Journal of King Saud University - Computer and Information Sciences

Formal development of planning problems by coupling Event-B and PDDL

--Manuscript Draft--

Manuscript Number:	JKSUCIS-D-21-01169
Article Type:	Review Article
Keywords:	correct by construction; Event-B; code generation; PDDL; automatic planning; refinement
Abstract:	The PDDL language, planners and validators associated with this language allow the description, resolution and validation of planning problems. However, they only allow the reliability of PDDL descriptions to be examined in a posteriori. In this paper, we recommend a formal process coupling Event-B and PDDL that favors the obtaining of reliable PDDL descriptions from an ultimate Event-B model that is correct by construction. Such a process is successfully experimented on the Sliding puzzle game.

Formal development of planning problems by coupling Event-B and PDDL

Abstract

The PDDL language, planners and validators associated with this language allow the description, resolution and validation of planning problems. However, they only allow the reliability of PDDL descriptions to be examined in **a posteriori**. In this paper, we recommend a formal process coupling Event-B and PDDL that favors the obtaining of reliable PDDL descriptions from an **ultimate** Event-B model that is **correct by construction**. Such a process is successfully experimented on the Sliding puzzle game.

Keywords: correct by construction, Event-B, code generation, PDDL, automatic planning, refinement

1. Introduction

Automatic planning is a full-fledged discipline in artificial intelligence. It allows describing and solving planning problems in many fields such as robotics, project management, internet navigation, crisis management, logistics and games. The planning problems are expressed in a concise language of state change operators. A state models a stable situation of the addressed planning problem. It can be an initial, final (also called goal) or intermediate state. Moving from one state to another is governed by labeled transitions. A labeled transition has an origin state, a destination state and a well-defined action. The automatic planning community has developed a de facto standard PDDL (Planning Domain Definition Language) (McDermott et. al., 1998) to formally describe planning problems. In addition, this community has been interested in the generation of plans. Moreover, it has developed validation tools (Howey et. al., 2004) to check whether a given solution plan can be derived from a PDDL description. The formal PDDL[1] language is used to describe the two components of a planning problem: state and state change operators. The state is described by types-containing unstructured objects-and logical predicates. State change operators are described in PDDL by actions with Pre/Post semantics. The precondition describes the applicability condition and the post condition describes the effect of the action. PDDL descriptions are difficult to write, read and develop. Complex PDDL descriptions are subject to errors that are difficult to identify **a priori**, i.e. when writing PDDL specifications. This is because the planners and validators associated with the PDDL language make it possible, at most, to detect errors **a posteriori**. Locating errors in a PDDL description is not easy. This is also true for error correction.

In this work, we advocate the **correct by construction** paradigm (Abrial, 2010) for the formal modeling of planning problems. To achieve this, we propose an approach to **couple** Event-B (Abrial, 2010) and PDDL for automatic planning. The formal Event-B method supports a very rich refinement technique with mathematical proofs for step-by-step correct construction development at various levels of abstraction of Event-B models. The ultimate Event-B model described by **a subset of Event-B** is easily translated into PDDL. Thus, the resulting PDDL description is considered to be correct by construction.

Section 2 presents the PDDL language and its verification and validation tools. Section 3 presents the formal Event-B method with its modeling, refinement, proof and validation steps. Section 4 presents the specifications of the selected case study: sliding puzzle game. Section 5 presents and evaluates two different PDDL models of this case study. Section 6 proposes and evaluates a formal Event-B development of the sliding puzzle game. Section 7 proposes a formal development process for planning problems by combining Event-B and PDDL. Section 8 describes our MDE Event-B2PDDL tool. Section 9 examines related work referring to PDDL and Event-B. Finally, section 9 provides the conclusion.

2. PDDL language

PDDL (McDermott et. al., 1998) is a language for formal modeling of planning problems. Given that a planning problem is defined by states and state change operators. PDDL is based on the first-order predicate language to model the states and on a precondition/post-condition specification to model the state change operators. In this section, we will describe the main components of the PDDL formal language. First of all, we will introduce the discipline of automatic planning as an integral part of Artificial Intelligence (AI). Then, we will present the two macroscopic components of the PDDL, namely: **Domain** and **Problem**. In addition, we will illustrate our point by addressing the Hanoï Towers problem as a planning problem. Finally, we will describe the functionalities of the software tools associated with the PDDL language, namely planners and validators.

