Event-B Formalization of a Variability-Aware Component Model Patterns Framework

Jean-Paul Bodeveix, Arnaud Dieumegard, M Filali

▶ **To cite this version:**

Jean-Paul Bodeveix, Arnaud Dieumegard, M Filali. Event-B Formalization of a Variability-Aware Component Model Patterns Framework. 15th International Conference on Formal Aspects of Component Software (FACS 2018), Oct 2018, Pohang, South Korea. pp.54-74, 10.1007/978-3-030-02146-7_3 . hal-02181895

HAL Id: hal-02181895
https://hal.science/hal-02181895

Submitted on 12 Jul 2019

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.

Open Archive Toulouse Archive Ouverte

OATAO is an open access repository that collects the work of Toulouse researchers and makes it freely available over the web where possible

This is an author's version published in:
http://oatao.univ-toulouse.fr/22579

Official URL

DOI : https://doi.org/10.1007/978-3-030-02146-7_3

To cite this version: Bodeveix, Jean-Paul and Dieumegard, Arnaud and Filali, Mamoun *Event-B Formalization of a Variability-Aware Component Model Patterns Framework.* (2018) In: 15th International Conference on Formal Aspects of Component Software (FACS 2018), 10 October 2018 - 12 October 2018 (Pohang, Korea, Republic Of).

Any correspondence concerning this service should be sent to the repository administrator: tech-oatao@listes-diff.inp-toulouse.fr

Event-B Formalization of a Variability-Aware Component Model Patterns Framework

Jean-Paul Bodeveix[1(✉)], Arnaud Dieumegard[2,3], and Mamoun Filali[4]

[1] IRIT-UPS, 118 Route de Narbonne, 31062 Toulouse, France
Jean-Paul.Bodeveix@irit.fr
[2] IRT Saint Exupéry, 3 Rue Tarfaya, 31400 Toulouse, France
Anaud.Dieumegard@irit.fr
[3] ONERA, 2 Avenue Edouard Belin, 31055 Toulouse, France
[4] IRIT-CNRS, 118 Route de Narbonne, 31062 Toulouse, France
Mamoun.Filali@irit.fr

Abstract. In the domain of model driven engineering, patterns have emerged as an ubiquitous structuring mechanism. Actually, patterns are used for instance at the requirement analysis level, during system design, and during the deployment and code generation phases. In this paper, we are interested in making precise the use of such a notion during system design. More precisely, our ultimate goal is to provide a semantic framework to support correct by construction architectures, i.e., the structural correctness of the architectures obtained through the application of patterns. For this purpose, we propose an Event-B modeling scheme for hierarchical component models. This model is built incrementally through horizontal refinements which introduce components, ports and lastly connectors. Patterns with variability are defined, instantiated and applied to user models. We show that these operations preserve the structural properties of the component model.

Keywords: Design patterns · Formal refinement · Variability
System engineering · Critical systems

1 Introduction

In the domain of model driven engineering, patterns have emerged as an ubiquitous structuring mechanism. Patterns are used for instance to express and structure requirements and to ease their analysis [8]. During the process of system development, during deployment, or for code generation activities, they ensure knowledge capitalization, and production homogeneity. Patterns may take various form and are specifically structured: textual patterns expressed as sentences, structural patterns for describing components combinations, code patterns for

This work was done while working on the MOISE project at IRT Saint Exupery.

https://doi.org/10.1007/978-3-030-02146-7_3

code generation ... We focus here on the development of system architectures that shall be refined until they can be used for equipment level refinement. The work we present here results from exchanges with safety system engineers who practice patterns for solving identified safety issues. We specifically focus here on automatically instantiated safety refinement patterns. The patterns we rely on are based on the ones detailed in [22].

We are1 interested in making precise the use of such a notion in the system development process. More precisely, our ultimate goal is to provide a semantic framework to support correct by construction architectures, i.e., the structural correctness of the architectures obtained through the application of patterns. For this purpose, we propose to use Event-B as the support platform to define the notion of hierarchical component. This Event-B model is built incrementally through horizontal refinements which introduce components, ports and lastly connectors. Patterns with variability are then defined, instantiated so that variable elements are fixed. Lastly we define pattern application on user models. We show that this transformation preserves the structural properties of the component model. We remark that Event-B is mainly used to assess the correctness of the patterns that engineers apply to given architectures.

We remark that achieving the production of structurally correct component models may be reached using different approaches. The *translation validation* approach [23] consists of verifying each individual translation whereas *transformation verification* [7] consists of verifying once and for all the generator itself. In this paper, we adopt the transformation verification approach. The specification of the transformation and its verification are done incrementally through successive refinements as supported by the Event-B method [2].

Section 2 motivates our proposal by means of a small case study and presents our pattern model. In Sect. 3, we describe how patterns are applied to component models. Section 4 introduces our formal modeling framework based on Event-B. Section 5 details within this formal setting the steps followed to apply a pattern to a model. Section 6 discusses some related works. Section 7 concludes and suggests some future works.

2 Motivating Example

We describe here an example that will be used first to clarify what we mean by a pattern and by pattern application. The next paragraph features a very simple component model. We then showcase the transformation of this model to replicate one of its components. Focus will be on the N-Version programming pattern as described in [22]. This pattern is proposed in the context of architecture safety in order to enhance system robustness. We detail its structure and content, the solution that is provided by the application of such a pattern, and its complexity from the scope of its variability.

2.1 Component Models

In our work, we consider hierarchical components models as an abstraction of the classical "boxes and arrows" modeling formalism used to model: systems as for example Capella [27], SysML [21] or AADL [14]; software as for example BIP [6], UML [25], Scade [4]; or hardware as for example VHDL [16]. In each of these formalisms, components are connected through arrows (and sometimes ports or interfaces).

We provide in Fig. 1 an example of a very simple component model. This model features the `Sub1` component with 2 input ports and 1 output port. Each port of the component is connected to another component through links. In addition to the structural description of our simple model, we have attached to the `Sub1` component one comment (yellow box) representing the association of a component with its specification, its verification or its validation artifacts or any other information of interest related to the component.

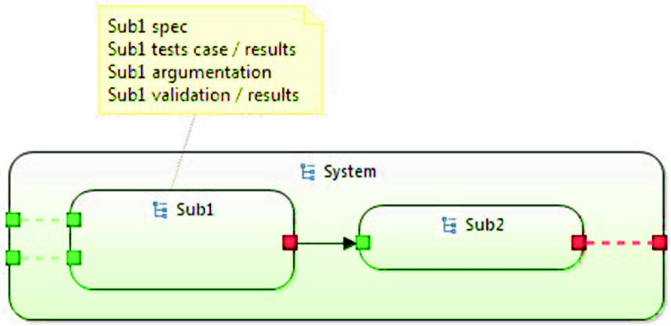


Fig. 1. A simple component model

2.2 Replicating a Component

In the use case depicted in this paper, we propose to take the example that some analysis of the system leads to the need to provide different implementations of the `Sub1` component in order to make the system fault tolerant. This mechanism is in this context referred to as replication of a component.

