A model transformation approach for multiscale modeling of software architectures applied to smart cities

Ilhem Khlif, Mohamed Hadj Kacem, Cédric Eichler, Khalil Drira, Ahmed Hadj Kacem

A model transformation approach for multi-scale modeling of software architectures applied to smart cities

Ilhem Khlif*[1] | Mohamed Hadj Kacem[1] | Cédric Eichler[2] | Khalil Drira[3] | Ahmed Hadj Kacem[1]

[1]University of Sfax, ReDCAD Research Laboratory, Sfax, Tunisia
[2]LIFO, INSA Centre Val de Loire, Bourges, France
[3]LAAS-CNRS, Univ. de Toulouse, Toulouse, France

Abstract

Modeling and specifying correct software systems is a challenging task that can be supported by providing appropriate modeling abstractions. This paper proposes an approach for graphical multi-scale modeling of such systems using model transformation techniques. The approach is founded on a guided rule-based iterative modeling process ensuring controlled transition from a coarse-grained description to a fine-grained description. It provides also user-friendly graphical descriptions by extension of UML notations, hence preserving the common practices from software architectures design. The iterative design process is supported by a set of model transformation rules. The rules manage the refinement process (by adding or removing sub-systems or by adding or removing details on a given sub-system) as a model transformation. Our approach is supported by a rule-based generator that implements the automatic transformation of UML diagrams into Event-B specifications allowing formal verification of their correctness properties, and relieving software architects of mastering formal techniques. To experiment and validate our approach, we consider a case study dedicated to the smart cities.

KEYWORDS:
Software Architecture; multi-scale description; UML models; formal specifications; Event-B method; model transformation; refinement.

1 INTRODUCTION

Complexity of software systems is increasing and their architecture is becoming more distributed. Such complex systems can be described in a hierarchical way as an interconnection of subsystems. The complexity of these systems make their design arduous, in particular when formal verification must be conducted. Software architecture description can characterize the system design at a high level and constitutes the focus of our contribution. Software architecture design and description is a challenging task especially with the continuous growth in the size and complexity of software systems. On the one hand, we have to describe the system with enough details to allow its understanding without ambiguity and its implementation in conformance with architects' requirements and users' expectations. On the other hand, we have to control the complexity induced by the increasing model details both at the human and automated processing levels. Some high level properties can be expressed on abstract descriptions with a high level of abstraction and checked on simple formal descriptions. Some other properties need more detailed descriptions to be expressed and detailed specifications to be elaborated. Description details may be system-independent and mainly structure-centric, such as component decomposition, or system-specific and mainly behavior-based, such as message ordering in communication protocols.

There is a need for a new approach reconciling understandability and complexity that automates the architectural design step and guarantees its correctness. Such an approach shall rely on multiple architectural meta-models and support an iterative modeling process that helps architects to elaborate complex yet tractable and appropriate architectural models. Different properties of correctness have to be maintained between the models and the specifications at the different iterations. Each intermediate iteration provides a description with a given abstraction that allows the validation to be conducted significantly while remaining tractable w.r.t. complexity. The iterative process involves both system-independent

structural features ensuring the model correctness, and system-specific features related to the expected behavior of the modeled system. For this purpose, we propose to consider different architecture descriptions with different levels of modeling details, called **"scales"**. We define a step-wise iterative process starting from a coarse-grained description and leading to a fine-grained description. The proposed multi-scale approach relies on a two dimensional refinement process including both vertical and horizontal transformations. Vertical refinements add decomposition details to specify the internal structure of previously defined components. Horizontal refinements add details on the interconnections between components. We start with modeling the first scale by a given coarse grain description using a UML component diagram. This diagram is refined through model transformation operations. We execute successive refinements until reaching a fine-grain description representing the necessary details. We rely on a modeling solution to describe software architectures using visual notations by extending the UML graphic language. Such notations make it possible to describe the structural properties as well as the behavioral properties of the software architectures. Accordingly, we propose a formal description of both scales and refinement rules to enable correctness verification. We translate automatically the UML-based architectural models into Event-B specifications that we execute to check the correctness of both structural and behavioral properties of software architectures.

Model transformation is the process of converting one model to another, both implementing the same system. Many interrelated models are organized along levels of abstraction in a complex software architecture, with mappings defined from one model to another. When the target model is more expressive than the source model, the transformation has to ensure that all the properties expressed in the first model are all preserved into the target models. We implement a rule-based generator that automatically translates UML-based architectural models into Event-B specifications. We validate our approach using the Rodin theorem prover tool supporting Event-B, and check the syntax and the correctness of the generated specifications [1]. We apply our approach to the smart cities case study. We experiment and evaluate the performance of our approach as well as the functional aspect of our developed tool supporting the multi-scale approach. The remainder of the paper is organized as follows. The necessary background knowledge about the main concepts used in the proposed approach are introduced in Section 2. It defines software architectures and multi-scale modeling. We describe the multi-scale modeling rules in Section 3. In Section 4, we detail the model transformation approach for multi-scale modeling. Section 4.3 presents the case study. In Section 5, we present a survey of related work. We conclude and outline the main perspectives in Section 6.

2 BACKGROUND

The concept of scale appears in the domain Science of materials where there is two classical points of view [2]. On the one hand, scientists are interested in the macroscopic behavior of a system where they model the effect of big scales by constitutive relations. As an Example for solids, engineers use the continuum models and represent the atomic effects of constitutive relations without acquiring much knowledge about the origins of the cohesion between the atoms of the material. On the other hand, the interest is in the microscopic mechanism of a process: it is assumed that there is nothing interesting that happens in finer scales. For example, physicists are more interested in the behavior of solids at the atomic level, they often work under the assumption that processes are homogeneous on a macroscopic scale. The multi-scale modeling is helpful to handle a problem simultaneously from different scales and different levels of detail. The challenge is to have the efficiency of macroscopic models and the accuracy of microscopic models.

Most problems in science and engineering are multi-scale in nature. The multi-scale nature of the problems related to the dimensions of global change demands that researchers address key issues of scales in their analyses.

2.1 Scale concepts

The scale is considered as generic concept that includes the spatial or temporal dimensions used to measure any phenomenon as defined by Nested-CA: a foundation for multi-scale modeling of land use and land cover change [3]. A scale is characterized by two maYesor concepts: **the grain** is the finer spatial resolution, the resolution refers to the granularity used in measurements; **the extent** refers to the size of the total study area.

In our context, the **extent** scale refers to the abstract description considering a sub-system of the System of Systems (SoS). Variation in extent can be used, for example, to describe a given description level or a given communication layer in communicating systems. It allows the architect to describe the necessary details to understand the system architecture and validate the associated software properties. Besides, the **grain** scale refers to the level of details and precision pertaining to the abstract description, providing more communication details of a given current description.

Multi-scale modeling takes advantage of data available at distinct scales by modeling the interactions among those scales, accordingly managing the complexity of phenomena involved. Practically, this can be achieved by decomposing a problem into a set of single scale models that exchange information across the scales.

2.2 Refinement concepts

The concept of architecture refinement is a key aspect to design any software system. Enabling the architecture refinement elaborates what was already present in the abstraction and leads to an appropriate level of details. The refinement ensures integrity and consistency of software architecture, reduces costs and improve software quality. However, the refinement of a software architecture is considered as a current step in the process of research because of the complexity of conversion from abstract to specific architecture [4].

We propose to introduce our approach for the multi-scale modeling and description of software-intensive systems. At the first scale, the system is described by a simple model that can be reduced to a single component. This component is successively refined to reach a finer grain description that contains all relevant details. The proposed multi-scale approach relies on a two dimensional refinement process including both vertical and horizontal transformations. Vertical refinements add architectural decomposition details by specifying the internal structure of previously defined components. Horizontal refinements add details on the interconnections between components. We start with modeling the first scale by a given coarse grain description using a UML component diagram. This diagram is refined through model transformation operations. We execute successive refinements until reaching a fine-grain description representing the necessary details.

2.3 Scale/Multi-scale in our context

We define a vertical description scale “Sv” (Grain scale) as a model that provides additional details of design (decomposition details) that pertain to “Sv-1”. Under each vertical scale there are several horizontal description scales “Sv.h” (Extent), enriched with horizontal refinements and thus providing communication description and details of a given current description. It allows the architect to describe the necessary details to understand the system architecture and validate the associated properties. A top-down scale transformation process, much like regular refinement, begins with a high level description of a system which we describe as a whole. Then, scale changes are applied to obtain a more detailed description, by describing components and connections. A bottum-up scale transformation process is much like regular abstraction, begins with a low level of design details which describe a system. Then, scale changes are applied to obtain a more abstraction.

We propose a hybrid approach. The top-down approach is presented by the refinement process which transforms architecture in both a vertical and a horizontal way. The bottom-up approach is described by the abstraction process, which consists of vertical and horizontal transformations. A multi-scale description guarantees the execution of the necessary model transformations rules. These rules manage the refinement/abstraction between scales.

3 MULTI-SCALE MODELING RULES

Our approach is based on three principal steps (1) allowing multi-scale modeling and specifying correct software architectures. This approach supports the modeling of multi-scale architectures, automatic transformation of UML diagrams to Event-B specifications and formal verification.

3.1 Our approach in a nutshell

- Step 1: Our approach supports the modeling of multi-scale architectures using a visual notation based on the UML graphic language [5]. Our approach allows to describe the structural properties as well as the behavioral properties of the multi-scale architecture. Each model is submitted to vertical and horizontal refinements. The system automatically maps UML models towards XML language. This function ensures that each generated XML descriptions is valid according to the appropriate Ecore defining the proposed meta-model. To ensure model consistency, our approach supports model validation of UML models using OCL constraints.

- Step 2: Our approach implements a rule-based generator that automatically translates UML-based architectural models (structural and behavioural properties) into Event-B specifications. The transformation is based rules and implemented through the XSLT language [6].

- Step 3: To enhance confidence level of UML models, our approach provides a formal definition of their syntax and semantics. The main contribution of this work is the translation of UML models into Event-B in order to verify functional properties of our models (such as deadlock freedom, and liveness) automatically. We use the Rodin theorem prover tool supporting Event-B and check the syntax and the correctness of the generated specifications. During changes, the system must be left in a correct and coherent state. This concern is twofold. The first aspect is related to architectural style. The system must maintain its conformity to the style and its intrinsic constraints. By an example of the Publish-Subscribe style, we illustrate our approach and we check the information dissemination property, which dictates that produced information reaches all subscribed consumers. The second aspect relates to multi-scale architecture and refinement-based approaches. Accordingly, we prove the consistency between scales through the proposed formal refinement process.

