describes a directed graph. When (n_i, n_j) is in *physArchitecture* then the node n_i can access n_j; consequently the components supported by n_i can access the components hosted by n_j. Moreover the nodes in the architecture are those used to host the components[7].

$\mathsf{dom}(physArchitecture) \cup \mathsf{ran}(physArchitecture)$
$\quad \subseteq \mathsf{ran}(supportNode)$

A. Checking the Correctness of a Distribution of Components: the Principle

First we consider the case of a normal deployment; that means without consideration of failure on the targeted architecture. The case with the risk of failure is considered later in the article. Figure 1 depicts an example of a component-based model to be deployed; the ■ denotes a provided service; a □ denotes a required service. Fig.2 depicts a target physical architecture where the filled nodes represent the ones used as support of component deployment. In Fig.3 we have a proposed distribution of the components on the nodes.

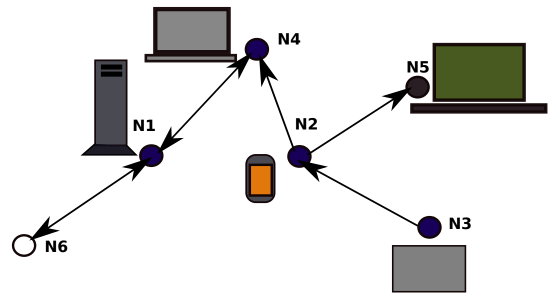


Fig. 2. A target physical architecture

Assume that we have the component-based model of a distributed application with the components and their client-server relationship as shown in Fig. 1 and a physical

[5]Rodin http://wiki.event-b.org/index.php/Main_Page

[6]denoted by the symbol →

[7]The set operators: card, dom, ran mean cardinal of a set, domain and range of a relation

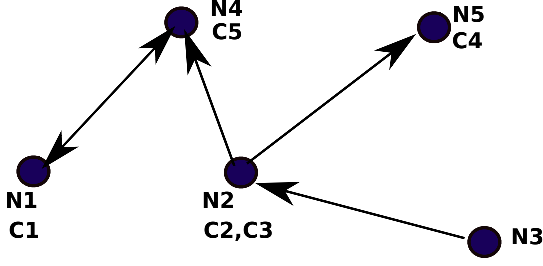


Fig. 3. An abstract view of the target architecture

host architecture (Fig.2); A deployment proposed in Fig. 3 is: $[(N1, \{C1\}); (N2, \{C2, C3\}); (N5, \{C4\}); (N4, \{C5\})]$.

The correctness checking approach is based on the following set of hypothesis and rules.

- The component model is represented by a set of links between the services of the components; this is denoted by a relation between the components.
- The link between the components via their services denotes a dependency relation between the involved components.
- The targeted physical architecture is represented by an accessibility graph of nodes.
- A node can support the deployment of zero, one or several components. Consequently the number of nodes used for the deployment may be less than the number of components to be deployed.
- An abstract component-based model can be deployed on only one node (in the particular case where we do not have distribution).

Definition (CorrectDistribution) A distribution of the abstract model of a component assembly is correct if

- all the components of the component model are deployed on the nodes of the physical architecture in such a way that,
- the client-server relationship between the services of the abstract model is preserved in their distribution on the physical architecture.

Basic rules for the distribution correctness

Let *Component* and *Node* be the set of components and the set of nodes; in the following c_i are elements of *Component* and n_i are elements of *Node*. According to Definition III-A, the correctness of a proposed distribution is established by the following rules.