[1] In this work we limit ourselves to the initial PDDL language known as PDDL1.2.

2.1 Automatic planning

In AI, planning designates a field of research aiming at automatically generate, via a formalized procedure, a hinged result called plan. This is intended to orientate the action of one or more executors (robots or humans). Such executors are called upon to act in a particular world to achieve a predefined goal. Automatic planning concerns various fields such as: robotics, crisis management, logistics, web services composition, resource management, games, assembly of complex objects, storage management, transportation, elevator management, etc. In (Haslum et al., 2019), a more or less exhaustive list of planning domains is established. Automatic planning makes a clear distinction between two activities: modeling and resolution. The modeling activity aims to describe a planning problem using an appropriate formalism in PDDL, in this case. As for the resolution activity, it concerns the automatic generation of solution plans using planning algorithms (Ghallab et al., 2004) from a PDDL description, knowing that a plan is a sequence of elementary actions leading from an initial state to a goal state. The software tools used to automatically generate solution plans are called planners (Roberts and Howe, 2009) They are based on a planning algorithm. In this work, we will use these planners as black boxes.

2.2 Domain and Problem constructs

A PDDL description has two parts: **domain** and **problem**. The **domain** part encompasses the static and dynamic aspects of a planning domain, while the **problem** part includes the definition of an initial state and the logical condition of the goal states of a given planning problem bearing in mind that a planning domain contains several planning problems. PDDL is a **declarative** language and has a prefixed syntax: operator followed by operands. The **domain** construct offered by the PDDL language essentially allows to define the state and state change operators of a planning domain. A state has several properties and each property is defined by a PDDL predicate. The state change operators are formalized in PDDL by elementary actions. Technically speaking, the Domain construct has several clauses. The construction **problem** depends on the **domain** construct. It reuses the modeling elements coming out from the **domain** construct in order to define its constituents.

2.3 Hanoï Towers in PDDL

In this section, we will illustrate the possibilities offered by PDDL on the famous Hanoï Towers algorithmic problem.

2.3.1 Informal description

This area is a representation of the famous game of Hanoï Towers. In this very constrained domain, a problem is constituted by p pegs and n discs of decreasing size stacked in a pyramid on one of the rods. The pyramid must be moved on another rod. The disks can only be transferred to another rod by respecting the constraint that no larger disk may be placed on top of a smaller disk.

2.3.2 Planning domain Hanoï

```
(define (domain hanoi)
  (:requirements :strips :equality :typing )
  (:types disc - object peg - object)
  (:predicates
    (clear ?x - object) (on ?x - disc  ?y - object) (smaller ?x  - disc  ?y - object) )
  (:action move
    :parameters (?disc - disc  ?from - object ?to - object)
    :precondition (and (smaller ?disc ?to)(smaller ?disc ?from) (on ?disc ?from) (clear ?disc) (clear ?to) (not (= ?from ?to)) )
    :effect (and (clear ?from) (on ?disc ?to) (not (on ?disc ?from)) (not (clear ?to)))))
```

Listing 1: Hanoï planning domain

We have studied, retrieved and adapted a PDDL model of the Hanoï Towers. Indeed, we have attributed a type of the said modeling by bringing two types for efficiency and readability reasons (See Listing 1). The state of this domain has three predicates: *on*, *clear* and *smaller*. The predicate *on* admits two parameters: *?x* and *?y*. The first one is of type *disc* and the other one is of type **object**. Thus, *?y* is polymorphic: it can be either *disc* type or *peg* type. This facilitates inter-pegs displacements. Indeed, we can move either a disc on another disc, or a disc on an empty peg. In conclusion, the predicate **codifies** in PDDL the stacks (or pyramids) of discs including the pegs. The predicate “clear “admits a polymorphic parameter (*?x*) of **object** type and allows to know the availability of a disk or a peg. In other words, the *clear* predicate **codifies** in PDDL the vertices of the stacks. Finally, the predicate *smaller* admits two parameters (*?x* and *?y*): one of disc type and the other polymorphic of **object** type. Such a predicate is supposed to answer the following query: is *?x* smaller than *?y* ?. Naturally, the predicate *smaller* is static and it remains un-