An example of our simple system where the `Sub1` component has been replicated is provided in Fig. 2. In this new version of the model, we have three versions (`Sub1_X`) ($X \in \{1, 2, 3\}$) whose inputs are taken from the original inputs of the `Sub1` component and dispatched using specific duplication (`dupl_Y`) ($Y \in \{1, 2\}$) components whose purpose are to replicate their inputs on each one of their outputs (the specification of these components is provided in the `dupl spec` comments). Then, the outputs of the replicated `Sub1_X` components are connected to a new component (`vote_1`) in charge of taking the decision of which one of the `Sub1_X` component output shall be relied on and sent to the outputs of the original `Sub1` component.

One may remark that applying such a replication not only preserves the original structure of the model (the interface of the `Sub1` component is the same), but also duplicates elements of the original model (the new `Sub1_X` components) and introduces new elements such as replication specifications (`Diverse implementation of Sub1_1, Sub1_2, Sub1_3`, `Vote spec`, and `dupl spec`).

2.3 Pattern Model

The previously depicted model modifications are considered in our setting as an example of the application of a design pattern. The model of Fig. 1 is the source model, and the model of Fig. 2 is the target or destination model where the pattern have been applied. What remains to be defined is what is the model of the pattern itself.

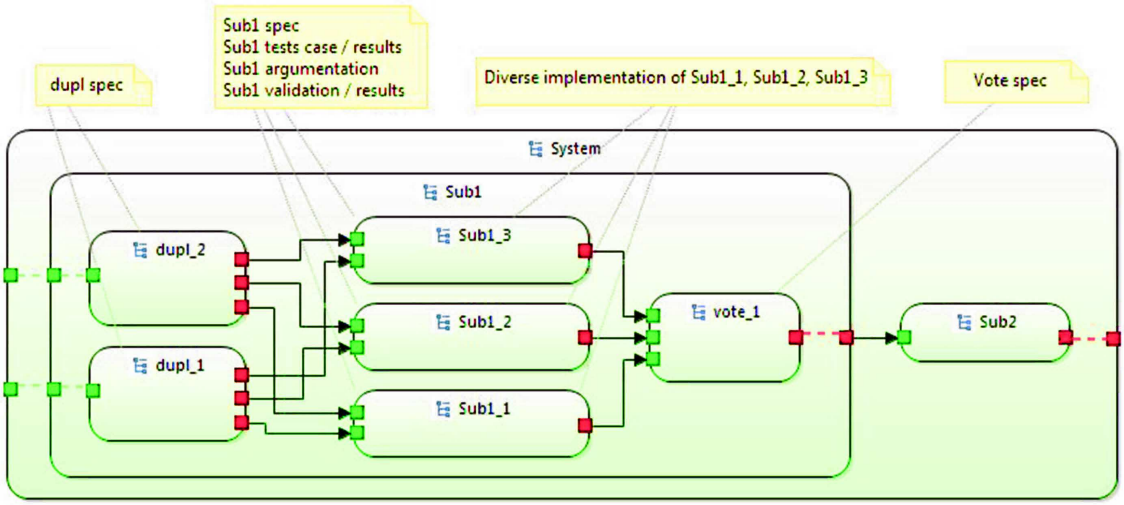


Fig. 2. A component model where a component is replicated twice

Many design pattern description formats have been defined in the literature. We decided to rely on the classical pattern description format proposed by Coplien [9]. This comprises many information among which are the name of the pattern, its context of use and the problem solved by the pattern, the strength and weaknesses of the pattern, a graphical model representation and many other information. We also rely on the work of Preschern et al. [22] where a set of architectural safety patterns are proposed for high level architecture definition. These patterns are connected to IEC 61508 standard methods for achieving safety and are extended with formal argumentation models. A simple example of such a pattern is provided in Fig. 3.

While this representation of a pattern is very interesting, it is nevertheless restrictive on the structural description of the pattern. It may be interesting to explicitly express the parameters of the pattern and its variability. In the context of the `N-Version Programming` pattern, N is a parameter meaning the number of times the software is developed. This parameter also impacts the implementation of the voting algorithm (`Voter` block). In addition to these, the links between the blocks described in the `Solution` section of Fig. 3 are a simplification of the actual possible links between blocks as there may be multiple links between these blocks: the N version of the block all have the same number of input values and output values. Both of these numbers are also parameters

Pattern Name	N-Version Programming Pattern	Pattern Type	software, fail-over
Also Known As	-		
Context	A safety-critical software without a fail-safe state which probably contains software faults.		
Problem	How to design a system which continues operating even in the presence of software faults.		
Forces	- software often contains faults - high safety certification levels require handling of systematic faults - the safety standard requires high fault coverage for single-point of failure components		
Solution	N software versions are developed independently from the same initial specification. The outputs of these versions are sent to the *Voter* which determines the best output.		

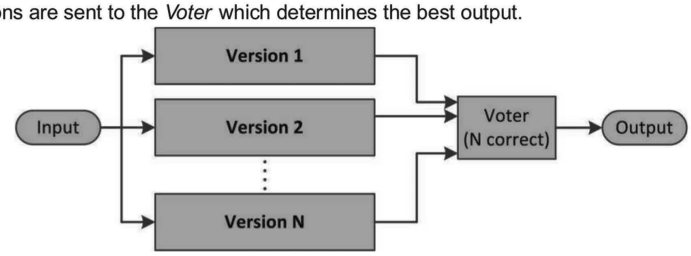


Fig. 3. N-Version Programming pattern extract from [22]

of the pattern that shall be made explicit. We thus propose an extension of this model representation of patterns where these parameters are made explicit.

Figure 4 is our proposal for an alternative graphical representation of the N-Version programming pattern model. In this model, we rely on structural elements like components, ports, and links between components through ports, and `multiplicity` objects attached to components and ports. `multiplicity` objects are represented as small grey boxes in the figure. In this pattern model, three different multiplicity elements are defined: `nb_comp`, `nb_in`, and `nb_out`. They respectively stand for the number of times the component is replicated, and the number of input and output ports of the replicated component.

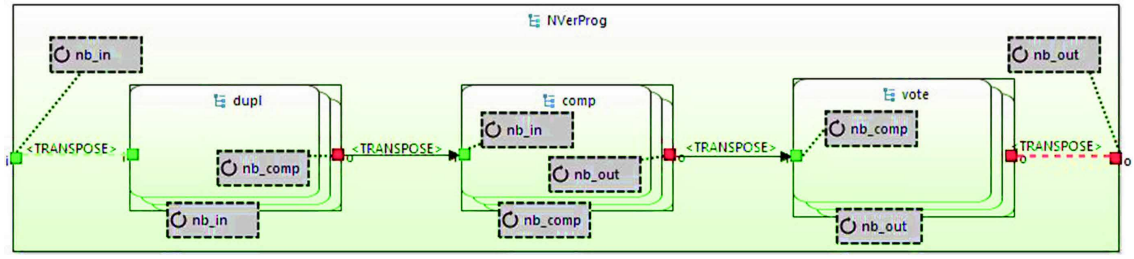
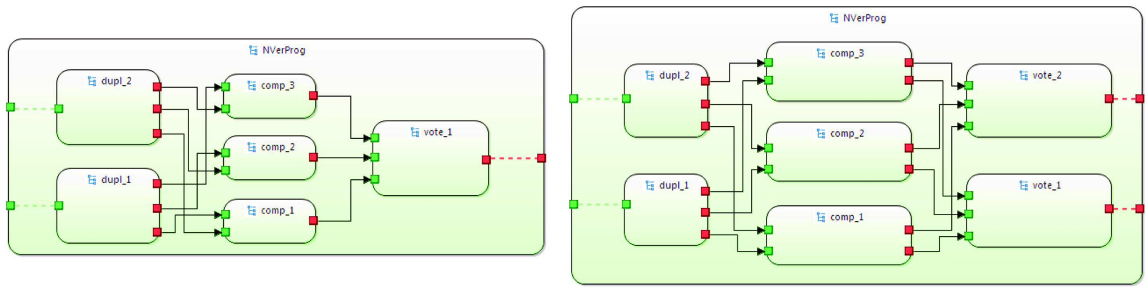