$\mathsf{R}_{existPhys}$. There is at least one node in the target physical architecture:

$$\mathsf{card}(physArchitecture) > 0$$

$\mathsf{R}_{totalDepl}$. Each component is deployed on one node and all the components of an assembly are deployed. This is already obtained by defining *supportNode* as a total function.

$$\mathsf{dom}(componentModel) \cup \mathsf{ran}(componentModel) \subseteq \mathsf{dom}(supportNode)$$

$\mathsf{R}_{allLinkAccess}$. For all link (c_i, c_k) in the component model, the node where c_i is deployed has access to the one that supports c_k; that means $(supportNode(c_i), supportNode(c_k))$ is in the accessibility graph of the physical architecture.

$$\forall (c_i, c_k) \in componentModel \; . \; (supportNode(c_i), supportNode(c_k)) \in physArchitecture^{*8}$$

$\mathsf{R}_{transAcces}$. Transitivity: Closure of the accessibility. If the accessibility graph contains the edges $(supportNode(c_i), supportNode(c_j))$ and $(supportNode(c_j), supportNode(c_k))$, then the component c_i can access the component c_k.

$\mathsf{R}_{selfNodeAccess}$. Reflexivity. To enable the deployment and the mutual access of several components on the same nodes, we need the reflexivity of the accessibility such that any node n_i in the hosting network architecture can access itself.

$$\forall n_i \in Node . (n_i, n_i) \in physArchitecture$$

Consequently several components or a whole assembly can be deployed on a single node.

Illustration

Applying the rules on the example of Fig.3, we detect that the proposed distribution is wrong (at least one rule —$\mathsf{R}_{allLinkAcces}$— is not respected).

Inputs:

- The component assembly:
  $componentModel =$
  $\{(C2, C1), (C3, C1), (C5, C1), (C4, C1), (C4, C5)\}$
- The hosting network architecture:
  $physArchitecture =$
  $\{(N1, N4), (N4, N1), (N2, N4), (N2, N5), (N3, N2)\}$
- The proposed deployment (see Fig.3):
  $[(N1, \{C1\}); (N2, \{C2, C3\}); (N4, \{C5\}); (N5, \{C4\})]$

Output:

- Incorrect distribution,
- $C4$ lacks access to $C5$.

Indeed, $(C4, C5)$ is a link in the component assembly, but the node $N5$ where $C4$ is deployed does not have access to $N4$ which supports the component $C5$. The rule $\mathsf{R}_{allLinkAcces}$ is not respected. That

[8] the closure of *physArchitecture*

is $(supportNode(C4), supportNode(C5)) = (N5, N4)$ and $(N5, N4) \notin physArchitecture$ and $(N5, N4) \notin physArchitecture^*$. If the component $C4$ was proposed to be deployed on $N3$ instead of $N5$, we won't have a wrong distribution.

B. Checking the Correctness of a Given Distribution: the Method

Consider as input, an abstract component-based model A, a host distributed architecture H and a proposed distribution D of the abstract component model on the host architecture. We have to check that the distribution D is correct. The steps of the correctness checking are as follows:

1) Check that the physical architecture A exists, by applying the rule $\mathsf{R}_{existPhys}$.
2) Check that all the components of A are deployed, by applying the rule $\mathsf{R}_{totalDepl}$. In case of failure, the process stops.
3) Check that all the assembly links of A are supported by accessible nodes of H, by applying rules $\mathsf{R}_{allLinkAcces}$, $\mathsf{R}_{selfNodeAccess}$, $\mathsf{R}_{transAccess}$. In case of failure, the process stops.

The verdict of a correctness analysis may have various forms due to the step of the failure.

C. Handling Multi-services Component Model

A component may provide several services and require other ones. Accordingly we can have symmetric links and cycles of links between components. In the basic case presented above, (c_i, c_j) is in the assembly if a service s_k of c_i uses a service s_u of c_j. In the case where a service s_u of c_j also uses a service s_k of c_i we should have the link (c_j, c_i) in the description of the assembly, either directly via $physArchitecture$ or indirectly via the closure $physArchitecture^*$.

As far as the deployment is concerned, c_i and c_j should be deployed on nodes that are mutually accessible in order to have a correct deployment. The deployment will be incorrect if the components are deployed on nodes that are not directly or indirectly[9] connected. Consequently, the correctness rules of our method are appropriate to deal with multi-services components models.