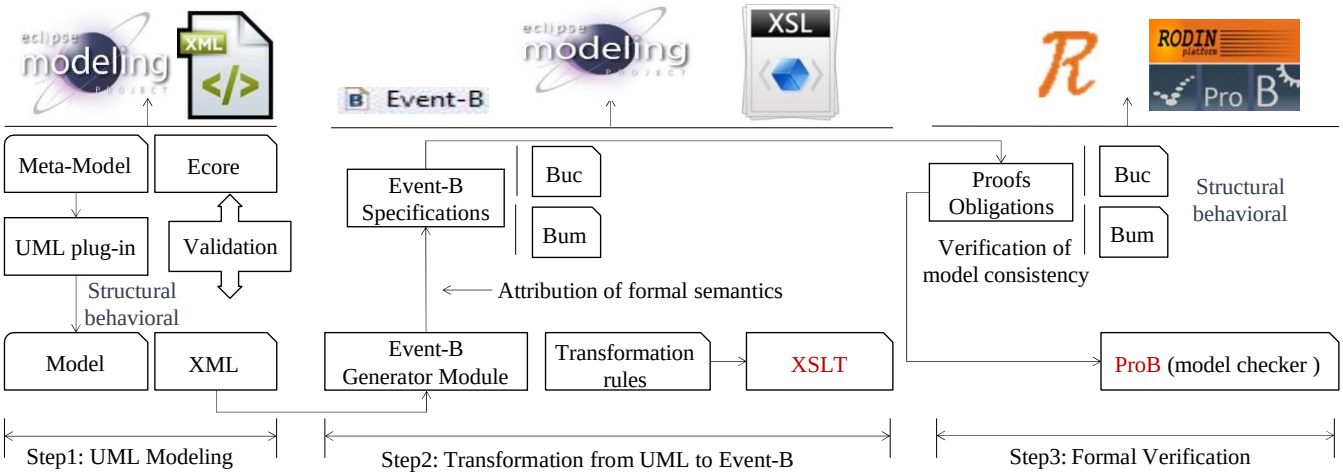

FIGURE 1 Chaining of the three steps composing the implemented approach

As depicted in (Figure 1) showing the chaining of the three steps that compose the implemented approach, the implemented design approach is founded on UML notations and uses component diagrams. The diagrams are submitted to vertical and horizontal transformations for refinement; this is done to reach a fine-grain description that contains necessary details. The model transformation ensures the correctness of UML description, and the correctness of the modeled system.

We present the multi-scale approach by a two dimensional array describing vertical and horizontal refinements. We start by modeling the first scale, which is defined by a UML component diagram. This diagram is refined, through model transformation operations, until reaching the last scale.

3.2 UML meta-models

We present the multi-scale approach by a two-dimensional array describing vertical and horizontal refinements. Vertical description scales allow the architect to describe the same inherent requirements while providing multiple descriptions having different granularity scales. Under each scale, there are several horizontal refinements. We start by modeling the first scale, which is defined by a UML component diagram. This diagram is refined, through model transformation operations, until reaching the last scale that represents the architectural style.

We present the multi-scale approach by a two-dimensional array describing vertical and horizontal scales (Figure 2). Gray classes represent the added details in each refined scale and white classes represent the conserved architectural properties. Vertical scales are the vertical description levels that allow the architect to describe the same inherent requirements while providing multiple descriptions having different granularity levels. Under each vertical description scales there are several horizontal description scales. The first scale Sv_0 begins with specifying the application requirements. It defines the whole application by its name.

Two horizontal refinements called horizontal scales are associated with the first level Sv_1. The first horizontal scale shows all components that compose the application. The second one describes the links between those components.

Four horizontal refinements are associated with the second level Sv_2. The first scale presents subcomponents for components, and enumerates all the roles that each component can take. The second one identifies the list of communication ports for each component, and refines those roles. The third one shows the list of interfaces for communication ports. The last one is obtained by successive refinements while adding the list of connections established between components and subcomponents. This scale allows us to define the architectural style.

An iterative modeling process involves both structural properties ensuring the model correctness, and specific properties related to the expected behavior of the modeled domain. We follow a top-down strategy where a model of the larger scale is built and refined with details for smaller scales until reaching all levels of details. We define a first scale architecture. In general, a scale i represents coarse grained components, such $i \in [0, n]$ where n corresponds to the depth of the hierarchy defined in the SoS model. For $i = 0$, we obtain the first scale. Then, it is refined by adding the next scale components. The obtained architecture is refined in turn until reaching the last scale, i.e., where all system components are defined. The transition between scales is implemented following a rule-oriented refinement process.

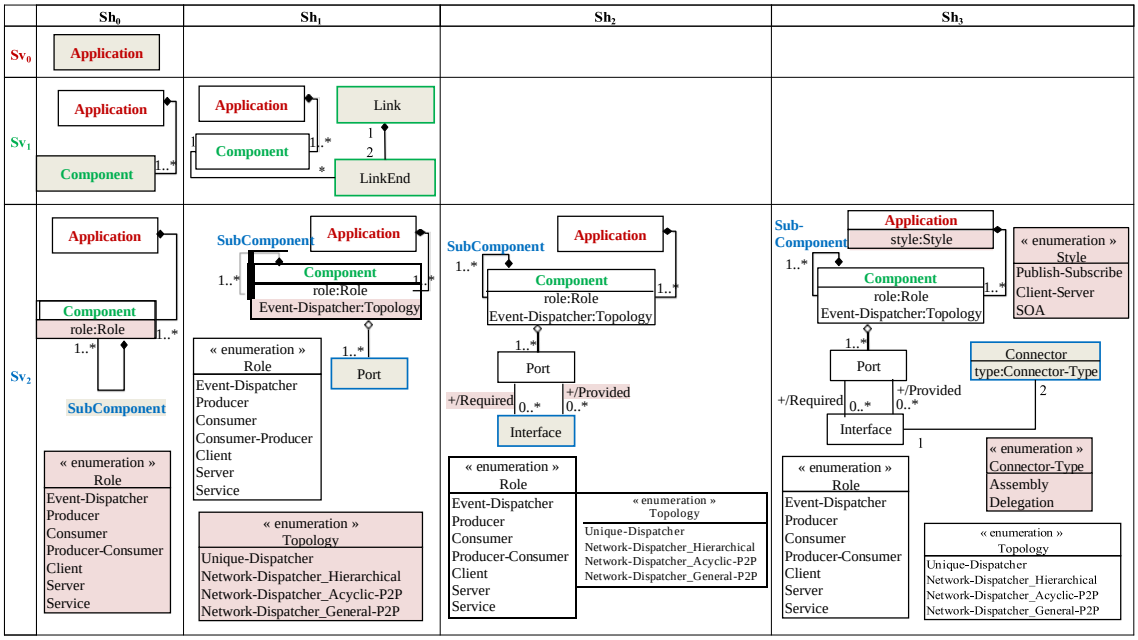

FIGURE 2 UML meta-models of a multi-scale architecture description

3.2.1 Structural properties

We elaborate an initial abstract architecture description from the user requirements. In the first iteration, application requirements are specified (a unique component C_0 is identified) This is the beginning of the traceability.

A new iteration is required for providing details on the application. Three component types named C_1, C_2, and C_3 that are interconnected are added (Figure 3).

The second iteration is helpful for checking that, at the next scale, the components identification is preserved, as we keep the traceability of a component from one scale to another. This notation is used for identifying a component C_m where m represents a cursor on the current component ($m \geq 0$). It can be decomposed in the next scale. As illustrated in ((Figure 3), the component C_1, will be refined with two composites in the next scale identified as follows $C_{1.1}$, $C_{1.2}$. The component C_2 will be refined with two sub-components named $C_{2.1}$ and $C_{2.2}$. Similarly, the component C_3 is composed of the sub-components named $C_{3.1}$ and $C_{3.2}$.

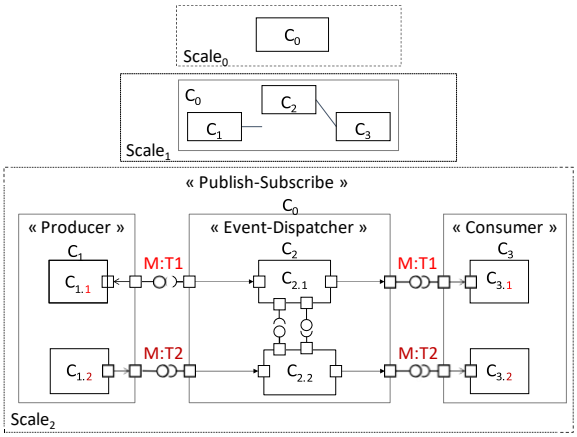

FIGURE 3 Structural modeling

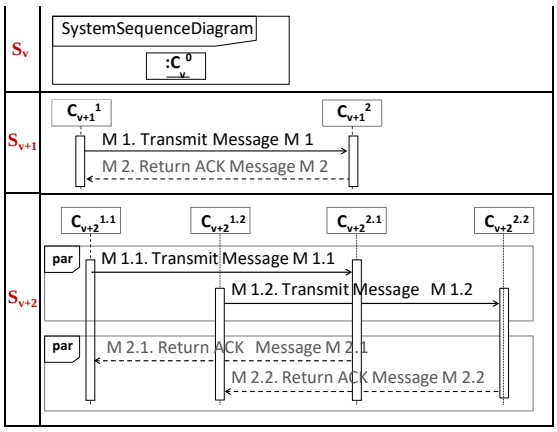

FIGURE 4 Behavioral modeling

We are especially interested in refining an enabling architectural style for component-based systems: the Publish-Subscribe style. The strength of this event-based interaction style lies in the full decoupling between producers, and consumers. This decoupling is provided by the event dispatcher. The approach may be applied in many different domains and across different architectural styles, for example Service Oriented Architecture(SOA), Client-Server, etc.

Then, in the last scale, roles are associated with components such as “Event-Dispatcher", “Producer", “Consumer", “Producer-Consumer", etc, and thus connections between them are established. A possible configuration to refine the interaction (link) between the components C_1 and C_2 is illustrated. If the component C_1 is a “Producer" and the component C_2 performs the role of an “Event-Dispatcher", the link between C_1 and C_2, in the first scale, will be decomposed into a simple assembly connection, in the next scale, extending from the source $C_{1.1}$ to the target $C_{2.1}$. As a link is divided according to its identifiers, then a trace of the link decomposition is added. Moreover, the interaction between the two components C_2 and C_3 will be refined at the next scale as follows: if the component C_3 is a “Consumer" and the component C_2 is an “Event-Dispatcher", the link between C_3 and C_2 will be decomposed into a simple assembly connection extending from the source $C_{2.1}$ to the target $C_{3.1}$. During the iterations, we have to check an intrinsic property ensuring the model correctness w.r.t. UML description (interface compatibility). We preserve the multi-scale architecture consistency stating whether the components of the architecture are correctly typed and well connected (each interface is connected to a compatible one). From a structural viewpoint, an architecture is complete if all its required interfaces (related to the consumers $C_{3.1}$, $C_{3.2}$) are connected to compatible provided ones (related to the dispatchers $C_{2.1}$ and $C_{2.2}$).

In addition, we preserve the model traceability from one scale to another by decomposing links, at the abstract scale, and refining them, at the next scale, to show possible connections established between components. Reaching a fine grain description in a software architecture promotes the description of the types of components and sub-components and the kinds of relationships that can connect them. Different properties of correctness have to be maintained between the models at the different levels of iterations. The decoupling property states that producers and consumers do not communicate directly, but their interactions have to be mediated by the event dispatcher. Then, the produced information must reach all the subscribed consumers. This is to ensure the principle of information dissemination. Let M be the produced information, represented as a message with a type T (eg. Information, Coordination, Cooperation, etc.). For all producers (eg. $C_{1.1}$) in the application, there is one consumer (eg. $C_{3.1}$) while the produced information (eg. M:T1) is transmitted by the producer ($C_{1.1}$). So, the message (M:T1) is received by one consumer $C_{3.1}$). Similarly, the producer ($C_{1.2}$) will transmit the produced information with a different type (M:T2) to another consumer ($C_{3.2}$).