changed during the evolution of the state of the planning domain by the *move* action. The other two predicates *on* and *clear* are dynamic subject to modification by the *move* action. The *move* action moves a disk from an origin to a destination and admits three parameters (*?disc*, *?from* and *?to*). The first parameter is of type *disc* and the two others are polymorphic of type **object**. The precondition of the *move* action stipulates its applicability. It is a non-atomic predicate obtained by composing the atomic predicates (*on*, *clear* and *smaller*) using the two logical connectors "**and**" and "**not**". The post-condition (clause **effect**) updates the state of the planning domain by adding and removing (**not**) from atoms or facts.

2.3.3 The three-disk Hanoï problem

Listing 2 describes in PDDL the three-disk Hanoï problem using the planning domain described in Section 2.3.2. The **objects** clause introduces the three pegs (*peg1*, *peg2* and *peg3*) and disks (*d1*, *d2* and *d3*). The **init** clause initializes the static predicate *smaller* and the two dynamic predicates *clear* and *on* stipulating the initial state of the planning problem handled: all the disks are on the peg peg1 and the available vertices are the two pegs peg2 and peg3 and the disk d1. The goal clause expresses the condition on the goal states: the pyramid on peg peg1 is moved to peg peg3.

```
(define (problem pb3)
(:domain hanoi)
(:objects peg1 peg2 peg3 - peg d1 d2 d3 - disc)
(:init (clear peg2) (clear peg3) (clear d1) (on d3 peg1) (on d2 d3) (on d1 d2)
  (smaller d1 peg1) (smaller d1 peg2) (smaller d1 peg3) (smaller d2 peg1) (smaller d2 peg2) (smaller d2 peg3)
  (smaller d3 peg1) (smaller d3 peg2) (smaller d3 peg3) (smaller d1 d2) (smaller d1 d3) (smaller d2 d3) )
(:goal (and (on d3 peg3) (on d2 d3) (on d1 d2))))
```

Listing 2: Three-disk Hanoï problem

2.3.4 Plan-solution

Listing 3 gives the solution plan corresponding to the three-disk Hanoï problem described in PDDL in 2.3.3. Such a plan is generated by the planner supported by the Web Planner (Magnaguagno et al., 2017) platform. It includes seven instantiated *move* actions, i.e. the parameters are replaced by suitable objects declared in the **objects** clause of the **problem** part. We can see the optimality of the generated plan: 2^3-1.

```
(move d1 d2 peg3)
(move d2 d3 peg2)
(move d1 peg3 d2)
(move d3 peg1 peg3)
(move d1 d2 peg1)
(move d2 peg2 d3)
(move d1 peg1 d2)
```

Listing 3: Obtained plan-solution

2.4 Planner and validator

A planner (Roberts and Howe, 2009) is a rather complex software tool. It is based on heuristic planning algorithms. A planner accepts as input a PDDL description (both **domain** and **problem** parts) and produces, as output, plan-solutions. In addition to the generation of plan-solutions, a planner provides more or less elaborate lexico-syntactic checks. In this work, we used the planners provided online by the two platforms: Web Planner (Magnaguagno et al., 2017) and Planning.domains (Muise and Lipovetzky, 2020).

A validator (Howey et. al., 2004) is used to check whether a given solution plan can be generated from a PDDL description.

3. Formal Event-B method

The formal Event-B method (Abrial, 2010) encompasses a formal language with the same name and a formal development process. The Event-B language allows formalizing both data and treatments. On one hand, data are described using a logico-set language (first-order predicates and set theory). On the other hand, treatments are described using the event concept. The event concept consists of three parts of local parameters, a guard and an action. The first two parts are predicates described using the Event-B logico-set language and the third part is described using a simple Event-B action language. This has five types of actions: deterministic assignment (:=), the two non-deterministic assignments (:∈, :|), parallel action (||) and skip action. In Event-B, data and processing are grouped in two syntactic constructions CONTEXT and MACHINE (See 3.1).