D. Handling Fault Tolerance and Non-Functional Properties

A given component which is considered as critical may be deployed on a specific node or even deployed on more than one node in order to have a redundancy which favors failure management. In the same way, a given risky node or a given vulnerable node may be relayed by an emergency node. In order to handle a fault tolerance in case of a node failure, a component may be deployed on more than one node. This breaks the working hypothesis considered in the normal situation. The $supportNode$ function should be then transformed into a relation in order to express that a component may be deployed on several nodes.

To deal with this requirement of fault tolerance we adopt the following policy. When nodes are redundant, only one of them is active, that is, the tasks are performed by only one node which however, may be replaced by another one in case of failure. The adopted solution is that we require the description of the effective redundancy: which node is an emergency node of another? we use the notion of *active node* to support the deployed critical components; a function *state* is introduced to give the state of each node. Consequently the previous abstract model and rule set is increased with specific rules as follows.

$$
\begin{array}{l}
supportNode : Component \leftrightarrow Node \\
redundancy : Node \leftrightarrow Node \\
state : Node \rightarrow \{active, inactive\}
\end{array}
$$

$\mathbf{R}^{NF}_{oneNodeActive}$: If a component c_i is deployed on more than one nodes, then these nodes are redundant and at most one of them should be active[10].

$$
\begin{array}{l}
\forall c_i \in Component.\ (\mathsf{card}(supportNode(c_i)) > 1 \Rightarrow \\
\quad (\mathsf{card}(supportNode(c_i)\ \cap\ state^{-1}[\{active\}]) = 1))
\end{array}
$$

Concerning non-functional properties, a component may have some requirements with respect to the host support node; for instance the energy consumption, the battery autonomy or any other resource availability. The proposed models with the correctness rules together with the checking policy are easily extensible. We have to define the categories of requirements which are common to components and hosts. Let NF_c and NF_h be the functions that give respectively a given non-functional property of components (p_c) and hosts (p_h). Depending on the properties, a conformance relation $\mathcal{C}$ should be defined on properties in such a way that $\mathcal{C}(p_c, p_h)$ holds.

Assume that we have to deploy a component c_i with a requirement $P_i = NF_c(c_i)$ with respect to its host support; the conformance should be stated for P_i and $NF_h(supportNode(c_i))$.

$\mathsf{R}^{NF}_{PConform}$. All components are deployed on support nodes and each component is deployed on a host node that has the appropriate non-functional properties: the rule $\mathsf{R}_{totalDepl}$ should hold and moreover we should have:

$$
\begin{array}{l}
\forall c_i \in \mathsf{dom}(supportNode). \\
\quad (supportNode(c_i) \in \mathsf{dom}(NF_h) \wedge c_i \in \mathsf{dom}(NF_c)) \Rightarrow \\
\qquad \mathcal{C}(NF_c(c_i), NF_h(supportNode(c_i)))
\end{array}
$$

The current rule set can be easily extended as shown by the previous cases.

[9]bidirectional connection is not required since a connection can be indirect via the closure.

[10]When r is a relation, $r[s]$ denotes the image of the set s

IV. An Implementation with a Generic Architecture

We have implemented our approach using the Event-B framework to effectively check the correctness of a component distribution on a target architecture; it supports the following two key features:

- the formalisation of the model of the component assemblies, the formalisation of the host architecture, and the formalisation of the proposed distribution of components;
- the appropriate way to check the correctness by applying the introduced rules.

While the formalisation in Event-B of the used models is straightforward, a challenging point is *to find a specific Event-B structuring to handle the correctness via the defined rules.* Indeed the classical approach in B is to build a correct model with respect to an invariant; here the question is how to check a given model with respect to a set of rules involving other models.