We indicate all possible connections established according to the used topology and respecting the Publish-Subscribe style. The Event dispatcher C_2 is refined with two sub-components (dispatchers) named $C_{2.1}$ and $C_{2.2}$. The following details are related to the principle of information dissemination: if two dispatchers communicate together then it is necessary that the information coming from the first reaches the second. So, the communication has to be bidirectional. Producers and consumers communicate symmetrically as peers, adopting a protocol that allows a bidirectional flow of communication (acyclic-P2P topology). All dispatchers have to be interconnected (direct or transitive connection). Each producer, consumer, or both denoted must have a single access point in the network of dispatchers. This interaction is governed by a principle of information propagation requiring that produced information have to reach all subscribed consumers. To guarantee this property in the case of a distributed event dispatcher, we have to check, on the one hand, that all dispatchers are interconnected (direct or transitive connection). On the other hand, if two dispatchers communicate together then it is necessary that the information coming from the first reaches the second. So, the communication has to be bidirectional. A double assembly connection is established between the two sub-components $C_{2.1}$ and $C_{2.2}$. In the acyclic peer-to-peer topology, dispatchers communicate with each other symmetrically as peers, adopting a protocol that allows a bidirectional flow of subscriptions. For all Producer with a type T (eg. $C_{1.1}$ with M:T1), the produced information M:T1 is transmitted by this producer $C_{1.1}$, there is a consumer ($C_{3.1}$) with the typed message M:T1 while M:T1 is received by $C_{3.1}$) and this via the dispatcher $C_{2.1}$ that ensures the correct transmission of the produced information M:T1. Based on this property, the consumer receives only once the same message arriving from the producer to the dispatcher.

During the iterative design process, architectural properties have to be preserved after each iteration and hence are part of the multi-scale modeling that must be enforced during the software evolution. Architectural models are refined during the design process. Architectural properties must be maintained between higher and lower level models All refined elements are preserved in the lower level model.

3.2.2 Behavioral system-specific properties

The aim of the multi-scale modeling is to study the required behavioral properties of the considered application. The application is initialized (at the first scale), and after successive iterations, the sets of components and interactions among them are identified in a way that supports the required behavior of the abstract application level. After identifying interactions, we consider giving structured representations of components behavior as a series of sequential steps over time. We describe the specified behavior of an application using the UML sequence diagram (Figure 4).

In the first scale, the whole application is presented as a black box to illustrate the System Sequence Diagram (SSD) named “C_0". The main issue here is to secure the message transmission and how elements cooperate to ensure correct information propagation. Several events may refine an abstract event: A single message (M_1) between actors from a coarse-grained description scale is translated into a set of messages ($M_{1.1}$ and $M_{1.2}$) at the next scale, or the content of translated messages depends on earlier received message.

The sequence diagram, represented in Figure 4, specifies behavioral features of the publish-subscribe architecture. When the Producer-Consumer component named C_1 sends a message (M_1:T1) to the Event dispatcher component C_2 at the first scale, the dispatcher tracks this message and, replies by sending an acknowledgement message (M_2:T1). At the next scale, those messages will be refined into a parallel sequence of messages and keep track of the type of message sent or received in the abstract scale. For example, the typed message (M_1:T1) sent from C_1 to C_2 is refined into two messages having the same type: the message ($M_{1.1}$:T1) is sent from $C_{1.1}$ to $C_{2.1}$ and ($M_{1.2}$:T1) is sent from $C_{1.2}$ to $C_{2.2}$.

We propose to check two properties to describe the behavior of a multi-scale architecture. First of all, a behavioral scale description adds information that reveals the order of interactions among the elements opportunities for concurrency time dependencies of interactions. We have to preserve the traceability property from a vertical description scale to another. This property deals with the event deadline and shows time sequences explicitly, making it easy to see the order in which event must quickly occur. An event will be refined from a vertical scale to another. This iteration allows to preserve the event structure until reaching the fine grain description. The traceability property is ensured through both the identification and the type of exchanged messages.

We check here the concurrency property of the system in which several behaviors can overlap in time. In the sequence diagram, refined messages are executed in parallel as shown in the last scale. We model the concurrency using a combined fragment with the par operator. So, while $M_{1.1}$ must be sent before $M_{1.2}$, and $M_{2.1}$ must be received before $M_{2.2}$ must be received, the parallel operator indicates that the messages of the two operands may be interleaved. This allows each lifeline to see six possible orders of the message-send/message-arrive events. In addition, because the messages may be transmitted at different speeds, the order seen by lifelines $C_{1.1}$ and $C_{2.1}$ is independent of the order seen by lifeline $C_{1.2}$ and $C_{2.2}$.

Our approach is based on a multi-scale modeling that helps to automate the construction of correct design architectures. So, we need to specify the software architecture model that describes the software components, their composition and their interactions. In fact, each model is represented as a set of scales, and each scale denotes a set of architectures. Following our approach, the designer starts by modeling the first scale architecture which is refined to give one or many architectures for the next scale. Then, these architectures are refined in turn to give the following scale architectures and so on until reaching the last scale. The transition between scales is ensured by applying both structural and behavioral refinement rules. After constructing the architectures of software architecture model, we apply the relation between the two models in order to obtain model-based architectures with different description levels.

3.3 Structural validation

Our approach suggests a structural validation technique as shown in Figure 5.

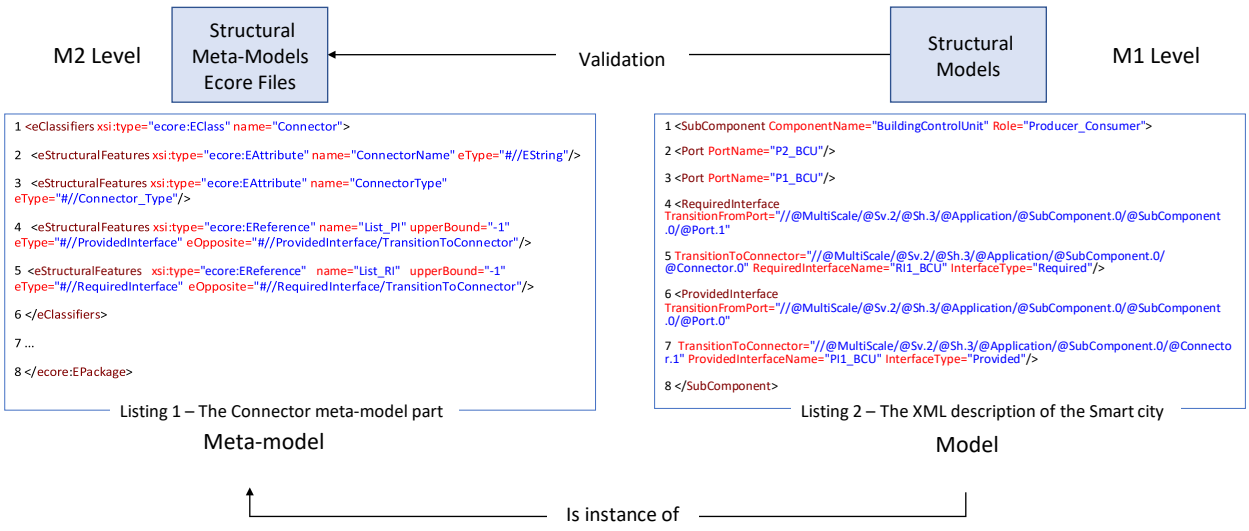

FIGURE 5 Structural Validation

The validation is used (i) to facilitate the identification of possible inconsistencies, (ii) to detect and correct specification errors and (iii) to ensure the model conformity with respect to its appropriate meta-model. The validation approach consists of two steps. First, we represent each meta-model scale by an Ecore file (Step 1, Figure 1). We define structural validation rules describing the structure and the composition of each proposed

meta-model scale. For example, a connector should establish a link between Components via two Interfaces t can be defined as Assembly or Delegation type (Scale SV2, SH2, Figure 2) as shown in the Listing 1, Figure5 . Second, we implemented in our Eclipse plug-in the translation of UML model of each scale into textual representation. We generate XML descriptions as shown in the Listing 2, Figure 5 from the defined UML notation (see Figure 10).

4 MODEL TRANSFORMATION APPROACH

The semi-formal language UML is one of the most widely used modeling language in software engineering. It provides unique meta-models to graphically describe systems, which makes the whole modeling process visual and easier to handle. However, during the modeling phase, the architect can easily fall into error. This is due to the lack of formal semantics for UML modeling languages that do not offer rigorous verification tools.

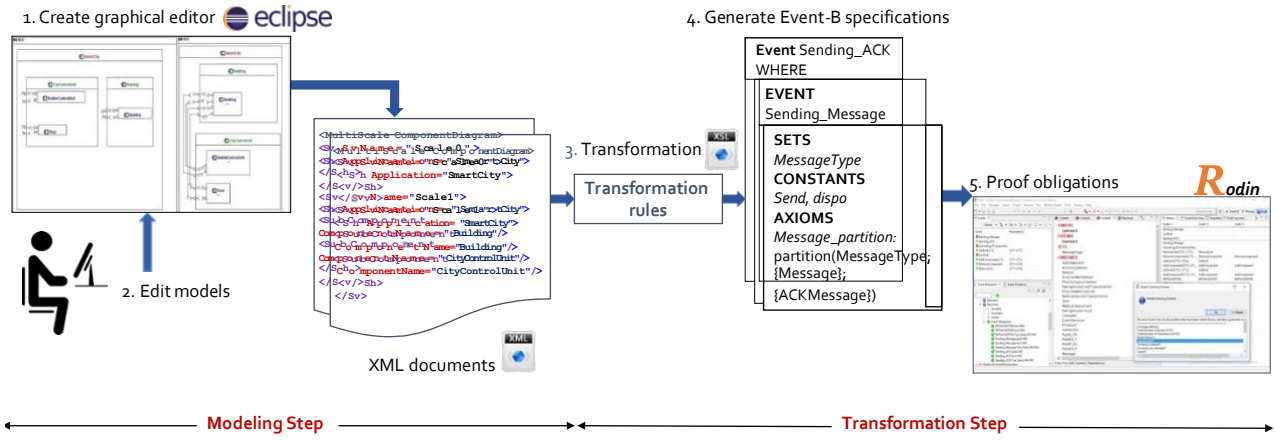

FIGURE 6 Transformation process

In order to solve this problem, we propose a two-step process following the UML modeling as depicted in Figure 1. We firstly translate UML diagrams into formal Event-B models and then formally verify the latter.

This work focuses on the construction of correct architectures, one of the most challenging tasks of software design. Accordingly, we adopt the Event-B formal method as it supports this process. In the Event-B method, an abstract scale is defined and successively refined by adding smaller scale details. In our approach, transformation rules automatically generate Event-B specifications for each scale by translating the different elements of the UML models into their corresponding concepts in the Event-B method, as illustrated in Figure 6. The generated formal specifications are then submitted to a verification step, their consistency being checked under proof obligations using the Rodin platform.

This section firstly introduces the target Event-B specifications and discusses the verification step. It then presents the transformation rules used to populate the target models. For each diagram introduced for each scale in the previous section, we present hereafter the corresponding target model and transformation rule.

4.1 Target model: scales formal specifications

This subsection introduces the target formal models corresponding to the UML models previously introduced. In event-B, a context describes the static part of a model, and a machine describes the dynamic behavior of a model. Accordingly, the component diagram that constitutes the static part of the architecture is specified as a context. The sequence diagram that constitutes the dynamic part of the architecture, it is specified as a machine.

4.1.1 Structural features: Event-B Contexts

Structural features are specified with several contexts. Each context has a name and clauses such as “Constants" to declare constants, “Sets" to declare sets (that can be viewed as data types), “Axioms" used to type constants and define predicates that must be verified. A context may extend another, inheriting all its axioms and declarations, as specified in the “Extends" clause.