The formal development process supported by Event-B via its Rodin platform (Voisin and Abrial, 2014) is based on successive refinements with mathematical proofs. Such a process starts with a coherent abstract model formalizing the concerned application. The coherence of this model is obtained by discharging the Proof Obligations (POs) associated with this model. These POs are considered as correction criteria defined by the Event-B theory. Technically speaking, these POs are mathematical lemmas, automatically generated by Rodin's Proof Obligation Generator component, to be discharged automatically or interactively using the proofs provided by Rodin. Then, a multi-step refinement strategy is applied. These steps form a refinement chain. Each step takes an abstract model as input and produces a refined or concrete model as output. The refinement relationship between the two abstract and refined models is formally verified by appropriate POs. The final refinement step produces a correct by construction concrete model with respect to the initial abstract model. This is explained by the fact that the refinement relationship is transitive.

3.1 CONTEXT and MACHINE constructions

The CONTEXT construction brings together modeling elements related to the static aspects of the application to be modeled. Such elements concern sets, constants, axioms and theorems. Axioms are supposedly true properties attached to sets and constants. The theorems must be demonstrated automatically or interactively using the proofs of the Rodin platform.

The MACHINE construction brings together modeling elements related to the dynamic aspects of the processed application. Such elements concern variables, invariant, theorems and events. The variables form the state of the machine. The machine invariant includes invariance properties describing intra and inter-variable constraints of the machine. The theorems must be deduced logically within the machine. Events can act on the state of the machine **by preserving** its invariant. A particular event called INITIALISATION allows initializing the state of the machine **by establishing** its invariant. Evidently, an Event-B machine can use modeling elements coming out from an Event-B context via the **SEES** relationship.

3.2 Set representation versus predicative representation

In Event-B, the data are formalized thanks to its logic-set language: logic of first-order predicates augmented by set theory (sets, relations and functions in the mathematical sense). The **set** data include constants and variables of a set type (abstract sets introduced in the SETS clause of an Event-B context and predefined set $\mathbb{N}$), function (partial, total, injective, surjective, bijective and lambda), relation and surjective relation. While **predictive** data includes constants and variables of type BOOL (predefined set with FALSE and TRUE) and total function. The latter has as starting set either an abstract set or the Cartesian product of several abstract sets. And its ending set is imperatively BOOL. In the formal Event-B development process based on successive refinements with mathematical proofs, the set data are generally used in the **beginning** of the refinement chain and the predicative data are used in the **end** of the said chain (See 3.3.3).

3.3 Simple Travelling Salesman Problem in Event-B

3.3.1 Specifications

It is a simple version of the Travelling Salesman Problem (TSP), in which the goal is all the places to be visited, and where the displacement operator can be applied between any two locations.

3.3.2 Abstract model

In this section, we propose an Event-B model that models a simplified version of the Travelling Salesman Problem: inter-city distances are not modeled, i.e. they are assumed to be identical. Listing 4 provides in Event-B the static data of the simplified Travelling Salesman Problem. The business concept '*city*' is modeled by the abstract set *CITY*. Two constants are introduced: *complete_graph* and *X*. The *complete_graph* is modeled by a total and surjective relation *complete_graph* from *CITY* to *CITY* (See *axm1*). The two axioms *axm2* and *axm3* stipulate that the relation *complete_graph* is non-reflexive (no loops in the city graph) and symmetric (city graph is oriented). The axiom *axm4* models the following fact: a given city is related to all other cities (completeness of the city graph). The constant *X* models the starting city. It is a given city (See *axm5*).

Listing 5 models the dynamic aspects of the TSP problem. the state of the TSP problem is modeled by two variables *visited* and *current_city* expressing respectively the cities already visited and the current city. The typing of these two variables is provided by the invariant properties labeled *inv1* and *inv2*. The *m0* machine offers two events: INITIALISATION and *move*. The INITIALISATION event sets the initial state: no city visited (*visited* := $\emptyset$) and the current city considered as the starting city is any city (*current_city* :=*X*). Moreover, the *move* event allows you to move from a valid state to another equally valid state. The tour of n cities will be carried out after executing n crossings of the *move* event. The (*c0*, *m0*) Event-B model is considered correct

with respect to the Event-B theory. Indeed, all the proof obligations relating to this model have been discharged by the provers of the Rodin platform.