On the one hand a model of a component assembly is expressed as a standalone Event-B abstract machine equipped with the relation and function $componentModel$, $supportNode$ (see Section II-A) describing the architecture of the component model. On the other hand a host distributed architecture is expressed as another standalone Event-B abstract machine that describes the graph of the architecture.

The input distribution proposed for the assembly, depends on both machines. We describe the distribution as a relation between the components (described in one machine) and the nodes of the host architecture (described in the other machine). We have studied several solutions in order to exploit the Event-B consistency checking for the rule-based correctness checking:

i) From the involved input structures, an event can be enabled to notice the correctness of the structures: here the correctness definition should be put in a guard of an event;

ii) An enabled event can notice that the input structures do not respect the definition, and the abstract machine deadlocks when the input structures meet the definition;

iii) From the involved input structures we build a machine with an invariant that states the definition of the correctness; any values of the state variables that establish the invariant will be correct. The issue here is that the machine should be initialised (and rechecked) with the distribution one wants to check; another solution is to use an non-deterministic initialisation that establishes the invariant and then to use a simulation tool to find the solution of initialisations (the ProB tool is able to deal with this constrained initialisation); this solution can be used to compute possible distribution.

iv) The defined rules that establish the correctness definition can be integrated in the invariant of a machine, with a generic initialisation. In this case, a correct distribution (used as an input Event-B model) will be proved correct whereas an incorrect distribution will not be proved;

v) The different rules can be integrated in the guards of events with their negative form; these events will be enabled if the rule that they denote is not true.

The last two solutions were retained; the solution *iv)* is simpler to implement than the others and, it helps checking gradually the correctness. The last solution is retained for debugging/simulation purpose (see Section IV-B), an enabled event raises the failure of a rule.

To ease the assessment of the method, we propose a reusable generic Event-B architecture (see Fig.5) to model this framework dedicated to the implementation of our method of correctness checking. Indeed, all the used models are treated as parameters and separated from the Event-B machine that contains the correctness checking rules. Therefore the machine on the top (`CheckDistrib`) is defined once for all, with the described correctness policy; it contains all the rules to be checked for the correctness of deployment (see Fig.6). In the same way, the abstract component model is defined once for all in the machine `CompoModel` (see Fig.5); here the structuring rules of components and their assemblies are formalised and may be extended independently from any further deployment checking rules. Besides, we have the network of nodes which will be used to host a deployment of a given component assembly; it is also a standalone machine `PhysArchi`. Note that a given distribution to be submitted for checking, needs both the machines `CompoModel` and `PhysArchi` which are therefore the parameters.

The leaf context machines (see Fig.5) contain the specific parameters values of an application context (for example the names of components, the names of host machines); they do not need to be changed. The intermediary context machine (`Distribution`, see Fig.4) contains the input structures that one wants to check; only this context machine is changed to reflect another given input. This is achieved very simply as an initialisation of variables as shown in Fig. 4.

A. Proving a Distribution Correctness via Consistency Checking in Event-B

A synoptic of a reusable architecture for checking the correctness of a given distribution is depicted in Fig.5. It is generated by the Rodin[11] framework which is used to perfom the experimentations.

The main point is how one can check, using the Event-B tools, that the rules defined to establish the correctness of a distribution are respected. The simple idea is to reuse the