Specification 1. Scale 0: global system

CONTEXT
Context0
SETS
Component
CONSTANTS
System, composition
AXIOMS
System_partition: partition(Component, {System})
Axm0: finite(Component)
Axm1: composition ∈ (Component ↔ Component) → (Component ↔ Component)
Axm2: ∀ r · r ⊆ composition(r)) ∀ r · composition(r); r ⊆ composition(r)
Axm3: ∀ r, s · r ⊆ s ∧ s; r ⊆ s ⇒ composition(r) ⊆ s
END

Specification 2. Scale 1: Components and relations

CONTEXT
Context1
EXTENDS
Context0
SETS
Association
CONSTANTS
C1 , C2 , .. , Cn , Link1 , .. , Link
AXIOMS
Component_part: partition(Component, {C1} , {C2} , .. , {Cn})
Association_part: Association ∈ Component ↔ Component
Link_part: partition(Association, {Link1} , .. , {Linkm})
END

Specification 3. Scale 2: Sub-components and message-based communication

CONTEXT
Context2
EXTENDS
Context1
SETS
MessageType
CONSTANTS
SubComponent, Consumer, EventService, Producer, Connector,PushC_ES,
PushES_C,PushP_ES, PushES_P, Message, ACKMessage, Can_Send, n
AXIOMS
axm1: partition(Component, SubComponent)
axm2: partition(SubComponent, { Consumer},{ EventService } , { Producer})
axm3: Connector ∈ SubComponent ↔ SubComponent
axm4: dom (PushC_ES) = {Consumer}
axm5: ran (PushC_ES) = {EventService} axm7: dom (PushES_C) = {EventService}
axm6: ran (PushES_C) = {Consumer}
axm7: ran (PushP_ES) = {Producer}
axm8: dom (PushES_P) = {Producer}
axm9: partition(MessageType, {Message}, {ACKMessage})
axm10: Can_Send ∈ Component ↔ MessageType
axm11: Can_Send = Producer ⤖ Message, EventService ⤖ ACKMessage
axm12: n=2
END

Scale 0: global system

At the first large scale of the modeling step, an abstract model is specified which is further refined in the next scales to add more details. We specify the first scale description using an Event-B specification named “Context0".

In the context 1, we specify a constant named “System" modeling the whole architecture at this abstract scale. In this initial model, we define the carrier set “Component" of all components and we axiomatize that it is finite (Axm0). In fact, at this scale, it is solely composed of the singleton System, as specified in the “AXIOMS" clause by the “System_partition" axiom. In the constant clause, we define a function “composition" that formalizes the transitive composition of binary relations between Components (Axm1). Then, we specify the forward relational composition and we note with “;" (Axm2). For example, if two components A and B are in relation and if components B and C are in relation, then the composition of r = {(A, B), (B, C)} is r ∪{(A, C)}. In fact, the relations between components are symmetric transitive and composition is the transitive closure of these relations (Axm2, Axm3). A component A and component B communicate (potentially indirectly) via the relation r, which implies that the pair (A, B) is necessarily included in the composition (r). By ensuring this property, we ensure the dissemination of data between these two components.

Scale 1: Components and relations

The refinement of the architecture continues until reaching the level of details necessary to verify the associated architectural properties. Each new iteration either refines components with new components or introduces connectors. A new context 2 named “Context1" extends the context 1 and specifies the type of components constituting the system. We define two new kinds of constants in the “CONSTANTS" clause: components (C_1, C_2, .. , C_n) and their connections (Link1, . . . ,Linkm). These links composed the set “Association" (“Link_part”) Links are further specified as an Event-B relation between components (“Association_part").

Scale 2: Sub-components and message-based communication

A “Context2" extends the previous “Context1", and refines the architecture with new sub-components and connections. At this scale, components are possibly composed of new sub-components named SubComponent (axm1). These components are assigned a role (axm2) among “Producer", “Consumer" and “EventService". Connectors between components are then specified with an Event-B relation between two components (axm3). Connectors named (“PushC_ES", “PushES_C", “PushP_ES", “PushES_P") are specified as constants, each connector having a name (axm4). Axioms 5 to 10 specify sources and targets of each connector type. For example, a connector named “PushC_ES" establishes a communication from a consumer (axm3) to an event service (axm4). Finally, components are formally related to the kind of messages they can send through the “Can_Send relation" (axm12, axm13). To this end, two types of messages are declared as constants as specified in (axm14) where n is the number of messages to be sent ; the initially sent message “Message" and acknowledgement message “ACKMessage". These types compose the set “MessageType" (axm11).

4.1.2 Behavioral specifications: Event-B Machines

In the first machine 4, we specify the system requirements by using the context 1. This is the beginning of traceability. In fact, each trace is analyzed for correctness to verify that all system requirements are satisfied and are essential to obtain correct models.

Scale 1-1: Link suppression and addition

In the Machine 4, we formalize the behavior of the system, where links (represented as pairs of components) may be created or removed at any time. These actions are specified by two events (“AddLink", “RemoveLink"). A direct one-way link between a pair of distinct components represents their capacity to perform direct communication. The variable “AddedLinks" represents the set of links that currently exist. The variable “RemovedLinks" represents the set of links that existed at some point, but have been deleted. These sets are disYesoint (inv03) since a link cannot have both status.

Specification 4. Scale 1-1: Link suppression and addition

```
MACHINE
    Machine0
SEES
    Context0
VARIABLES
    AddedLinks, RemovedLinks
INVARIANTS
    inv01: AddedLinks ∈ Component ↔ Component
    inv02: RemovedLinks ∈ Component ↔ Component
    inv03: AddedLinks ∩ RemovedLinks= ∅
EVENTS
    INITIALISATION
        BeginAct
            act01: AddedLinks := ∅
            act02: RemovedLinks := ∅
        EndAct
    EVT
        AddLink:
            ANY Link
            WHERE
                grd1: Link ∉ AddedLinks
            THEN
                act01: AddedLinks := AddedLinks ∪ {Link}
                act02: RemovedLinks := RemovedLinks \ {Link}
        RemoveLink:
            ANY Link
         WHERE
                grd2: {Link} ∈ AddedLinks
           THEN
                act03: AddedLinks :=
                AddedLinks \ {Link}
               act04: RemovedLinks := RemovedLinks ∪ {Link}
END
```

Specification 5. Scale 1-2: Update links information by components

```
MACHINE
    Machine1
REFINES
    Machine0
SEES
    Context0
VARIABLES
    addedlinks, removedlinks
INVARIANTS
    inv011: addedlinks ∈ Component → (Component ↔ Component)
    inv022: removedlinks ∈ Component → (Component ↔ Component)
    inv033: ∀ n.addedlinks(n) ⊆ AddedLinks ∪ RemovedLinks
    inv044: ∀ n.removedlinks(n) ⊆ AddedLinks ∪ RemovedLinks
    inv055: ∀ n.removedlinks(n) ∪ addedlinks(n)= ∅
EVENTS
    INITIALISATION
        BeginAct
            act011: addedlinks:= ∅
            act022: removedlinks:= ∅
        EndAct
    EVT
        addlink:
            ANY n, link
            WHERE
                grd11: n ∈ Component
                grd12: link ∈ AddedLinks ∪ RemovedLinks
            THEN
                act021: addedLinks(n) := addedLinks(n) ∪ link
                act023: removedlinks(n) := removedlinks(n) \ link
        removelink:
            ANY n, link
            WHERE
                grd11: n ∈ Component
                grd12: link ∈ AddedLinks ∪ RemovedLinks
            THEN
                act021: removedLinks(n) := removedLinks(n) \ link
                act023: removedlinks(n) := removedlinks(n) ∪ link
        PreserveLink:
            status: ordinary
            WHERE
                grd11: ∀ x, y.x ↣ y ∈ AddedLinks ⇔ x ↣ y ∈ addedlinks(y)
                grd12: ∀ x, y.x ↣ y ∈ RemovedLinks ⇔ x ↣ y ∈ removedlinks(y)
                grd13: ∀ n, n.m ↣ n ∈ composition (AddedLinks) ⇒
                            ∀ k.k ↣ m ∈ addedlinks(n) ⇔ k ↣ m ∈ addedlinks(m)) ∧
                             k ↣ m ∈ removedlinks(n) ⇔ k ↣ m ∈ removedlinks(m)))
END
```

Scale 1-2: Update links information by components

Machine1 is a refinement of Machine0, using Context1. This first refinement is helpful to introduce events for checking stable states and model how components update their link information. Indeed, in our meta-model, each component stores information about available links. This information can be updated when a link is established or removed. Accordingly, we introduce two variables “addedlinks" and “removedlinks" that represent the current information stored by each component about links states. The first two invariants formalize that each component stores its own local information as a binary relation between “Components". If a component has some information about a link, then this link exists or used to exist and belongs to either “AddeLinks" or “RemovedLinks" (inv033 and inv044). The last invariant (inv055) states that a component can not store contradictory information about the same link. Finally, we specify the “PreserveLink" event. This event has no effect on this system state itself as its action is “skip". Its guard is used to define the notion of a stable state of the system. The first two guards require that every component knows the correct status of all its inward links, and has detected all architectural changes with respect to its links. The last guard requires that if there is a trace from a component “m" to “n", then “n" has the same (added/removed) information as m for all links to m.

Scale 2: Message based communication

At this scale, we specify communication messages occurring in the links specified in the previous machine. This is done in “Machine2" which refines “Machine1", using “Context2" and adding communication between the sub-components. The invariants “Send" and “Available" are used to ensure that each sub-component can send a message or receive an acknowledgment only if it is authorised. The behavior is specified as follows: the producer sends a “Message" to the consumer. When the consumer becomes available, it receives the “Message", processes it and sends the “Ack-Message". When the producer becomes available, it receives the “ACK-Message". The invariants (inv1, inv2) specifies what is the sent message, who is the sender and the receiver.

Specification 6. Scale 2: Message based communications

```
MACHINE
    Machine2
REFINES
    Machine1
SEES
    Context2
VARIABLES
    Send , Available
INVARIANTS
    Send_inv: Send ¨ ∈ Connector ↔ MessageType
    Available_inv: Available ∈ SubComponent ⇸ BOOL
    Can_Send_INV:∀ z,x,y·z ∈ Component ∧ {x ↦ y} ∈ Connector ↔ MessageType∧ dom({x})= z ∧ (x ↦ y)
    ∈ Send ⇒(z ↦ y) ∈ Can_Send
EVENTS
    INITIALISATION
        BeginAct
            act1:                    Send:=                    ∅
            act2: Available:= {Producer ↦ TRUE, EventService ↦ FALSE}
        EndAct
    EVT
    Sending_Message
        WHERE
            grd1: Producer ∈ dom(Available) ∧ Available(Producer)=TRUE
            grd2: Send= ∅
        THEN
            act1: Send := {Send ∪ PushC_ES ↦ Message}
  Sending_ACK:
        WHERE
            grd1: EventService ∈ dom(Available) ∧ Available(EventService)=TRUE
            grd2:    (PushC_ES    ↦    Message)    ∈    Send
            grd3: (PushES_C ↦ ACKMessage ∈ Send
        THEN
            act1: Send ∈ Send ∪ PushES_C ↦ ACKMessage
END
```

4.1.3 Checking properties using proofs obligations

Mathematical proofs allow to verify model consistency and consistency between refinement levels. Proof obligations in Event-B are useful for checking properties. They define what is to be proved to ensure the consistency of an Event-B model and can be experimented either as predicates (“INVARIANTS", “AXIOMS",“THEOREMS") or with “GUARDS" in the events. Behavioral properties such as liveness, reachability and information dissemination are thereby checked. We formulate those properties as predicates (“INVARIANTS", “AXIOMS").

For example, the principle of information dissemination states that the produced information must reach all the subscribed consumers. We specify this property as follows: Let “MessageType" be the produced information, represented as a message with a type. For all producers in the system, there is one consumer that must receive the sent message. We formulate this property as predicates (axm11,axm12) in the context3.