Fig. 4. The N-Version programming pattern model

By selecting the model element to be replicated (`Sub1` in Fig. 1), the last two `multiplicity` objects values are set (respectively to 2 and 1). The user shall then provide the value for `nb_comp` which is set to 3 in this case. Based on the user selection and provided values, the pattern model is instantiated: (1) its root component will have two input ports and one output port; (2) the root component of the pattern will be renamed as `Sub1`; (3) the `comp` component will be replicated `nb_comp` times as copies of the `Sub1` component with `nb_in` input ports and `nb_out` output ports; (4) the `dupl` component will be instantiated

`nb_in` times with one input port and `nb_comp` output ports; (5) the `vote` component will be instantiated `nb_out` times with `nb_comp` input ports and one output port; (6) links between components are elaborated depending on their connection pattern (detailed in the following section); Here, we use the *Transpose* pattern to connect port i of component j to port j of component i; (7) finally, the original `Sub1` model element is replaced with the newly produced `Sub1` component and its content. Figure 5 shows two instances of our `N-Version-Programming` pattern, the first one with (`nb_in`=2 `nb_out`=1 `nb_comp`=3) and the second one with (`nb_in`=2 `nb_out`=2 `nb_comp`=3).



(a) (b)

Fig. 5. (a) shows a first instance of the pattern. (b) shows a second instance.

In our setting, a pattern model is thus a family (in the product line [18] terminology) of models. Each combination of multiplicities allows for the definition of a *pattern instance*. The production of design pattern instances and the application of the produced pattern instance on a model shall thus be implemented.

The object of the Event-B model proposed in this paper is first to formally define the previously presented structure of a pattern, and second to propose a formal definition of the pattern instantiation and application algorithms. We have also produced a formalization of the structure of component models and pattern models as Ecore[1] metamodels that is a de-facto standard formalism for the specification of graph grammars[2]. This second formalization is used in order to easily produce tools for the creation, edition and display of model instances used throughout this paper. We do not detail these elements here.

3 Pattern Application

Our starting point is a parameterized pattern of which parameters are the multiplicities attached to pattern elements (components and ports). This pattern is to be applied to a model. We distinguish three steps for pattern application: initialization, elaboration, and application of patterns as depicted in Fig. 6.

[1] https://www.eclipse.org/modeling/emf/.
[2] Unlike abstract syntax which usually describe trees.

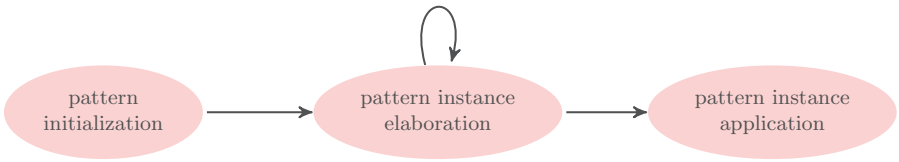


Fig. 6. Pattern application process

3.1 Pattern Initialization

During the initialization step, some of the pattern parameters are set and the root component of the pattern is identified. Patterns are parameterized by the multiplicity of their components and ports. These parameters must be fixed in order to create the pattern instance that will be applied to the model.

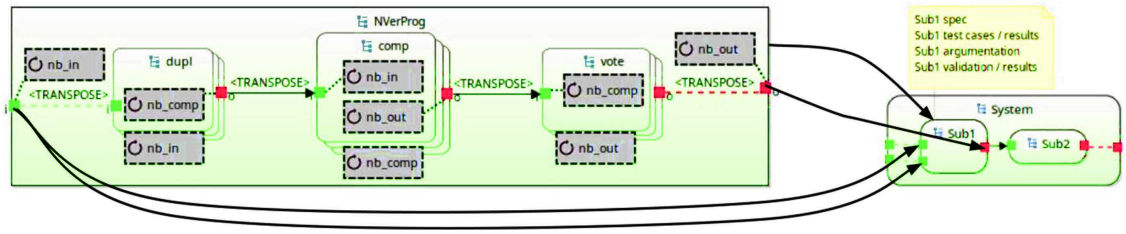


Fig. 7. Pattern (Fig. 1) to model (Fig. 4) mapping

Parameters of the pattern interface are defined through the mapping (Fig. 7) of pattern elements to model elements. This mapping constrains the multiplicity of some elements (`nb_in` = 2, `nb_out` = 1 in the figure). The `nb_comp` multiplicity should be set by the user.

3.2 Pattern Instance Elaboration

The goal of the second step is to "elaborate" the pattern. This elaboration leads to a pattern instance where multiplicities have been suppressed and which can be directly applied to a given model. The elaboration of a pattern is a complex operation since a pattern can be considered as recursive along two dimensions: horizontally due to the multiplicities and vertically due to the nesting of components. This leads to the fact that the number of instances of a sub-component is the product of the multiplicities of all its ancestors including itself. When a component is replicated its contained ports are also replicated.

Links between ports are unfolded depending on their semantics which specify how the multiple instances of the source and destination port in the pattern should be connected. It is illustrated in Fig. 8 where components s and t have respective multiplicities m_s and m_t and are linked through ports of respective multiplicities m_p and m_q. We note that these link semantics are called *connection patterns* in AADL [14]. Also, frameworks like BIP [6] and

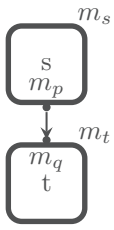
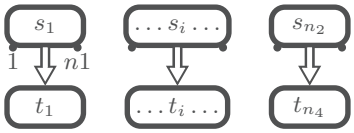
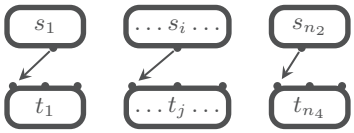
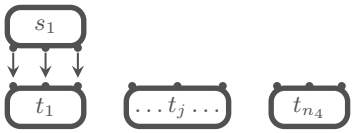
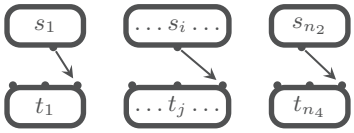
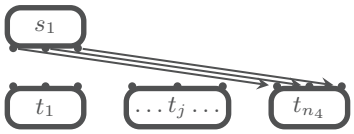
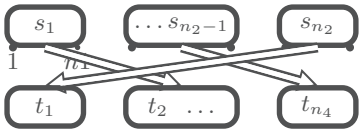
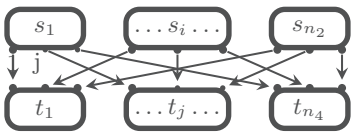


Fig. 8. Pattern link

Reo [5] provide expressive ways to define connectors between given components. However, our work is more concerned by the *application* of a pattern to an initial design.