CONTEXT c0
 SETS CITY
 CONSTANTS complete_ graph,X
 AXIOMS
  axm1:complete_ graph∈CITY<<->>CITY
  axm2:∀v·(v∈CITY⇒v↦v∉complete_graph)
  axm3:∀x,y·(x∈CITY∧y∈CITY∧x≠y∧x↦y∈complete_graph⇒y↦x∈complete_ graph)
  axm4:∀v1·(v1∈CITY⇒(∀v2·(v2∈CITY∧v2≠v1⇒v1↦v2∈complete_graph)))
  axm5:X∈CITY
END

Listing 4: Static aspect of simple TSP

MACHINE m0
 SEES c0
 VARIABLES current_city,visited
 INVARIANTS
  inv1:visited⊆CITY
  inv2:current_city∈CITY
 EVENTS
 INITIALISATION ≙
  act1: visited≔∅
  act2: current_city≔X
 END
 move≙
  ANY arrived
  WHEN
   grd1: arrived∈CITY
   grd2: current_city≠arrived
   grd3: current_city↦arrived∈complete_graph
   grd4: arrived∉visited
  THEN
   act1: current_city≔arrived
   act2: visited≔visited∪{arrived}
 END
END

Listing 5: Dynamic aspects of the simple TSP

3.3.3 Refined model

We propose a data refinement to switch from a set representation of the state of the machine *m0* (see Listing 6, part a) to a desired predictive representation (See Listing 6, part b). To achieve this, we have introduced the business concept *connection* (see Listing 7) which models the notion of a complete graph. This concept is modeled by a constant of the total function type whose starting set is the Cartesian product *CITY*×*CITY* and the ending set is imperatively BOOL. The axiom *axm_gluing* allows to logically link the *tsp_static* context to its abstract counterpart *c0*. Moreover, we have replaced the two set variables *current_city* and *visited* of total function type by the two predictive variables *position* and *already_visited* of total function type whose arrival set is of BOOL type. The INITIALISATION and *move* events are updated according to the new state of the *tsp_dynamic* machine (see Listing 8) and the gluing invariants (*inv1_gluing* and *inv2_gluing*).

The proof obligations associated with the model (*tsp_static*, *tsp_dynamic*) have been discharged by the proof tools of the Rodin platform (See Figure 1). This provides a formal proof for the correctness of the refinement relation between the two models (*c0*, *m0*) and (*tsp_static*, *tsp_dynamic*).

VARIABLES current_city,visited
INVARIANTS
 inv1: visited⊆CITY
 inv2: current_city ∈ CITY

Listing 6: Part a, Event-B set representation

VARIABLES position,already_visited
INVARIANTS
 inv1: position∈CITY→BOOL
 inv2: already_visited∈CITY→BOOL

Listing 6: Part b, Event-B predicative representation

CONTEXT tsp_static
EXTENDS c0
CONSTANTS connexion
AXIOMS
axm1: connexion∈CITY×CITY→BOOL
axm_gluing: ∀x,y·(x∈CITY∧y∈CITY∧connexion(y↦x)=TRUE⇒y↦x∈ complete_graph)
END

Listing 7: Business concept connexion

MACHINE tsp_dynamic
REFINES m0
SEES tsp_static
VARIABLES position,already_visited
INVARIANTS
inv1: position∈CITY→BOOL
inv2: already_visited∈CITY→BOOL
inv2_gluing: already_visited~[{TRUE}]=visited
inv1_gluing: ∀x·(x∈CITY∧position(x)=TRUE⇒x=current_city)
EVENTS
INITIALISATION ≙
act1: already_visited≔CITY×{FALSE}
act2: position≔(CITY×{FALSE})<+{X↦TRUE}
END
move≙
REFINES move
ANY start,arrived
WHEN
grd1_1: start∈CITY
grd1_2: arrived∈CITY
grd1_3: start≠arrived
grd1_4: position(start)=TRUE
grd1_5: already_visited(arrived)=FALSE
grd1_6: connexions(start↦arrived)=TRUE
THEN
act1: position≔position<+{start↦FALSE,arrived↦TRUE}
act2: already_visited(arrived)≔TRUE
END
END