Reachability and decoupling are also formalized in such a way. Reachability means that the components are able to capture all exchanged messages; i.e., a message sent is necessarily received by at least one component. The decoupling property stating that producers and consumers do not communicate directly. Their interactions have to be mediated by the event service. Invariants are supposed to hold whenever variable values change. Obviously, this does not hold a priori for any combination of events and invariants and, thus, needs to be proved. The corresponding proof obligation is called invariant preservation. We use this kind of proof obligation ensuring that each invariant is preserved by each event. In this manner, we check that each component only sends a message if it is authorised. This is controlled by the invariants “Send_inv", “Available_inv" and “Can_Send_Inv". In fact, possible state changes are described by means of the events (“Sending_Message" , “Sending_ACK") and each event maintains respectively the invariants (“Send", “Available"). For sequence diagrams, we require that every message start an activation; i.e. each invariant is checked and preserved. For example, with “Can_Send_Inv" we check:

$$\text{Can\_Send\_INV}: \forall\, z,x,y \cdot z \in \text{Component} \land \{x \mapsto y\} \in \text{Connector} \leftrightarrow \text{MessageType} \land \text{dom}(\{x\}) = z \land (x \mapsto y)$$

Each property is verified at the appropriate abstraction level. When we enrich the model by using refinement techniques, proofs automatically generated by the Rodin Platform to make sure that the refined models are not contradictory. Hence, previously checked properties are preserved during each refinement. Ultimately, the last scale is correct by design and all properties stand.

4.2 Transformation rules

This subsection introduces the rules used to translate UML concepts into Event-b, populating the target model presented previously. These rules are classified depending on whether they impact the structural or behavioral vertical properties of the multi-scale architecture.

4.2.1 Structural features

Structural features of a multi-scale architecture are generally specified by describing its components and their inter-relations. We use the UML

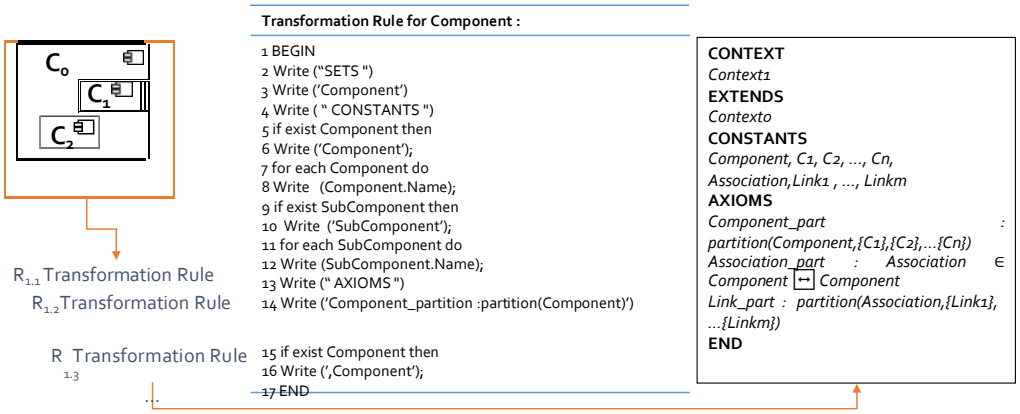

FIGURE 7 An example of structural transformation

component diagram for modeling structural properties. This diagram presents the components that make up the architecture, their types and their connections. We propose to transform a component diagram into an Event-B context. As a result of this step, we describe the static part of the Event-B model by identifying the constants and their properties. To conduct model transformations, we specify a set of transformation rules to be applied on the UML component diagrams. We associate to each component diagram concept a transformation rule translating it into its equivalent construction in Event-B, as illustrated in Figure 6. The corresponding rules related to scales $S_{1.1}$, $S_{1.2}$ and $S_{2.1}$,..,$S_{2.4}$ depicted in (Figure 7) are presented thereafter. A rule noted Ri.Yes when related to vertical i and horizontal scale Yes. Rules related to scale 0 are trivial and therefore not reported.

The following algorithm describes how components and sub-components are translated from UML into Event-B. To ease comprehension, we break it down into several sub-rules hereafter.

Algorithm 1: Transformation Rule for Component and Sub-Component:

```
1  BEGIN
2  if exist SubComponent then
3      Write("SETS ") ;
4      Write ("SubComponent");
5  end
6  Write ( " CONSTANTS ") ;
7  for each Component do
8      Write (Component.Name);
9  end
10 for each SubComponent do
11     Write (SubComponent.Name);
12 end
13 Write (" AXIOMS ") ;
14 if exist Component then
15     Write ('Component_partition : partition(Component');
16     for each Component do
17         Write (', ' + Component.Name);
18     end
19     Write (")") ;
20 end
21 if exist SubComponent then
22     Write ('SubComponent_partition : partition(SubComponent');
23     for each SubComponent do
24         Write (', ' + SubComponent.Name);
25     end
26     Write (")") ;
27 end
28 END
```

- **R. Component (Alg. 1):** This rule populates the target model (context 2) with the *component*s of the source model (i.e. elements of the component diagram describing the first scale).

Components' name are translated into constants in the "CONSTANTS" clause (lines 7 to 9 in algorithm 1). For example, two UML components named C_1 and C_2 will lead to C_1 and C_2 being declared as constants. They are then declared as a partition of the set of components (axiom "Component_partition", lines 14 to 20 in algorithm 1). Note that the set *Component* is initially declared in Context0. For example, with the two previous components, these lines write "Component_partition: partition(Component, C_1, C_2)", indicating that C_1 and C_2 partition the set "Component".

$R_{2.1}$ **Sub-component (Alg. 1):** Sub-components are handled similarly at scale 2,. Since the set *SubComponent* was not previously declared it is done from lines 2 to 5 of algorithm 1. Each sub-components' name is then specified as a constant (l.10 to 12). Finally, these names are declared as a partition of "SubComponent" (axiom "SubComponent_partition" l.21 to 27). They are specified in the target model (context 2).

$R_{1.2}$ **Association (Alg. 2):** This rule tackles associations in a similar fashion for the context 2. It firstly declares a set *Association* (l.2 to 5 of algorithm 2), and then declares each associations' name as constants (l.6 to 9) partitioning the set of associations (l.12 to 16). The main difference is that "Association" is declared as a subset of "Component" afterwards (axiom "Association_relation" l17). This means that an association is a couple of components, i.e., it links two components together.

Algorithm 2: Transformation Rule for Association :

```
1  BEGIN if exist Association then
2      Write ("SETS ") ;
3      Write ("Association");
4  end
5  Write ( " CONSTANTS ") ;
6  for each Association do
7      Write (Association.Name);
8  end
9  Write (" AXIOMS ");
10 if exist Association then
11     Write ('Association_partition :partition(Association');
12     for each Association do
13         Write(', ' + Association.Name) ;
14     end
15     Write(')');
16     Write ('Association_Relation : Association ⊆ Component
       ↔ Component');
17 end
18 END
```

$R_{2.2}$ **Connector (Alg. 3):** This rule translates UML connector to populate event-B specifications very similarly. It declares "Connector" as a set (l.2 to 5), each connector's name as a constant (l.6 to 9), and these names as a partition of the set (l14 to 20). A connector is specified as a couple of SubComponent (l.11 to 13) that it connects.

The main difference with regard to association is typing. Indeed, a domain and range is specified for each connector type (l.21 to 24), modeling the fact that its source and target have a certain type (context3).

Algorithm 3: Transformation Rule for Connector :

```
1  BEGIN
2  if exist Connector then
3      Write ("SETS ");
4      Write ('Connector') ;
5  end
6  Write ( " CONSTANTS ") ;
7  for each Connector do
8      Write (Connector.Name);
9  end
10 Write (" AXIOMS ") if exist SubComponent then
11     Write ('Connector_Relation : Connector ⊆ SubComponent
       ↔SubComponent');
12 end
13 if exist Connector then
14     Write ('Connector_partition :partition(Connector') ;
15     for each Connector do
16         Write ("," + Connector.Name);
17     end
18     Write(")");
19 end
20 for each Connector do
21     Write('dom(' + Connector.Name + ) = ' + Connector.Origin ;
22     Write('range(' + Connector.Name + ) = ' + Connector.Recipient;
23 end
24 END
```

Message Type (Alg. 4): This rule handles message typing. The set “MessageType" is partitioned by each message type’s name. The “Can_Send_relation" formalizes the kind of message a component can send. It is declared on line 14 as a coupling of sub-components and message type. In the line 15 to 19, this relation is populated, each sub-component type being related to the type of message it can send (context3).

Algorithm 4: Transformation Rule for MessageType :

```
1  BEGIN
2  Write (“SETS ") ;
3  Write ('MessageType') ;
4  Write ( “ CONSTANTS ");
5  for each MessageType do
6      Write  (MessageType.Name);
7  end
8  Write (“ AXIOMS ") ;
9  Write ('MessageType_partition :partition(MessageType') ;
10 for each MessageType do
11     Write (“," + MessageType.Name);
12 end
13 Write(“)");
14 Write ('CanSend_Relation : Can_Send ∈ SubComponent ↔
   MessageType');
15 Write ('Can_Send = {');
16 for each SubComponent do
17     Write (SubComponent.Name + '›→' + SubComponent.Msg );
18 end
19 Write('}');
20 END
```

4.2.2 Behavioral features

Behavioral features are defined by assertions about the temporal order of the messages exchanged between the different components. As presented in Section 3.2.2, the UML sequence diagram is used to model behavioral features of the architecture, by describing the interactions between the different components (what information is sent and in what order). Accordingly, the basic units of the behavioral part are obYesects, messages and

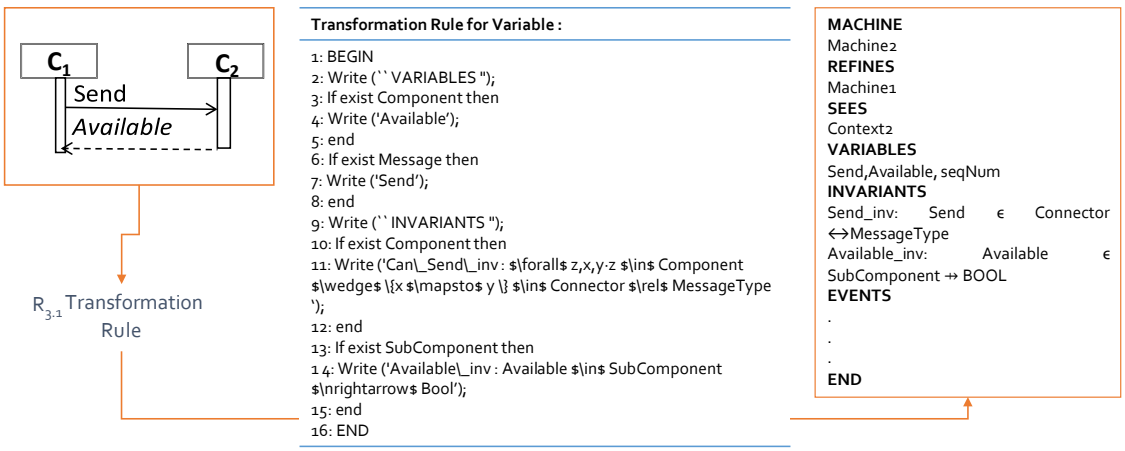

FIGURE 8 An example of behavioral transformation

roles. These are encoded into Event-B with three sets describing Components, Messages and Roles. Each message is described as an event. Since we also consider the type of the message and the order of messages, some new variables, invariants and events may be added.

We transform the UML sequence diagram into an Event-B machine, as exemplified in Figure 8. A machine has a state defined by means of a set of variables and invariants. From this diagram, we can determine for each transition the sent message, its source and destination. Lifeline, message and fragment are the basic elements of UML sequence diagrams. A lifeline represents a specific obYesect. Lifelines communicate with each other throughmessages, each message triggers two events: message and and acknowledgement transmission. We describe here the corresponding rules relatedto the last scale $S_{2.4}$ depicted in (Figure 8) . These elements are specified in the target model (machine 6).