Table 1 gives for each connection pattern the constraints on its elements multiplicities, and its textual and graphical mapping schemes.

Table 1. Connection patterns

Name	Multiplicities constraints	Source to target mapping scheme	Graphical mapping scheme
One_To_One	$m_p = m_q \wedge m_s = m_t$	$s_{i,j} \rightarrow t_{i,j}$	
First	$(m_p = 1 \vee m_q = 1) \wedge$ $m_s = m_t$	$s_{i,1} \rightarrow t_{i,1}$ $(m_p = 1$ case$)$	
	$m_p = m_q \wedge$ $(m_s = 1 \vee m_t = 1)$	$s_{1,i} \rightarrow t_{1,i}$ $(m_s = 1$ case$)$	
Last	$(m_p = 1 \vee m_q = 1) \wedge$ $m_s = m_t$	$s_{i,1} \rightarrow t_{i,n}$ $(m_p = 1$ case$)$	
	$m_p = m_q \wedge$ $(m_s = 1 \vee m_t = 1)$	$s_{1,i} \rightarrow t_{n,i}$ $(m_s = 1$ case$)$	
Rotate	$m_p = m_q \wedge m_s = m_t$	$s_{i,j} \rightarrow t_{(i+1)\%n_2,j}$	
Transpose	$m_p = m_t \wedge m_s = m_q$	$s_{i,j} \rightarrow t_{j,i}$	

This table can be extended to support additional connection patterns. For example, variants of `Rotate` could be parameterized by the number of shifts, shifting could be applied to ports or components or both...

3.3 Pattern Instance Application

In the final step, the unfolded pattern instance can be applied to the model. Applying a pattern instance comes to merging instance model elements into the user model while keeping mapped elements identical. In category theory, this operation can be seen as a pushout where mapped elements are identified.

4 Formal Framework and Component Model

Our formal framework is modeled in Event-B which supports powerful data modelling capacities inherited from set theory and offers events as the unique control structure to define data evolution. After an overview of Event-B, we describe our methodology and our component model in an incremental way.

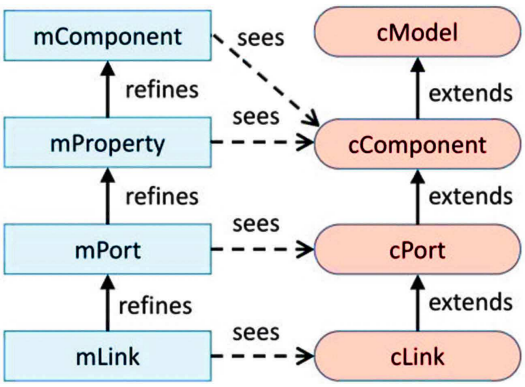


Fig. 9. Event-B model structure

4.1 A Brief Overview of Event-B

The Event-B method allows the development of correct by construction systems and software [2]. It supports a formal development process based on a refinement mechanism with mathematical proofs. We take as example the framework we have developed to illustrate the structure of an Event-B project (Fig. 9). In this figure, boxes represent Event-B machines, rounded boxes contexts, and arrows relations between these elements. Static data models are introduced incrementally through a chain of context extensions (here, with `cModel` as root). Dynamic data updated by events are introduced in machines (here, with `mComponent` as root) and subsequently refined. Each machine can access context data through the `sees` link. Contexts define abstract data types through sets, constants and axioms while machines define symbolic labelled transition systems. The state of a transition system is defined as the value of machine variables. Labelled transitions are defined by events specifying the new value of variables while preserving invariants. Moreover, the **theorem** clause expresses facts that should be satisfied. Proof obligations for wellformedness, invariant preservation and theorems are automatically generated by the Rodin tool [26]. They can be discharged thanks to automatic proof engines (CVC4, Z3 ...) or through assisted proofs.

Notations. For the most part, Event B uses standard set theory and its usual set notation. Some notations are specific to Event B:

- **pair construction:** pairs are constructed using the maplet operator $\mapsto$. A pair is thus denoted $a \mapsto b$ instead of (a, b). The set of pairs $a \mapsto b$ where $a \in A$ and $b \in B$ is denoted $A \times B$.
- A subset of $A \times B$ is a *relation*. The set of relations from A to B is denoted $A \leftrightarrow B = \mathcal{P}(A \times B)$. A relation $r \in A \leftrightarrow B$ has a domain: $\mathbf{dom}(r)$ and a codomain: $\mathbf{ran}(r)$. When a relation r relates an element of $\mathbf{dom}(r)$ with at most one element, it is called a function. The set of partial functions from A to B is denoted $A \nrightarrow B$, the set of total functions is denoted $A \rightarrow B$. The image of a set A by a relation r is denoted $r[A]$.
- The relation composition of two relations $r_1 \in A \leftrightarrow B$ and $r_2 \in B \leftrightarrow C$ is denoted as $r_1; r_2$.
- The **direct product** $r_1 \otimes r_2$ of relations $r_1 \in A \leftrightarrow B_1$ and $r_2 \in A \leftrightarrow B_2$ is the relation containing the pairs $(x \mapsto (y_1 \mapsto y_2))$ where $x \mapsto y_1 \in r_1$ and $x \mapsto y_2 \in r_2$.
- **domain restriction:** $D \lhd r = \{x \mapsto y \mid (x \mapsto y) \in r \wedge x \in D\}$
- **range restriction:** $r \rhd D = \{x \mapsto y \mid (x \mapsto y) \in r \wedge y \in D\}$
- **overwrite:** $f \lhd\!\!+ g = ((\mathbf{dom}(f) \backslash \mathbf{dom}(g)) \lhd f) \cup g$. For instance, such a notation is used to denote a new array obtained by changing the element of an array a at index i: $a \lhd\!\!+ \{i \mapsto e'\}$.

As already said, Event-B machines specify symbolic transitions through events. An event has three optional parts: parameters (**any** p1 ... pn), guards (**where** ...) specifying constraints to be satisfied by parameters and state variables, and actions (**then** ...) specifying state variables updates. Guards are defined in set-based predicate logic.

4.2 Methodology

The aim of our work is to provide a formal semantics to the application of patterns with multiplicities. This formal semantics is obtained through horizontal refinements [2]. We first elaborate an initial machine dealing with our basic components. Then, through refinements, we introduce new machines dealing successively with components having properties, ports and links. The last machine can be considered as the specification of a pattern application and the starting point for code generation through vertical refinements [1]. In the following, we first give a global overview of the considered development, then, we detail the Event-B machines underlying this development.