[11] http://wiki.event-b.org

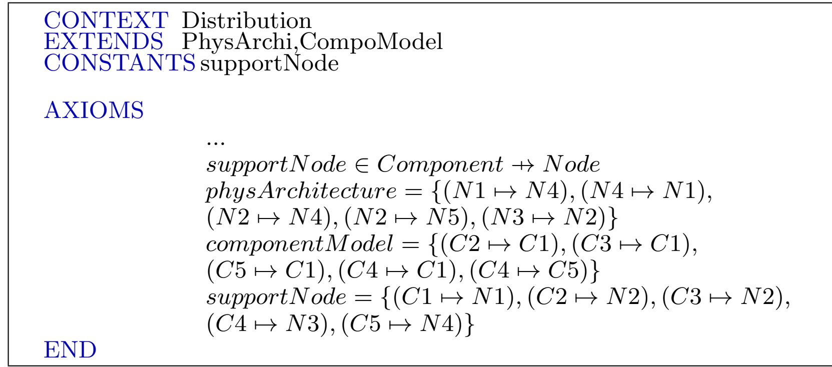


Fig. 4. The machine dedicated to a specific distribution to be checked

consistency checking of the Event-B approach to establish the correctness of the distribution. Accordingly we have defined an abstract machine equipped with a rule-based invariant and parameterised by the distribution that we have to check. Therefore all the rules that we have to check have been put into the invariant. Consequently if the machine is proved correct then we conclude that the given distribution is correct since the invariant is made of the correctness rules.

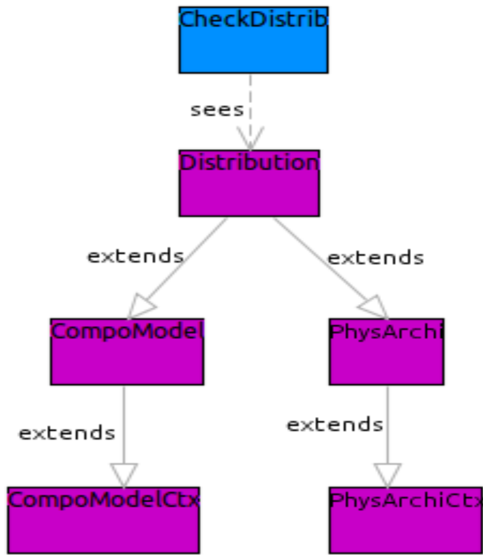


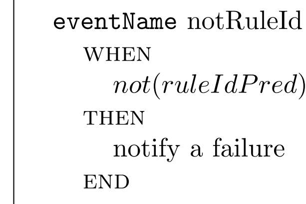
Fig. 5. The reusable architecture for checking distribution correctness

The proposed architecture is generic and it enables one to manage the parameterisation of the distribution checking process without changing anything in the main machine dedicated to the verification process. The parameter of the generic structure is the machine `Distribution`; it is the only machine to be changed according to a new distribution. The other machines are defined once and they remain unchanged.

In the same way, the used component-based model and the structure of the physical architecture are separated from their valuations which are done via the extensions of contexts as shown in Fig.5. The leaf layer context machines of the structure contain the fixed contexts for component assemblies and for host architectures.

B. A Simulation Approach

Using the same generic architecture, we simulate the correctness checking using the Event-B tool. The adopted solution is as follows; for each rule defined by a predicate (say $ruleIdPred$), we build an event named notRuleId; the event is guarded with the negation of the rule that it checks. The shape of the event is as follows:



The machine equipped with these events (one for each rule) is used to simulate the correctness checking. Indeed when one rule is not respected, the related event is enabled. This approach is very practical and helpful; when we use the tools such as ProB[12], we get immediately the rules which are not true as a feedback of the correctness checking.

C. Implementation with a Prototype Tool

Based on the proposed correctness definition we have developed a prototype tool to experiment quickly with examples. The input of the tool is as described in Section III-B: a model of an assembly, the host architecture and the proposed deployment. The prototype is developped with the Python language. This complementary approach provides a feedback as a graphical display (see Fig.9) of the distribution of components on the host architecture, if the distribution is correct; otherwise the feedback makes the wrong link explicit, as shown in Fig.7.