Listing 8: Data refinement - predictive modeling of the simple TSP application

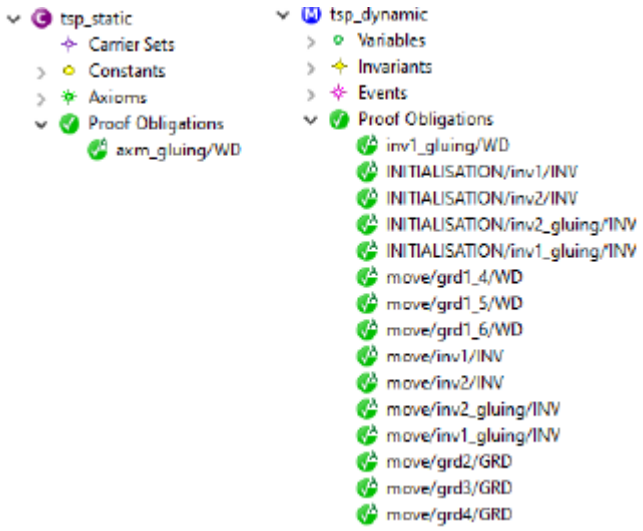

Figure 1: Proof Obligations associated with refined model

4. Case study: Sliding Puzzle

Sliding puzzle is a solitaire game in the form of a checkerboard created around 1870 in the United States by Sam Loyd. This problem consists of moving numbered tokens on an n × n grid to achieve a given configuration. The constraints imposed on displacements are as follows:

– A movement can be carried out horizontally or vertically (diagonal movements are prohibited).
– To move a token numbered t_i, the destination location on the grid must be empty.

5. Modeling in PDDL

In this section, we will present two PDDL models of the Sliding puzzle game case study. The first one comes from (Bibai, 2010). The second one is established by ourselves. Then, we will compare these two models and question the means offered by PDDL to describe **a rigorous relationship** between both models.

5.1 First modeling

The modeling proposed here is derived from (Bibai, 2010). It has a state described by three predicates: two dynamic and one static. Also, it has a single state change operator that allows browsing the state space associated with the Sliding puzzle game.

5.1.1 Domain construct

Listing 9 provides the **Domain** part of the Sliding puzzle game. The predicates *at* and *empty* are the dynamic predicates. They are modified by the state change operator *move*. While the predicate *neighbor* is static. It remains unchanged by the *move* action. The types construction introduces the types of objects used: *position* and *tile*. The predicate *at* allows to see if the object *?tile* of type *tile* is in the position (*?position*) of type *position*. The predicate *empty* models the notion of empty position. It allows to see if the object *?position* of type *position* is indeed the empty position. The predicate *neighbor* models the notion of neighbor with respect to two directions: horizontal and vertical. It allows to see if two objects *?p1*and *?p2* of type *position* are two neighbors.

```
(define (domain n-sliding-puzzle)
(:types position tile)
(:predicates (at ?position - position ?tile - tile) (neighbor ?p1 - position ?p2 - position) (empty ?position - position))
(:action move
  :parameters (?from ?to - position ?tile - tile)
  :precondition (and (neighbor ?from ?to) (at ?from ?tile) (empty ?to))
  :effect (and (at ?to ?tile) (empty ?from) (not (at ?from ?tile)) (not (empty ?to)))))
```

Listing 9: Domain construct of the first modeling

The move action allows three parameters: *?from*, *?to* of type position and *?tile* of type *tile*. Intuitively, move allows to move the *?tile* object from the *?from* position to the *?to* position. The formal before/after semantics of move are in two parts:

- a precondition (**precondition** clause) described by the connector **and** three elementary predicates, namely *neighbor*, *at* and *empty*. It states that the two objects *?from* and *?to* are neighbors via the predicate *neighbor*, the object *?to* is empty via the predicate *empty* and the object *?tile* is in position *?from* via the predicate *at*.
- a post-condition (clause **effect**) described by the connector **and**. It updates the current state: addition of atoms (*(at ?to ?tile)* and *(empty ?from)*) and deletion of atoms indicated by **not**.