4.3 Incremental Description of the Component Model

Since our focus is on system engineering, our basic entity is a model denoted by the set `Model`. Each model has its *own* components which belong to the set `Component`. `Patterns` are introduced as a subset of `Model`. Except for these base sets, modeling elements are introduced through machine variables as we intend to build and update models. These modeling elements will be introduced incrementally using a horizontal refinement-based approach. At first, we introduce components in the machine `mComponent`. A model is related to a finite set of components. Each component belongs to at most one model. Furthermore, components associated to patterns have a multiplicity which will be used to parameterize the elaboration of pattern instances (c.f. Listing 1.1[3]).

We adopt a hierarchical *component* model. We formalize the hierarchy property over the components of each model. The partial function `container` returns the parent of a component, if any. In order to be well defined, containment should be acyclic. To ensure this property, we assume the existence of an irreflexive superset of the transitive closure of the `container` function: it is represented by the existentially quantified relation `f` [12]. Note that Event-B does not provide a transitive closure operator and even if it was available, using a superset is sufficient and leads to simpler proof obligations (c.f. Listing 1.2).

Listing 1.1. Models and components

```
@comp components ∈ Model ↔ Component
@comp_finite ∀m·finite(components[{m}])
@comp_not_shared components⁻¹ Component ⇸ Model
@c_mult c_multiplicity ∈ components[Pattern] → ℕ
```

Listing 1.2. Hierarchy of components

```
@cont_ty container ∈ ran(components) ⇸ ran(components)
@cont_ctr components;container;components⁻¹ ⊆ id
@acycl ∃f· f ∈ Component ↔ Component ∧ container ⊆ f ∧ f;f ⊆ f ∧ id ∩ f = ∅
```

In order to support modular descriptions, a component defines a set of input or output *ports* (c.f. Listings 1.3 and 1.4). A base set `Port`, partitioned into input and output ports (`IPort` and `OPort`) is introduced in an extension `cPort` of the context `cComponent`.

Listing 1.3. Ports context

```
context cPort extends cComponent
sets Port
constants IPort OPort
axioms
  @part partition (Port , IPort , OPort)
end
```

Listing 1.4. Port invariants

```
@port_ty ports ∈ ran(components) ↔ Port
@port_finite ∀c·finite(ports[{c}])
@port_not_shared ports⁻¹ ∈ Port ⇸ Component
@p_mult p_multiplicity ∈ (components;ports)[Pattern] → ℕ
```

[3] In Event-B, proposition labels are introduced by the @ symbol.

Modeling elements related to ports are declared in a refinement, named `mPort`, of the root machine. A port belongs to at most one component. Ports of pattern components have a multiplicity. A base set `Link` is added in the context `cLink`. Our component model is refined (machine `mLink`) to add *links* between pairs of ports. A link is also defined through its source and destination ports. For this purpose, we introduce the `src` and `dst` functions (Listing 1.5).

Listing 1.5. Links

```
@link_ty   links ∈ ran(components) ↔ Link
@link_finite  ∀c· finite( links [{ c }])
@src_ty  src ∈ ran(links) → ran(ports)
@dst_ty  dst ∈ ran(links) → ran(ports)
```

The direction of links must be compatible with the one of its source and destination ports. A link can connect a component port and a sub-component port or two sub-component ports, which leads to four cases (graphically pictured in Fig. 10 and formalized in Listing 1.6). For example, if an input port is connected to an output port (case (1)), these ports belong to the same component. Thus, the source and destination ports, supposed to be an input and an output, are ports of the component to which the link is attached. In the same way, case (2) can be read as follows: if a link of a given component connects an input to an input, its source is a port of this component and its destination is a port of a direct sub-component.

Listing 1.6. Connection constraints

```
@link_cio   links ;  ((src ⊗ dst) ▷ (IPort × OPort)) ⊆ ports ⊗ ports
@link_cii   links ;  ((src ⊗ dst) ▷ (IPort × IPort)) ⊆ ports ⊗ (container⁻¹; ports)
@link_coi   links ;  ((src ⊗ dst) ▷ (OPort × IPort)) ⊆ (container⁻¹; ports) ⊗ (container⁻¹; ports)
@link_coo   links ;  ((src ⊗ dst) ▷ (OPort × OPort)) ⊆ (container⁻¹; ports) ⊗ ports
```

We comment these constraints by expanding the formula labelled `link_cio`[4]

$$\forall\ c\ p_1\ p_2.$$
$$(\exists l.c \mapsto l \in \text{links} \wedge \overbrace{l \mapsto p_1 \in \text{src}}^{p_1=\text{src(l)}} \wedge \overbrace{l \mapsto p_2 \in \text{dst}}^{p_2=\text{dst(l)}} \wedge p_1 \in \text{IPort} \wedge p_2 \in \text{OPort})$$
$$\Rightarrow c \mapsto (p_1 \mapsto p_2) \in \text{ports} \otimes \text{ports}$$
$$\forall\ c\ l.\ c \mapsto l \in \text{links} \wedge \text{src}(l) \in \text{IPort} \wedge \text{dst}(l) \in \text{OPort}$$
$$\Rightarrow c \mapsto \text{src}(l) \in \text{ports} \wedge c \mapsto \text{dst}(l) \in \text{ports}$$

which can be read as the source and destination ports belong to the same `c` component.

The presence of links between components and ports with multiplicities impose constraints on these multiplicities. Multiplicities are attached to ports and components of the subset `Pattern` of `Model`. Pattern links must be coherent with these multiplicities and depend on the nature of the link.

[4] The equations over braces are deduced from the functionality of `src` and `dst` (Listing 1.5).

We only consider here (Listing 1.7) *Transpose* links which should connect instance port number i of instance component number j to instance port number j of instance component number i where i and j are in the range of pattern port and component multiplicities. In order to make unfolding possible, the multiplicity of the source port should be equal to the multiplicity of the target component, and conversely.

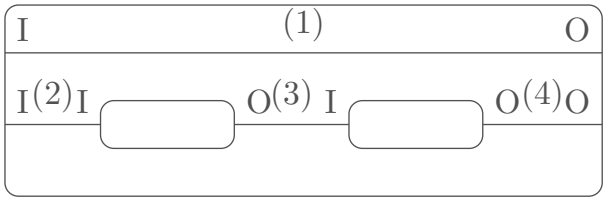


Fig. 10. Component (I/O) ports links

Listing 1.7. Multiplicity constraints

```
@tsrc ∀l·l∈ (components;links)[{Pat}]∩Transpose ⇒ p_multiplicity (src(l))= c_multiplicity (ports⁻¹(dst(l)))
@tdst ∀l·l∈ (components;links)[{Pat}]∩Transpose ⇒ p_multiplicity (dst(l))= c_multiplicity (ports⁻¹(src(l)))
```

Properties (e.g. requirements, tests, constraints...) may be associated to components, ports and links. We thus have defined the `Property` set, the elements of which are attached to components through the `cProperties` relation. We have only considered here properties attached to components.

The invariant properties we have introduced apply either to specific models (patterns when multiplicities are concerned) or to any model. The events we will present now let patterns unchanged, but create instances and update user models. They should thus establish or preserve these wellformedness properties. Three identifiers are introduced to designate these models: `Pat` for a pattern, `Inst` for a pattern instance and `Mdl` for a user model. These identifiers are declared as constants but they designate models defined through the variables introduced by the successive refinements, which allow them to evolve.