[12] www.stups.uni-duesseldorf.de/ProB/

MACHINE CheckDistrib

SEES Distribution

VARIABLES

$failure$

INVARIANTS

$failure \in BOOL$

$card(physArchitecture) \geq 1$

/* Rule $\mathrm{R}_{existPhys}$ */

$dom(componentModel) \cup ran(componentModel) \subseteq dom(supportNode)$

/* Rule $\mathrm{R}_{totalDepl}$ */

$\forall ci, ck \cdot (ci \in Component \wedge ck \in Component$
$\wedge (ci \mapsto ck) \in componentModel) \Rightarrow$
$(supportNode(ci) \mapsto supportNode(ck)) \in$
$(physArchitecture; physArchitecture)$

/* Rule $\mathrm{R}_{allLinkAccess}$ */

EVENTS

...

END

Fig. 6. The machine gathering the rules to check the correctness

```
support nodes   C4 : N5 |-> C1 : N1
access needed from C4 to C1
no access from N5 to N1
check link  C4 |-> C5
support nodes   C4 : N5 |-> C5 : N4
access needed from C4 to C5
no access from N5 to N4
Incorrect distribution
```

Fig. 7. An example of feedback for an incorrect distribution

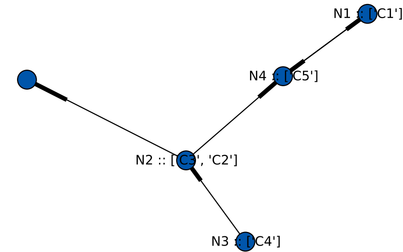


Fig. 9. An example of feedback for a correct distribution

The simulation of the example introduced in Section III with the prototype goes as follows; first the tool displays the submitted inputs (Fig. 8) and then displays the distribution (Fig. 9).

```
----------- The inputs are ---------
Assembly Description:
[('C3', 'C1'), ('C2', 'C1'), ('C5', 'C1'),
      ('C4', 'C1'), ('C4', 'C5')]
Target Network:
[('N1', 'N4'), ('N2', 'N4'), ('N2', 'N5'),
      ('N3', 'N2'), ('N4', 'N1')]
Proposed deployment of components on network nodes:
'C3': 'N2', 'C2': 'N2', 'C1': 'N1',
      'C5': 'N4', 'C4': 'N3'
```

Fig. 8. An example of feedback for a correct distribution

V. Conclusion

Summary of the results. We have proposed a method to check at preliminary step of deployment process, that a given distribution of an assembly of components on a host architecture is correct. The method is based on the inputs of an abstract model of the assembly, an abstraction of the host architecture and a proposed distribution of the components. This distribution can be viewed as a specific deployment of the assembly. The early verification of the mapping between the assembly and the target host architecture can save considerable effort at design time. In the context of a highly evolving architecture due to the mobility of some devices, it is very important to study the deployment of components and services in such a way that the provided services continue to be available. The verification of correctness can happen in different situations: to help in configuring the assembly, to check the robustness of an assembly, to study or to debug an assembly. Very often the physical network to host forthcoming applications exist with accessibility constraints. The deployment of new applications should be compliant with this accessibility relation.

This work complements the various properties that one should check on the assemblies of components such as: structural correctness of the assembly, correctness

of behavioural interactions, functional properties, non-functional properties.

Related works. The contributions presented here should be linked with the numerous proposals to tackle the complexity of component-based software deployment. We have addressed the preliminary steps of the process; other works focused on many other steps such as dependencies for installability of components[7], runtime deployment, deployment constraints in architectural language[8], [9], unit deployments, version management, predictability of deployment latency[10], synthesis of deployment solution based on multi-criteria requirement [11], realtime constraints and embedded systems [13], etc. A synthetic view of some aspects of component deployment is presented in [12], but distributed deployment is not dealt with. In [13] the authors propose an incremental approach of deployment of model level concepts into runnable entities (*i.e.* how functionality is distributed over the nodes of a target system) via an intermediate level of virtual nodes. Their approach is not formalised. Around the AADL architecture language, the authors of [14] propose a prototyping methodology and a tool suite that covers the design, analysis and deployment of models dedicated to distributed real-time embedded systems. Their results on the code generation level could be beneficially used in complement to our framework.