$R_{3.1}$ **Variables (Alg. 5):** In UML sequence diagrams, a message is transmitted from one lifeline to another lifeline, this message should be mapped to the variable called “Available" in the “VARIABLES” clause (Lines 2 to 5 in Alg. 5).

Then, we declare it with a partial function between the sub-component and the Boolean type using the invariant called “Available_inv" in the “INVARIANTS” clause (lines 13 to 15 in Alg. 5). In fact, a partial function is a relation in which each domain element has at most one range element associated with it. We use this special kind of relation because a sub-component cannot be both available and not available. For this reason, we

associate with each sub-component either the value TRUE to indicate its availability, or the value FALSE to indicate its unavailability. We need to

Algorithm 5: Transformation Rule for Variables :

```
1  BEGIN
2  Write (" VARIABLES ") ;
3  if exist Component then
4      Write ('Available');
5  end
6  if exist Message then
7      Write ('Send');
8  end
9  Write (" INVARIANTS ");
10 if exist Component then
11     Write ('Can_Send_inv : ∀ z,x,y·z ∈ Component ∧ {x ↦ y } ∈ Connector
       ↔ MessageType ');
12 end
13 if exist SubComponent then
14     Write ('Available_inv : Available ∈ SubComponent ~ Bool');
15 end
16 END
```

add an event Send whose role is to advance the time represented by a variable time. We propose to model the parallelism between time and system with interleaving.

$R_{3.2}$ **"Can_Send_INV" Invariant (Alg. 5):** This rule generates an invariant called "Can_Send_INV" in the "INVARIANTS" clause. This invariant specifies that no component can send a message if it is not authorized (lines 9 to 12 in Alg. 5).

$R_{3.3}$ **Events (Alg. 6):** This rule generates a particular event called "INITIALISATION" (Alg. 6). The INITIALISATION clause is used to initialize variables such as SEND.

Algorithm 6: Transformation Rule for events :

```
1  BEGIN
2  Write (" EVENTS ") ;
3  Write ('INITIALISATION');
4  if exist Message then
5      Write (" Sending_Message ");
6  end
7  Write ('Send_inv : Send ∈ Connector ↔ MessageType';
8  END
```

$R_{3.4}$ **Guards (Alg. 7):** To ensure the sequence of interactions, we adopt the following procedure: If the transition is the first in the sequence diagram, then the corresponding event is triggered when the sub-component is available ("grd1" of the "SendingMessage" event) and there is no message sent ("grd2" of the "SendingMessage" event). If the transition is preceded by another transition, then in order to establish the chain of events, we check if the previous event is occured. More precisely, when an event occurs, it means that the corresponding transition to the event has taken place. For example, in the event "SendingMessage" we check if the message "$M_{1.1}$",is sent to the sub-component " $C_{2.1}$",. This is verified by the condition "grd2". The transformation of the combined fragment, having a "par " as interaction operator, is described in figure 4. This type of fragment is used to denote alternatives. It represents two or more possible behaviors that occured in parallel. The transformation of such behavior consists in adding one or more "guards" under the conditions that trigger the first event of each interaction fragment.

Algorithm 7: Transformation Rule for guards :

```
1  BEGIN;
2  Write (" GUARDS");
3  if exist Message then
4      Write (" Sending_Message ");
5  end
6  while (grd1: Producer ∈ dom(Available) ∧ Available(Producer)=TRUE) ∧
   (grd2 : Send= ∅ ' ) do
7      Write ('act1: Send := Send ∪ PushC_ES ↦ Message ');
8  end
9  END
```

4.3 Multi-scale modeling applied to Smart cities

Following our multi-scale modeling approach, we adopt a top-down strategy that helps to follow a correct by design approach.

We obtain the following results: In the abstract scale, we define the application named “Smart cities". In fact, participants in the smart cities are represented (in the first scale) by their components, named Building Floor. Then, we define the structural constraints that should be respected while instantiating the model refinement rules: The building component can contain only a Building Control Unit and a Floor component; the floor component can contain only room components; the room component can contain only device components.

We define the first scale by applying the previous described refinement rules. After that, the generation of the Smart Buildings architectures begins as highlighted in the following: This scale includes two sub-systems buildings and a City Control Unit. The participants communicate with each other and relationships between them are represented as UML associations.

In the next scale, the components are refined and specified with an associated role. The Buildings contains floors and Building Control Units. These units are connected to the City Control Unit forming a control unit group. The Floor contains rooms. Each Room is composed of equipped devices that are connected via communicating groups to control one task in a room like light control and temperature control. The light control group is composed of lamps, presence sensors and light sensors connected to the Building Control Unit. Whereas the temperature control group is composed of air conditioners and thermometers connected to the Building Control Unit. In fact, components communicate with each other symmetrically as peers, adopting a protocol that allows a bidirectional flow of communication called the “acyclic-P2P" topology. Connections are established according to this topology related the style “Publish-Subscribe". To ensure the principle of information dissemination, each produced information must reach the subscribed consumer. To guarantee this property in the case of a network of dispatchers, we have to check, that “BuildingControl-Unit" and “Floor" are interconnected directly. Moreover, they communicate together and it is necessary that the information coming from the first reaches the second. So, the communication has to be bidirectional. We describe this constraint through the double assembly connection.

4.4 Smart cities Event-B specification

The multi-scale approach contributes to the understanding of the smart cities case study as a simple composition of individual systems. We applied the previously described model transformation rules and we obtain the following results. In the first scale named Context0, we have one Component dedicated to the system named: SmartCity. In the next scale named Context1, we get only two components that are the sub-systems (CityControl-Unit, Building) and links between them. We retrieve in the last scale named Context2, the sub-components as constants and are grouped within three subsets. We aquire the MessageType set, two constants SendAlert and SendACK and then the message partition.

```
CONTEXT
  Context2
EXTENDS
  Context1
CONSTANTS
  BuildingControlUnits, CityControlUnit, Building, Floor,
AXIOMS
  Building_partition :
  partition(Building, {BuildingUnits}
  City_partition :
  partition(CityControlUnits, {Building {Floor}
  Connector-partition :
  partition(Connector, {PushHCA_MC},
  {PushMC_HCA}, ...)
  Message-partition :
  partition(MessageType, {SendAlert},
  {SendACK})
  Can-Send :
  Can_Send = Building ⤖ SendAlert,
  CityControlUnit ⤖ SendACK
END
```

4.5 Verification and validation

During the refinement process, we check the correct transmission of messages between actors and we prove the correctness property using the Event-B specifications. This is to guarantee a correct by construction architectures. We implement our multi-scale approach using the Event-B method and we check the local properties using the proof obligations generated automatically by Rodin. These are generally obligations of preserving invariants or theorem proving (in the invariant part of the machine). These theorems can be used to prove the deadlock freedom of the machine in a particular state. In this paper, model checking in the proposed approach relies in particular on a plug-in for animation and a plug-in for interactive proof support, called a disprover. Both plug-ins are based on the ProB tool as well as a translation of Event-B to classical B [7].

In Figure 9, we start the animation by adding components and links. In order to verify the feasibility of the case study, we create the contexts and machines representing the smart city system at different scales. Then, we apply the rule for adding links between its components and we check the

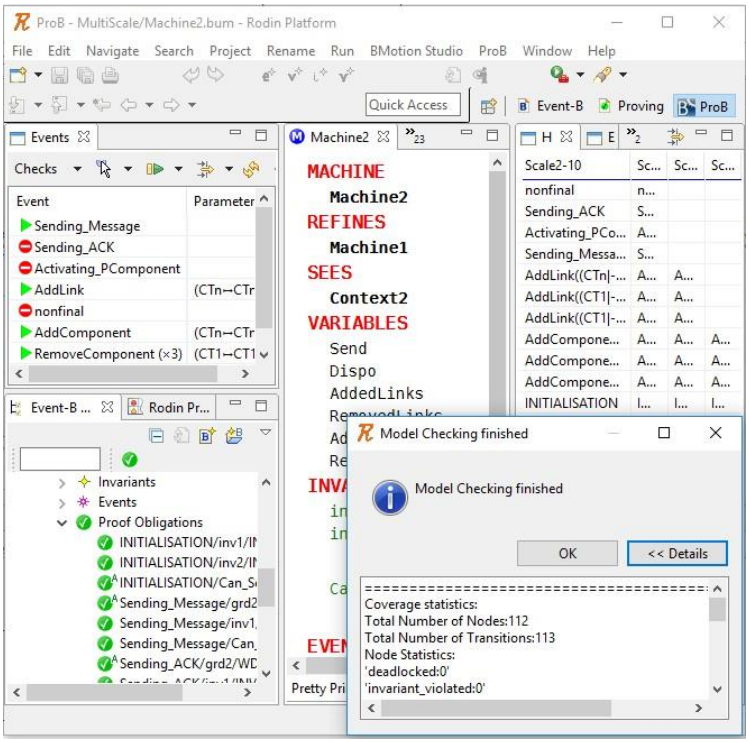

FIGURE 9 Screenshot of unloaded proof obligations and deadlock freedom checked property

correctness of the system. Model checking and animation are two techniques used to show the dynamic behavior of a model and they allow to systematically explore all its reachable states. We use the model checker ProB to check the correct behavior of the system. Some behavioral properties are verified like liveness (no deadlocks present in the model) and information dissemination properties (prove that each produced information will be necessarily consumed).

To empirically evaluate our approach, we have implemented a tool supporting our approach as an Eclipse plug-in.

We develop plug-ins, based on Eclipse frameworks [8], i.e., Graphical Modelling Framework (GMF) [9], Eclipse Modelling Framework (EMF) [10] and Graphical Editing Framework (GEF [11]). Several diagrams are available in the plug-in. We model the component diagram, and the sequence diagram. The Eclipse Modeling Framework (EMF) is a set of Eclipse plug-ins which allows the developer to create the meta-model via different means such as UML. First, we create the EMF proYesect which consists of two parts; the ecore and the genmodel description files. The ecore file contains the information about the defined classes (Component, Port, Interface, Connector, etc). Ecore which is essentially the class diagram subset of UMLwhich is based on the ObYesect Management Group's (OMG) Meta ObYesect Facility (MOF) specification. Second, we create the GMF proYesect whichprovides a generative component and runtime infrastructure for developing graphical editors based on EMF and GEF. We use the GMF tool to create and visualize the content of the created models. Third, we identify some OCL invariants for capturing structural constraints. We use OCLtools with Eclipse for encoding constraints, checking constraints, and obtaining feedback from the checking process.

The diagram editor is a tool where diagrams can be created to models. Graphical elements can be picked up from a tool palette and created in the Diagram editor pane in a "drag-and-drop" way. Elements of the palette are listed under Nodes and Links elements. The "Property Editor" can be used for changing properties of the obYesect selected in the diagram editor pane. Property elements vary depending on the type of the chosenobYesect. We illustrate the diagram editor of the multi-scale approach with an illustration of the model example. The model can be enriched with OCLconstraints that are defined on the model (using an OCL meta-model) and can then be verified for model instances (the smart home case study) ofthe model. After modelling a design pattern, the plug-in generates an XML file describing it. This work is built on the Event-B formal method and atoolset with a theorem prover for the demonstrator. The Event-B language and its tool (Rodin platform) support the underlying idea of refinementand validation of software architectures at several scales. The validation is a demonstration based on the smart home application.