5 Pattern Application in Event-B

We study the application of domain specific design patterns to produce refinements of architecture models and ensure that the produced model including the instantiated pattern is a structurally correct refinement. We present the successive steps, illustrated by Fig. 11, needed to perform pattern application in an iterative way. In this figure, loops express the repeated firing of events during the top down traversal of the pattern structure.

5.1 Pattern Initialization Step

The `initialize_pattern` event instantiates the parameters of the pattern and identifies the root components of the pattern. This event is enriched in each refinement:

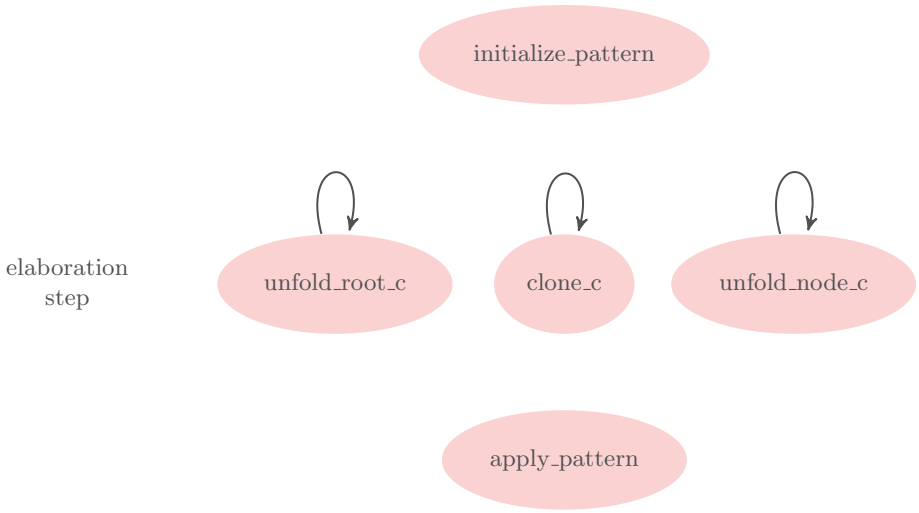


Fig. 11. Pattern application steps

Component and ports level Component (resp. ports) mapping between the pattern and the source model are provided. They allow for the extraction of components (resp. ports) multiplicities based on the number of source model elements mapped to the considered pattern element. Additional explicit pattern components (resp. ports) multiplicities are also provided by the user.

Link level. Finally at the link level, link mappings are provided and multiplicity constraints are checked depending on the link semantics (c.f. multiplicity constraints provided in Table 1).

5.2 Instance Elaboration Step

This step is initiated by the `unfold_root_c` which marks root components to be unfolded. Then, the events `unfold_node_c` and `clone_c` express the elaboration of a pattern along these two dimensions. Actually, these two events operate in a mutually recursive way. Auxiliary events and state variables are introduced to make the replication process iterative. These events are enriched in each refinement:

Component level

Root components unfolding. The instantiation event sets the variable `to_unfold_c` with the set of pattern components without containers. The `unfold_root_c` event takes one such component `c`, creates the associated instance components. The number of the created component is the multiplicity of `c`. The event stores the couples (instance, `c`) in the function `to_clone_c` used as a temporary variable to fire the next step. Links between instance and pattern components are stored in `i2p_c` (Fig. 12).

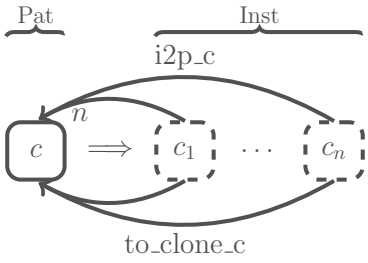


Fig. 12. Unfolding roots

Sub-component identification. This step is fired by the presence of a component `c` in the domain of `to_clone_c`. It adds to the relation `to_unfold_c_in` couples (sub-component, `c`) to prepare the unfolding (using multiplicities n_1 and n_2) of each sub-component of the image of `c` into `c` (Fig. 13).

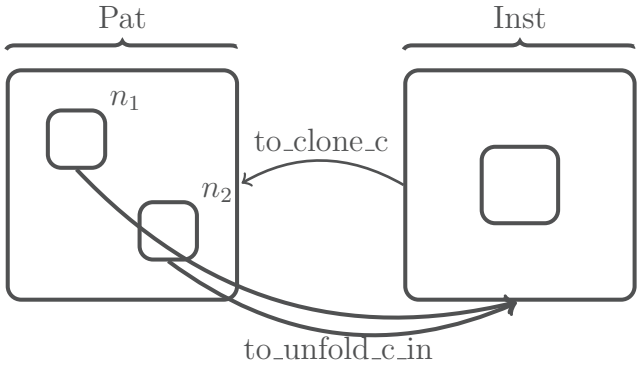


Fig. 13. Sub-component identification

Sub-component unfolding. This step (Fig. 14) is fired by a couple (`c, dest`) in `to_unfold_c_in`. It creates as sub-components of `dest` new instances $c_1, \ldots, c_n$ of `c`, the number of which corresponding to its multiplicity. The new sub-components are mapped to `c` in `to_clone_c` to pursue the unfolding process.

Port level. The sub-component identification step is enriched by storing in `to_unfold_p_in` ports to be unfolded with their destination component. A new event is added to unfold ports. It is fired by the presence of a pattern port in the relation `to_unfold_p_in`. Ports are created with the same direction as the pattern port and linked to the instance component. The variable `i2p_p` is used to store mappings between instance and pattern ports.

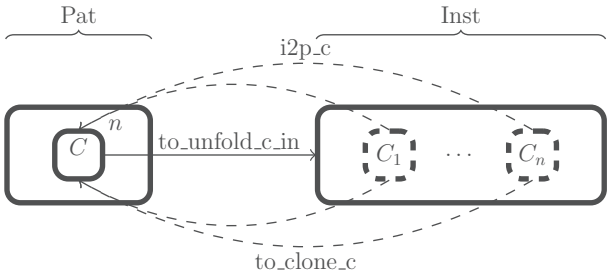


Fig. 14. Sub-components unfolding

Link level. We fire link creation within a given instance component c after its sub-components unfolding. For this purpose, four injective mappings, indexed by multiplicities, are declared from pattern source and destination components and ports to sub-components of c and their input or output ports. As a consequence, we only consider here links between two sub-components, not redirection links between a component and a sub-component, or cross-links from an input to an output of a component. In the Event-B model, we only consider *transpose* links. The corresponding multiplicity constraint is added and the array of links is created (Listing 1.8). We can see the four mappings (sci, spi, dci, dpi) to source (s) then destinations (d) components (c) and ports (p) instances (i) and the newly created links (`new_l`).

Listing 1.8. Link Unfolding

```
@links links := links ∪ ({c} × ran(new_l))
@nsrc src := src ∪ {ip,ic· ip ↦ ic ∈ dom(new_l) | new_l(ip↦ic) ↦ spi(sci(ic))(ip)}
@ndst dst := dst ∪ {ip,ic· ip ↦ ic ∈ dom(new_l) | new_l(ip↦ic) ↦ dpi(dci(ip))(ic)}
```

Properties. When elements (components, ports or links) are duplicated, their associated properties are also duplicated. This is done in the `mProperty` machine where the unfolding events are refined. An example of such a refinement for the `unfold_root_c` event is provided in Listing 1.9[5]. A similar refinement is applied for the replication of properties for ports and links in the respective events.