Perspectives. Mobile and embedded systems are highly dynamic and subject to recovery and many other constraints (resource, autonomy, time constraints for interactions). In this context it is desirable to (re)deploy efficiently software components. A perspective of this work is to extend the proposed method specifically for highly dynamic environment with the related constraints for their interactions.

References

[1] P. André, G. Ardourel, C. Attiogbé, and A. Lanoix, "Using Assertions to Enhance the Correctness of Kmelia Components and their Assemblies," *Electr. Notes Theor. Comput. Sci.*, vol. 263, pp. 5–30, 2010.

[2] ——, "Using Event-B to Verify the Kmelia Components and Their Assemblies," in *ASM*, ser. Lecture Notes in Computer Science, M. Frappier, U. Glässer, S. Khurshid, R. Laleau, and S. Reeves, Eds., vol. 5977. Springer, 2010, p. 410.

[3] C. Attiogbé, P. André, and G. Ardourel, "Checking Component Composability," in *5th International Symposium on Software Composition*, ser. LNCS, vol. 4089, 2006, pp. 18–33.

[4] E. Bruneton, T. Coupaye, M. Leclercq, V. Quéma, and J.-B. Stefani, "The fractal component model and its support in java," *Softw., Pract. Exper.*, vol. 36, no. 11-12, pp. 1257–1284, 2006.

[5] J.-R. Abrial, *Modeling in Event- B: System and Software Engineering.* Cambridge University Press, 2010.

[6] T. S. Hoang, H. Kuruma, D. A. Basin, and J.-R. Abrial, "Developing Topology Discovery in Event-B," *Sci. Comput. Program.*, vol. 74, no. 11-12, pp. 879–899, 2009.

[7] M. Belguidoum and F. Dagnat, "Dependency Management in Software Component Deployment," *Electr. Notes Theor. Comput. Sci.*, vol. 182, pp. 17–32, 2007.

[13] The Syndex Tool, www.syndex.org/index.htm

[8] D. Hoareau and Y. Mahéo, "Constraint-Based Deployment of Distributed Components in a Dynamic Network," in *ARCS*, ser. Lecture Notes in Computer Science, W. Grass, B. Sick, and K. Waldschmidt, Eds., vol. 3894. Springer, 2006, pp. 450–464.

[9] C. Tibermacine, D. Hoareau, and R. Kadri, "Enforcing Architecture and Deployment Constraints of Distributed Component-Based Software," in *FASE*, ser. Lecture Notes in Computer Science, M. B. Dwyer and A. Lopes, Eds., vol. 4422. Springer, 2007, pp. 140–154.

[10] W. Otte, A. S. Gokhale, and D. C. Schmidt, "Predictable Deployment in Component-based Enterprise Distributed Real-time and Embedded Systems," in *CBSE*, I. Crnkovic, J. A. Stafford, A. Bertolino, and K. M. L. Cooper, Eds. ACM, 2011, pp. 21–30.

[11] S. Voss, J. Eder, and F. Hölzl, "Design Space Exploration and its Visualization in AUTOFOCUS3." in *Software Engineering (Workshops)*, 2014, pp. 57–66.

[12] Y. D. Liu and S. F. Smith, "A formal framework for component deployment," in *OOPSLA*, P. L. Tarr and W. R. Cook, Eds. ACM, 2006, pp. 325–344.

[13] J. Carlson, J. Feljan, J. Mäki-Turja, and M. Sjödin, "Deployment Modelling and Synthesis in a Component Model for Distributed Embedded Systems," in *EUROMICRO-SEAA*. IEEE, 2010, pp. 74–82.

[14] J. Hugues, B. Zalila, L. Pautet, and F. Kordon, "From the prototype to the final embedded system using the Ocarina AADL tool suite," *ACM Trans. Embedded Comput. Syst.*, vol. 7, no. 4, 2008.