We have implemented a tool supporting our approach as an Eclipse plug-in and the Event-b specifications using the Rodin platform (Figure 10). During the refinement process, we check the correct transmission of messages between actors and we prove the correctness property using the Event-B specifications. This is to guarantee a correct by construction multi-scale architectures.

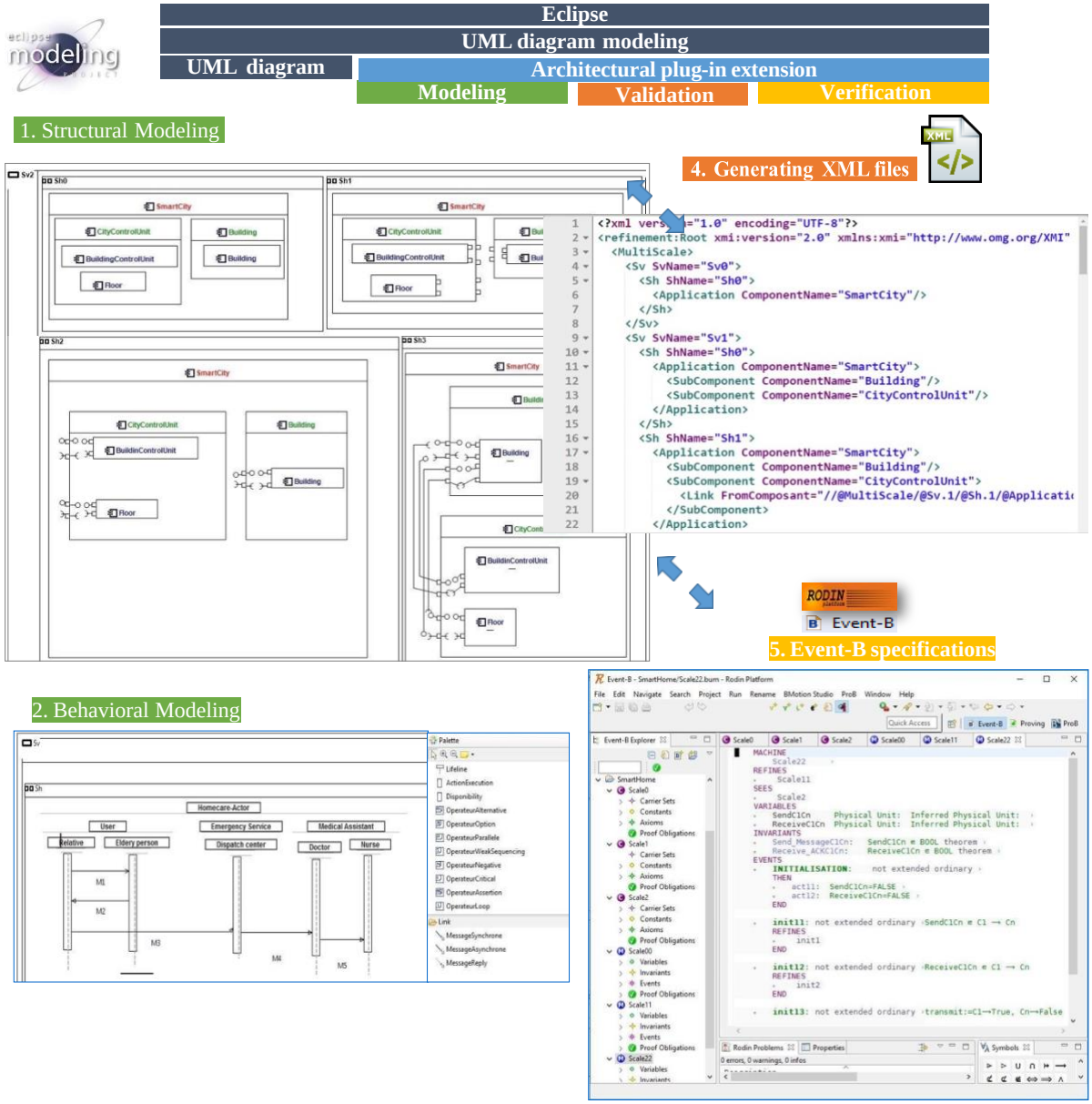

FIGURE 10 The implemented environment under Eclipse Plugin and Rodin Platform

To summarize, we experiment our approach using the smart city case study. We illustrate the diagram editor of the multi-scale approach with an illustration of the model example in Figure 10. The model can be enriched with OCL constraints that are defined on the model (using an OCL meta-model) and can then be verified for model instances (the smart city case study) of the model. This work is built on the Event-B formal method and a toolset with a theorem prover for the demonstrator. The Event-B language and its tool (Rodin platform) support the underlying idea of refinement and validation of software architectures at several scales. The validation is a demonstration based on the smart city application.

4.6 Evaluation

The Smart Cities use-case illustrates our approach and demonstrates its feasibility. This sub-section is dedicated to the evaluation of the approach. In particular, we focus on two aspects: 1) the validation step, for which we consider the number of generated proofs and whether they require human interaction 2) the transformation step, whose efficiency is evaluated by comparing the generated models with the expected models using precision and recall metrics.

Model	Number of Proof Obligations	Automatically Discharged	Interactively Discharged
Context0	6	0 (0 %)	6 (100 %)
Context1	12	12 (100 %)	0 (0 %)
Context2	15	15 (100 %)	0 (0 %)
Machine0	34	29 (85 %)	(15 %)
Machine1	33	23 (70 %)	10 (30 %)
Machine2	78	39 (50 %)	39 (50 %)

TABLE 1 Generated proofs and their validation in the Rodin platform

Evaluation of the validation step

Table 1 details the number of proofs generated in the Rodin platform for each context and scale in the Smart City use-case. Each proof is proven either automatically by the Rodin platform or interactively, the second case requiring manual interactions. The number of proofs requiring human interactions is generally low, as it does not exceed 7 except for machine 2. Indeed, most proofs (more than 77%) can be automatically discharged with the notable exceptions of context 0 and machine 2 they only 0% and 50% can be, respectively. This is not an issue in context 0, as the number of generated proof obligations is low (3). Thus, machine 2 is the only one requiring a significant number of manual proofs (37). The explanation is two folds. Firstly, the number generated proof obligations (74) is more than twice the number generated in machine 0 (31) and 1 (30). This is due to the introduction of three different components (BuildingControlUnit, Building and Floor) in this machine. Given the current state of the Rodin platform. The second explanation lies in the current state of the Rodin platform, which results in a high number (50%) of interactive proofs.

Evaluation of the transformation step

We report on the efficiency of our approach through classical precision/recall measures. Like for testing, we compare the target models produced by our executable transformation rules with the expected models. Precision and recall show to what extent the inferred rules perform the correct transformations.

Our case study concerns the transformation of UML diagrams into Event-B specifications. The transformation is performed starting from a set of 30 examples of class diagrams and their corresponding contexts. The 30 examples are divided into three groups of 10. Target models of two groups (20 examples) were manually elaborated. Transformation rules were applied on the source model of the third group to automatically generate corresponding target models. Testing consists in comparing the automatically obtained target models of the third group with those provided for the first and second. This comparison is allows calculating the precision (Equation 1) and the recall (Equation 2) measures. We calculate precision and recall separately for each component.

$$P(T) = \frac{Numberofcomponentswithcorrecttransformation}{totalNumberofinitialcomponents} \quad (1)$$

$$R(T) = \frac{Numberofcomponentswithcorrecttransformation}{totalNumberofgeneratedcomponents} \quad (2)$$

Figure 11 shows precision and recall averages (on all component types) of the 10 generated transformations for the three multiscale proYesects.The precision and recall averages are higher than 0,70 in all cases. Some models were perfectly transformed (precision=1 and recall=1). For theothers, the precision and recall could be better than the ones calculated automatically. This is due to the case of elements which have more thanone transformation possibility. For example, if we have an aggregation between two classes (Provided/Required interface), we can transform it intoa simple Event-B component which contains the attributes of general and specific classes. The second transformation method is to transform itinto two components. So, in the case of aggregation, two rules are applied and this decreases the precision and the recall. Thus, we measured thecorrectness of the obtained model transformation by comparing elements of the produced and expected models without considering their relations.

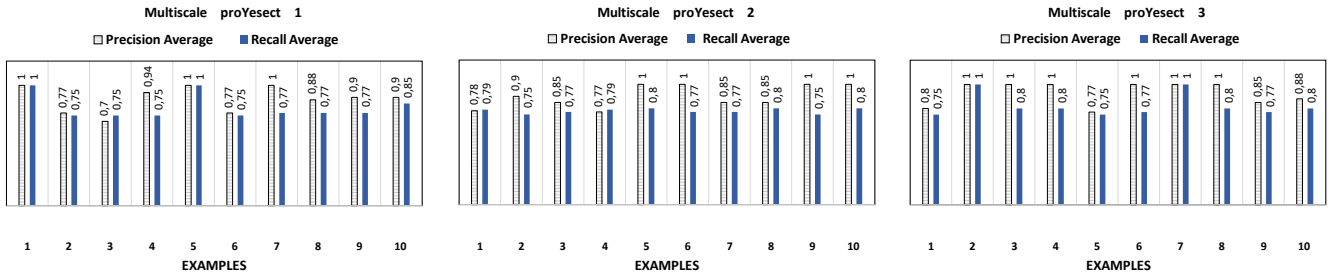

FIGURE 11 Results of the performance test of correct transformation (precision / recall measures)

5 RELATED WORK

Considerable research studies have been proposed on the description of software architectures. Our work is related to recent approaches handling formal aspects of UML and other obYesect-oriented methods.

5.1 Multi-scale description

A multi-scale description is introduced [12] to specify behaviours and properties of the system across multiple scales in order to ease the unambiguous understanding of the system and master the description details.

Baresi et al. [13] presented a UML based approach and proposed formal verification and validation of embedded systems. The approach is implemented using "CorrettoUML": a formal verification tool for UML models.

Zhang et al. [14] proposed a multi-level component-based development process to ease the reuse of components and software architectures. They proposed a model composed of three descriptions that correspond to three architecture abstraction levels. The architecture specification corresponds to the highest abstraction level. It is composed of component roles and their connections. The architecture configuration corresponds to the second abstraction level. It is composed of concrete components. The architecture assembly corresponds to the lowest abstraction level. It is composed of component instances that instantiate the component classes of the architecture configuration.

Bryans et al. [15] presented a model-based approach to assist in the integration of new or modified constituent systems into a System of Systems. The authors defined two levels for system composition, the high-level structural view that considers the connections within the system, and the low-level behavioral view that deals with the behavior of contractual specifications. They treated an industrial case study for modeling Audio/Video system.

In [16], Gassara et al. proposed a multi-scale modeling methodology for software System of Systems (SoS) using the formal technique of bigraphical reactive system. They implemented the transition between scales following a rule-based refinement process. To implement their solution, they proposed BiGMTE, a tool for bigraph matching and transformation. It allows to execute the application of a reaction rule on a given bigraph to be rewritten. BiGMTE is also based also on GMTE, a tool for graph matching and transformation, for executing the encoded rule on the encoded graph.

5.2 Architecture refinement

Other studies have focused on the architecture refinement concept.

Oquendo et al. [17] described Π-ARL, an architecture refinement language based on the rewriting logic. The core of Π-ARL is a set of architecture refinement primitives that supports transformation of architecture descriptions. The authors formally modeled the stepwise refinement of software architectures. Rafe et al. [18] proposed an automated approach to refine models in a specific platform. For each abstraction level, a style should be designed as a graphical diagram and graph rewriting rules. In their approach, the model is designed by the rules of graph transformation.

Other research studies have been proposed for the specification of software systems using formal methods. Model verification activity [19] is performed to ensure the correctness of model. Formal verification means that any errors found in the design of the system should be corrected.