Listing 1.9. Properties replication for components

```
event unfold_root_c extends unfold_root_c
  then
    @prop cProperties := cProperties ∪ (ran(new_c)×cProperties[{c}])
end
```

Instantiation properties. Properties of pattern instantiation are stated as invariants. We have already expressed that the instance model (as well as any model) is well structured. We have added additional properties stating that pattern and instance models seen as labelled graphs are bisimilar with respect to the component-to-component relation `container`, the component-to-port relation `ports`, the link-to-port relations `src` and `dst` and specified the semantics of transpose links:

```
@inst2pat_cont inst2pat_c;container = container;inst2pat_c
@inst2pat_comp inst2pat_p;ports⁻¹ = ports⁻¹;inst2pat_c
@inst2pat_l_src inst2pat_l;src = src;inst2pat_p
@inst2pat_l_dst inst2pat_l;dst = dst;inst2pat_p
@transp_correct1 ∀l·l ∈ (components;links)[{Inst}] ∩ Transpose ⇒
   p_index(src(l)) = c_index(ports⁻¹(dst(l)))
@transp_correct2 ∀l·l ∈ (components;links)[{Inst}] ∩  Transpose ⇒
   p_index(dst(l)) = c_index(ports⁻¹(src(l)))
```

5.3 Instantiated Pattern Application Step

Pattern application is specified by the event `apply_pattern` initially defined for component-only models and then incrementally specified to support ports and links. This event applies the pattern instance obtained through the preceding step to the user-supplied model. This event is enriched in each refinement:

Component level. Pattern instance application (Listing 1.10) is fired by providing a mapping `inst_components` from instance components to model components.

[5] In Event-B, event action labels are introduced by the @ symbol.

Listing 1.10. Instance application at Component level

```
event apply_pattern // transformation du mod\'{e}le
  any inst_components // instance mapping
      new_components
  where
    @ic inst_components ∈ components[{Inst}] ⤖ components[{Mdl}]
    @nc new_components ∈ components[{Inst}] \ dom(inst_components) ⤖ Component \ ran(components)
    @acycl_inst_components dom(inst_components) ◁ container;inst_components ⊆ inst_components;container
    @acycl_container container[dom(inst_components)] ⊆ dom(inst_components)
  then
    @m components := components ∪ ({Mdl}×ran(new_components))
    @f container := container ∪ ((inst_components ∪ new_components)^-1;container;
                    (inst_components ∪ new_components))
end
```

Listing 1.11. Updated superset of the model containment relation

```
new_components^-1; f; inst_components; f0 ∪
(new_components^-1; f; (new_components ∪ inst_components)) ∪ f0
```

Unmapped components (not belonging to the interface), designated by the `new_components` identifier, will be created and inserted to the set of components of the model. The container function of the model is updated to take into account containment coming from the pattern instance (Fig. 15).

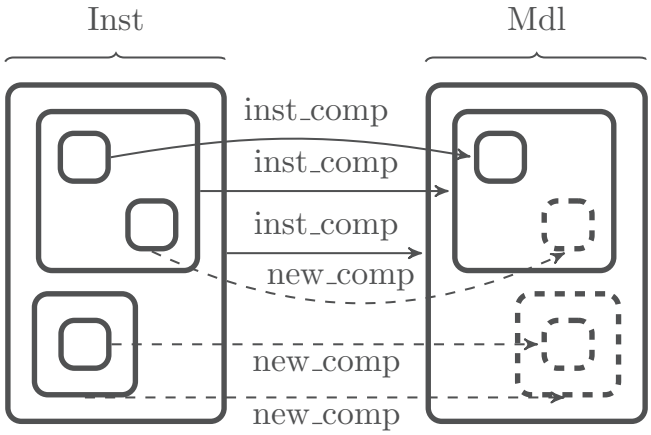


Fig. 15. Pattern application

The main point is to show that invariant properties are preserved, one of them being the acyclicity of the containment relationship. Some hypotheses (`@acycl_inst` and `@acycl_container`) are needed to avoid merging a graph and its inverse: if an instance component is mapped to a model component and has a container, this container should be mapped to the container of the model component. The acyclicity proof is then quite automatic once the superset of the transitive closure of the new container function has been provided. Given supersets `f` and `f0` of the pattern instance (resp. user model) containment function supposed to be closed for composition. The relation of Listing 1.11 contains the updated (after pattern application) model containment relation, it is closed for composition and is irreflexive. The added hypotheses ensure that the pattern instance is inserted as a subtree of the user model. Thus the two containment relations need not be interleaved.

Listing 1.12. Instance application at Property level

```
event apply_pattern extends apply_pattern
  any inst_props
  where
    @inst_prop  inst_props  ∈ Property ⇸ Property // pattern properties to model properties
  then
    @prop cProperties  := cProperties ∪
      ((inst_components ∪ new_components)⁻¹;cProperties; (id ⩤ inst_props))
```

Properties. Properties attached to pattern components are transferred to their corresponding model components (Listing 1.12). However, if pattern properties are instanciated by model properties, these ones are used instead.
Port and link levels. Instance pattern application is extended to ports using the same code schema as for components. Container update is replaced by port-to-component update. Furthermore port mapping and new ports should preserve port direction. In the same way, links are considered and link to port attachments are made consistent.

6 Related Works

A refinement pattern is a transformation rule that constructs a model refinement. The generation of correct-by-construction B/Event-B refinements has already been studied. They either propose a dedicated language for the expression of patterns ([24] for B, [17] for Event-B), or a pattern is seen as a usual Event-B machine that is mapped on the Event-B machine to be refined [15].

However, rather than focusing on patterns applied on Event-B models, our objective is the formalization using Event-B of the instantiation and the application of patterns for system architectures expressed using component models. Let us remark that a pioneering work advocating a formal approach, especially for architectural design patterns, is [3,11]. Behavioral semantics of the patterns is considered thanks to TLA: the Temporal Logic of Actions [19] and the behavioral correctness of the composition with respect to safety and fairness properties is proven. To the best of our knowledge, this work has not been mechanized.

We have chosen Event-B as a meta-level framework and used it to express a semantics for components models usually adopted by Model Based System Engineering frameworks [13,28]. Using this framework, we have defined a semantics for the definition, the instantiation and the application of patterns. As in [13], patterns are defined by adding multiplicities to target models and a pattern application algorithm is proposed. However, we consider component models, not argumentation models and our formalization is incremental (horizontal refinement) and its dynamics has been formalized through Event-B events. Thus, pattern elaboration and application are not monolithic algorithms and can easily be extended through refinement. As a consequence, correctness proofs can also be of finer grain.

As we said in the introduction, patterns are used in many stages of the development process. Temporal patterns have been proposed by [10] to promote the use of temporal logics for behavioural specifications. Also, in a context closer to ours, with respect to the underlying component model, [20] consider dynamic properties of patterns. However, their approach is based on model checking and consequently follows a translation validation approach whereas we follow a transformation verification approach. It should be interesting to investigate how such dynamic properties could be combined with the static properties presented in this paper and evaluate well suited verification approaches.

7 Conclusion

As said in the introduction, the work presented here results from exchanges with safety system engineers. Safety concerns lead to applying some design patterns selected among those solving the identified safety issues. In order to make the pattern library reusable, we provide a limited form of variability management through pattern model element multiplicities. We have presented an Event-B specification of two main operations needed to support the process: pattern instantiation taking into account variability and pattern instance application to the user model. These operations are modeled in an incremental way based on horizontal refinements and are shown to preserve basic structural properties of the component model.

Additional work may also be done in order to prove relevant properties on the pattern instantiation and application algorithms especially regarding the correctness of the application of the pattern. Such correctness shall be defined properly in terms of preservation of replicated model elements properties. Extensions of the pattern instantiation/application mechanisms may allow the mapping of sets of components/ports/links to a single pattern model element. This leads to a more powerful instantiation mechanism allowing in our example to replicate the chain of components used as input of the replicated component. As said in the introduction, we have used Event-B mainly to assess the correctness of pattern application. We believe that this "correct by construction" approach is interesting for the elaboration of frameworks dedicated to, e.g. safety, engineers.

References

1. Abrial, J.-R.: The B-book: Assigning Programs to Meanings. Cambridge University Press, New York (1996)
2. Abrial, J.-R.: Modeling in Event-B: System and Software Engineering, 1st edn. Cambridge University Press, New York (2010)
3. Alencar, P.S.C., Cowan, D.D., Lucena, C.J.P.: A formal approach to architectural design patterns. In: Gaudel, M.-C., Woodcock, J. (eds.) FME 1996. LNCS, vol. 1051, pp. 576–594. Springer, Heidelberg (1996). https://doi.org/10.1007/3-540-60973-3 108

4. Abdulla, P.A., Deneux, J., Stålmarck, G., Ågren, H., Åkerlund, O.: Designing safe, reliable systems using scade. In: Margaria, T., Steffen, B. (eds.) ISoLA 2004. LNCS, vol. 4313, pp. 115–129. Springer, Heidelberg (2006). https://doi.org/10.1007/11925040_8
5. Arbab, F.: Reo: a channel-based coordination model for component composition. Math. Struct. Comput. Sci. **14**(3), 329–366 (2004)
6. Basu, A., Bensalem, S., Bozga, M., Combaz, J., Jaber, M., Nguyen, T.-H., Sifakis, J.: Rigorous component-based system design using the BIP framework. IEEE Softw. **28**(3), 41–48 (2011)
7. Blazy, S., Leroy, X.: Mechanized semantics for the clight subset of the C language. J. Autom. Reason. **43**(3), 263–288 (2009)
8. Carson, R.S.: Implementing structured requirements to improve requirements quality. In: INCOSE International Symposium, vol. 25, pp. 54–67. Wiley Online Library (2015)
9. Coplien, J.O.: Software Patterns. SIGS Management Briefings. SIGS books & multimedia, New York (1996)
10. Dwyer, M.B., Avrunin, G.S., Corbett, J.C.: Patterns in property specifications for finite-state verification. In: Proceedings of the ICSE' 99, Los Angeles, CA, USA, May 16–22, pp. 411–420 (1999)
11. Dong, J., Alencar, P.S.C., Cowan, D.D., Yang, S.: Composing pattern-based components and verifying correctness. J. Syst. Softw. **80**(11), 1755–1769 (2007)
12. Damchoom, K., Butler, M., Abrial, J.-R.: Modelling and proof of a tree-structured file system in event-b and rodin. In: Liu, S., Maibaum, T., Araki, K. (eds.) ICFEM 2008. LNCS, vol. 5256, pp. 25–44. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-88194-0_5
13. Denney, E., Pai, G., Whiteside, I.: Model-driven development of safety architectures. In: 20th ACM/IEEE International Conference on Model Driven Engineering Languages and Systems, MODELS 2017, Austin, TX, USA, September 17–22, 2017, pp. 156–166. IEEE Computer Society (2017)
14. Feiler, P.H., Gluch, D.P.: Model-Based Engineering with AADL - An Introduction to the SAE Architecture Analysis and Design Language. SEI Series in Software Engineering. Addison-Wesley, Upper Saddle River (2012)
15. Hoang, T.S., Fürst, A., Abrial, J.-R.: Event-B patterns and their tool support. Softw. Syst. Model. **12**(2), 229–244 (2013)
16. Heinkel, U., Glauert, W., Wahl, M.: The VHDL Reference: A Practical Guide to Computer-Aided Integrated Circuit Design (Including VHDL-AMS) with Other. Wiley, New York (2000)
17. Iliasov, A., Troubitsyna, E., Laibinis, L., Romanovsky, A.: Patterns for refine-ment automation. In: de Boer, F.S., Bonsangue, M.M., Hallerstede, S., Leuschel, M. (eds.) FMCO 2009. LNCS, vol. 6286, pp. 70–88. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-17071-3_4
18. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, A.: Feature-oriented domain analysis (foda) feasibility study. Technical Report CMU/SEI-90-TR-021, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA (1990)
19. Lamport, L.: Specifying Systems: The TLA+ Language and Tools for Hardware and Software Engineers. Addison-Wesley, Boston (2002)
20. Marmsoler, D., Degenhardt, S.: Verifying patterns of dynamic architectures using model checking. In: Kofron, J., Tumova, J. (eds.) Proceedings International Workshop on Formal Engineering approaches to Software Components and Architectures, FESCA@ETAPS 2017, Uppsala, Sweden, 22nd April 2017, vol. 245 of EPTCS, pp. 16–30 (2017)

21. OMG. OMG Systems Modeling Language (OMG SysML), Version 1.3 (2012)
22. Preschern, C., Kajtazovic, N., Kreiner, C.: Building a safety architecture pattern system. In: Proceedings of the 18th European Conference on Pattern Languages of Program, EuroPLoP '13, pp. 17:1–17:55, ACM, New York, NY, USA (2015)
23. Pnueli, A., Siegel, M., Singerman, E.: Translation validation. In: Steffen, B. (ed.) TACAS 1998. LNCS, vol. 1384, pp. 151–166. Springer, Heidelberg (1998). https://doi.org/10.1007/BFb0054170
24. Requet, A.: BART: a tool for automatic refinement. In: Börger, E., Butler, M., Bowen, J.P., Boca, P. (eds.) ABZ 2008. LNCS, vol. 5238, pp. 345–345. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-87603-8_33
25. Rumbaugh, J., Jacobson, I., Booch, G.: Unified Modeling Language Reference Manual, 2nd edn. Pearson Higher Education (2004)
26. http://www.event-b.org/
27. Sango, M., Vallée, F., Vié, A.-C., Voirin, J.-L., Leroux, X., Normand, V.: MBSE and MBSA with Capella and safety architect tools. In: Fanmuy, G., Goubault, E., Krob, D., Stephan, F. (eds.) CSDM 2016, p. 239. Springer, Cham (2016). https://doi.org/10.1007/978-3-319-49103-5_22
28. Voirin, J.L.: Model-based System and Architecture Engineering with the Arcadia Method. Elsevier Science (2017)