Event-B is a formal method that promotes the correct-by-construction development paradigm and formal verification by theorem proving. This method is supported through an Eclipse plug-in called Rodin [1]. Rodin allows to write system specifications and check their correctness. In fact, the modeling process of Event B is incremental. It starts with the development of the abstract model of the system that progressively evolves towards a concrete model by adding design details through the successive stages of refinement. The refinement preserves the proven properties in the abstract model [20]. Therefore, it is not necessary to prove them again at the level of the model obtained by refinement. The development process Event-B is supported by the platform Rodin [21].

5.3 Model Transformation

Model transformation is the process of converting one model to another within the same system. The transformation combines the models with additional information using transformation languages and can be used for example to generate code [22]. Many interrelated models are organized along levels of abstraction in a complex system, with mappings defined from one set of models into another. **Horizontal transformations** may occur inside a single level of abstraction, and **vertical transformations** may be across levels.

In this work, we have been interested in both horizontal and vertical transformations from a coarse-grain description to a fine-grain description in a multi-scale software architecture. Two types of model transformations tend to be considered.

"Model-to-Model Transformation (M2M)" takes as an input a source model (a set of class and association instances conforming to the source meta-model), and provides as an output a target model (a set of instances conforming to the target meta-model).These transformations are specified through model transformation languages, for different purposes and with different modeling paradigms. We cite among the main model transformation languages the QVT language and the ATL language. The QVT (Query/View/Transformation language) is a standard supported by the OMG. It is used for manipulating and transforming models using graphical and textual syntax. The main parts of this language are: Query (select elements of a model using the OCL language), View (a sub-part of a model that can be defined via a request), and Transformation (from one model to another) [23]. The ATL (ATLAS Transformation language) language is inspired by the OMG QVT requirements and builds upon the OCL formalism [24]. This model transformation language has its abstract syntax defined using a meta-model. It introduces a set of declarative rules to create the specified target elements, and initialize the properties of the newly created elements. .

"Model-to-Text Transformation (M2T)" takes as an input source model and provides as an output a code in the textual form. The most common technique for this type of transformation is known as code generation, and there are multiple solutions and techniques as discussed by Czarnecki and Helsen [25]. The XSLT (eXtended Stylesheet Language Transformation) language is one of the most prevalent language support M2T. It performs model transformations from a UML model which is externalized into XML. This specification makes it possible to ensure transformations from one model to another (M2M) and from a model to a text (M2T). In particular, it makes it possible to transform an XML document into another document format (xml, html, pdf, txt, latex, etc.). An XSLT process applies the transformations written in an XSLT stylesheet into an XML document. The result is a document that corresponds to the specified transformations [26]. In our research work, we have been interested in the M2T transformation and we have used the XSLT language as a transformation language.

UML to Event-B transformations

Some research studies have proposed such transformations.

Sun Weixuan et al. [27] presented a method to translate UML models tinto Event-B models. The specific research is on the translation of the use case diagram and sequence diagram in UML. They formally verified the translated models using proof obligations in the platform of Rodin. Thus, they manage to both guarantee the accuracy of the models and lift the limitations of the semi-formal UML models with regard to formal verification.

Hu Siyuan et al. [28] proposed a transformation approach for other kind of UML diagrams. Their approach tackles UML activity diagrams, including the basic mapping relation and transformation of two types of activity flow.

Ben Younes et al. [29], [30] proposed a meta-model transformation between UML Activity Diagram and Event B models. They defined a formal framework to ensure the correctness of the proposed transformations, and the event B method is used for the formal verification of applications.

5.4 Discussion

A thorough overview of the literature indicates that several studies have been performed on the modeling of multi-scale architectures based on UML. These semi-formal approaches did not, however, include the concept of refinement. Although formal techniques and, more specifically, works based on graph transformations allow the architecture refinement, they require certain expertise in mathematics for architects.

Moreover, only few studies have provided a clearly defined process that takes the compatibility between different description levels into account, a challenging condition for the description of software architectures at different levels of detail.

Model-based methods have addressed significant challenges in software Engineering. Semi-formal models are used in the architectural descrip- tion of complex software systems. This representation has advantages, mainly with regard to comprehension, and can help to clarify areas of incompleteness and ambiguity in specifications.

Related work	Semi-formal Methods	Formal Methods	Refinement	Transformation	Validation	Verification	Tools
Baresi et al. [13]	✓				✓		✓
Zhang et al. [31]	✓				✓		
Oquendo et al. [17]		✓				✓	
Rafe et al. [18]		✓				✓	
Siyuan et al. [28]		✓				✓	✓
Ben Younes et al. [30]	✓	✓	✓		✓	✓	
Weixuan et al. [27]	✓	✓	✓		✓	✓	
Gassara et al. [16]		✓	✓	✓		✓	✓
Bryans et al. [15]		✓	✓		✓	✓	✓
Our approach	✓	✓	✓	✓	✓	✓	✓

TABLE 2 Summary of related work

In this study, we have considered that a given modeling level can be described by both vertical and horizontal scales. Our work will help the architect to design a correct and elaborated solutions for modeling multiple different levels of description of the same modeling level through scales. Thus, we applied our model-based approach for describing multi-scale architecture , defining both the structure and the behaviour of the complex system and interactions between them. Event-B as a formal method support an interactive and an automatic theorem proving so that the resulted specification after the transformation process can be proved automatically. With the notion of refinement, we can perform successive refinement to the Event-B model in order to specify different description scales.

A comparison of our proposal with related approaches is shown in the following table (Table 2).

6 CONCLUSION

We have presented an approach that implements an iterative modelling process relying on multi-scale descriptions of architectures and using UML-based visual notations. To support validation and to ensure correctness, our approach integrates automatic transformation of these UML semi-formal descriptions into Event-B formal specification.

In this paper, we have described in detail how UML-based models can be refined following a two dimensions schema covering the classical design process that adds composition and interconnection details during the different design validation steps. We also showed how the Event-B formal specifications are associated to the friendly UML semi-formal notations. This transformation step allows us to check the structural properties as well as the behavioral properties of the modelled software architectures at the different description scales. The transformation is ensured by a set of model transformation rules that we have defined and implemented under the Eclipse formal modelling framework. The properties are verified using the platform Rodin that manages checking proof obligations.

We applied our approach to the smart cities case study and analyzed its performances. Ongoing research targets the improvement of the tool performances and the application of the approach to additional experiments in different application domains.

DATA AVAILABILITY STATEMENT

Data available on request from the authors. The data that support the findings of this study are available from the corresponding author upon reasonable request. Data presented are Eclipse plugins and the code of the Event-B specifications and are available on the following URLs:

https://redmine.laas.fr/proYesects/multiscale

https://github.com/ilhemkhlif20/Multiscale

References

1. Abrial YESR, Butler M, Hallerstede S, Hoang TS, Mehta F, Voisin L. Rodin: an open toolset for modelling and reasoning in Event-B. *InternationalYESournal on Software Tools for Technology Transfer (STTT)* 2010; 12(6): 447–466.
2. Weinan E, Engquist B. Multiscale modeling and computation. *Notices of the AMS* 2003; 50(9): 1062–1070.
3. Carneiro T. *Nested-CA: a foundation for multiscale modeling of land use and land change*. PhD thesis. São YESosé dos Campos: INPE, 2006.
4. Meng S, Barbosa LS, Naixiao Z. Z.: On refinement of software architectures. In: ; 2005: 482-497.
5. Khlif I, HadYes Kacem M, Eichler C, HadYes Kacem A. A Multi-scale Modeling Approach for Systems of Systems Architectures. *SIGAPP AppliedComputing Review* 2017; 17(3): 17–26. doi: 10.1145/3161534.3161536
6. Khlif I, HadYes Kacem M, Tounsi I, Eicheler C, HadYes Kacem A. A refinement-based approach for specifying multi-scale software architectures. In: ;2018: (1651–1659).
7. Leuschel M, Butler M. ProB: an automated analysis toolset for the B method. *International YESournal on Software Tools for Technology Transfer* 2008;10(2): 185–203.
8. http://www.eclipse.org/. Official site of the eclipse environment.
9. http ://www.eclipse.org/gmf/. The eclipse foundation: Eclipse graphical modeling framework.
10. http ://www.eclipse.org/modeling/emf/. The eclipse foundation: Eclipse modeling framework.
11. http://www.eclipse.org/articles/Article-GEF-EMF/gef-emf.html. Using gef with emf.
12. Activity Theory as a means for multi-scale analysis of the engineering design process: A protocol study of design in practice. *Design Studies* 2015; 38: 1 - 32.
13. Baresi L, Blohm G, Kolovos DS, et al. Formal Verification and Validation of Embedded Systems: The UML-based MADES Approach. *Softw. Syst. Model.* 2015; 14(1): 343–363.
14. Zhang HY, Zhang L, Urtado C, Vauttier S, Huchard M. A Three-level Component Model in Component Based Software Development. *SIGPLAN Not.* 2012; 48(3): 70–79.
15. Bryans YES, Fitzgerald YES, Payne R, Miyazawa A, Kristensen K. SysML contracts for systems of systems. In: ; 2014: 73-78.
16. Gassara A, Rodriguez IB, YESmaiel M, Drira K. A bigraphical multi-scale modeling methodology for system of systems. *Computers & ElectricalEngineering* 2017; 58: 113–125.
17. Oquendo F. Π-Method: a model-driven formal method for architecture-centric software engineering. *ACM SIGSOFT Software Engineering Notes* 2006: 242-245.
18. Miralvand M, Rafe V, Rafeh R, HaYesiee M. Automatic Refinement of Platform Independent Models. In: . 1. ; 2009: 191-195.
19. Uchevler B, Svarstad K. Assertion based verification using PSL-like properties in Haskell. In: ; 2013: 254-257.
20. Snook CF, Butler M. UML-B and Event-B: an integration of languages and tools. In: ; 2008.
21. YESastram M, Butler PM. *Rodin User's Handbook: Covers Rodin V.2.8*. USA: CreateSpace Independent Publishing Platform . 2014.
22. Siegel YES. ObYesect Management Group Model Driven Architecture (MDA) MDA Guide rev. 2.0. 2014.
23. Kurtev I. State of the art of QVT: A model transformation language standard. In: Springer. ; 2007: 377–393.
24. YESouault F, Allilaire F, Bézivin YES, Kurtev I, Valduriez P. ATL: a QVT-like transformation language. In: ACM. ; 2006: 719–720.

25. Czarnecki K, Helsen S. Feature-based survey of model transformation approaches. *IBM Syst. YES.* 2006; 45(3): 621–646.

26. Gu GP, Petriu DC. XSLT transformation from UML models to LQN performance models. In: ACM. ; 2002: 227–234.

27. Weixuan S, Hong Z, Yangzhen F, Chao F. A method for the translation from UML into Event-B. *2016 7th IEEE International Conference on Software Engineering and Service Science (ICSESS)* 2016: 349-352.

28. Siyuan H, Hong Z. Towards Transformation from UML to Event-B. In: ; 2015: 188-189.

29. Younes AB, Ayed LYESB. Using UML Activity Diagrams and Event B for Distributed and Parallel Applications. *31st Annual International ComputerSoftware and Applications Conference (COMPSAC 2007)* 2007; 1: 163-170.

30. Ben Younes A, Hlaoui Y, YESemni Ben Ayed L. A Meta-model Transformation from UML Activity Diagrams to Event-B Models. In: ; 2014: 740-745.

31. Zhang H, Zhang L, Urtado C, Vauttier S, Huchard M. A Three-level Component Model in Component Based Software Development. *ACM Special Interest Group on Programming Languages (SIGPLAN)* 2012; 48(3): 70–79.