5.1.2 Problem construct

Listing 10 provides an instance of the Sliding puzzle game. The **objects** clause introduces all the objects of type *position* and *tile* introduced in the domain part (See Listing 9). The **init** clause initializes the three predicates *at*, *empty* and *neighbor* describing the initial configuration. The **goal** clause describes in PDDL the final configuration.

```
(define (problem p-sliding-puzzle)
(:domain n-sliding-puzzle)
(:objects p_1_1 p_1_2 p_1_3 p_2_1 p_2_2 p_2_3  p_3_1 p_3_2 p_3_3 - positiont t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 - tile)
(:init
 (at p_1_1 t_4) (empty p_1_2) (at p_1_3 t_8) (at p_2_1 t_6) (at p_2_2 t_3) (at p_2_3 t_2) (at p_3_1 t_1) (at p_3_2 t_5)
 (at p_3_3 t_7) (neighbor p_1_1 p_1_2) (neighbor p_1_2 p_1_1) (neighbor p_1_2 p_1_3) (neighbor p_1_3 p_1_2)
 (neighbor p_2_1 p_2_2) (neighbor p_2_2 p_2_1) (neighbor p_2_2 p_2_3) (neighbor p_2_3 p_2_2) (neighbor p_3_1 p_3_2)
```

(neighbor p_3_2 p_3_1) (neighbor p_3_2 p_3_3) (neighbor p_3_3 p_3_2) (neighbor p_1_1 p_2_1) (neighbor p_2_1 p_1_1)
(neighbor p_1_2 p_2_2) (neighbor p_2_2 p_1_2) (neighbor p_1_3 p_2_3) (neighbor p_2_3 p_1_3) (neighbor p_2_1 p_3_1)
(neighbor p_3_1 p_2_1)(neighbor p_2_2 p_3_2) (neighbor p_3_2 p_2_2) (neighbor p_2_3 p_3_3) (neighbor p_3_3 p_2_3))
(**:goal** (and
(at p_1_1 t_1) (at p_1_2 t_2) (at p_1_3 t_3) (at p_2_1 t_4) (at p_2_2 t_5) (at p_2_3 t_6) (at p_3_1 t_7) (at p_3_2t_8))))

Listing 10: Problem construct of the first modeling

5.1.3 Plan-solution

We submitted the PDDL description of the Sliding puzzle game (Listing 9 and Listing 10) to the WEB PLANNER (Magnaguagno et al., 2017). This planner established a solution plan including a sequence, of length 25, of execution of the action *move*.

5.2 Second modeling

The *move* action from the previous modeling of the Sliding puzzle game (see Listing 9) is a non-deterministic action. It expresses a movement in a generic sense: left, right, up and down.

5.2.1 Domain construct

Here we propose a less abstract modeling (see Listing 11) than the one provided in Section 5.1. To achieve this, we decompose the *move* action of the first modeling into four elementary actions: *move_right*, *move_left*, *move_up* and *move_down*. In the same way, the predicate *neighbor* of the first modeling is broken down into four predicates: *neighbor_right*, *neighbor_left*, *neighbor_up* and *neighbor_down*. Moreover, we keep the introduced types (*position* and *tile*) and the two predicates *at* and *empty* introduced in the first modeling. Finally, the four actions *move_right*, *move_left*, *move_up* and *move_down* are strongly inspired by the *move* action of the first modeling.

(**define** (**domain** n-sliding-puzzle)
(**:types** position tile)
(**:predicates** (at ?position - position ?tile - tile) (empty ?position - position)
(neighbor_left ?p1 - position ?p2 - position) (neighbor_right ?p1 - position ?p2 - position)
(neighbor_up ?p1 - position ?p2 - position) (neighbor_down ?p1 - position ?p2 - position))
(**:action** move_left
:parameters (?from ?to - position ?tile - tile)
: precondition (and (neighbor_left ?from ?to) (at ?from ?tile) (empty ?to))
:effect (and (at ?to ?tile) (empty ?from) (not (at ?from ?tile)) (not (empty ?to))))
(**:action** move_right
:parameters (?from ?to - position ?tile - tile)
:precondition (and (neighbor_right ?from ?to) (at ?from ?tile) (empty ?to))
:effect (and (at ?to ?tile) (empty ?from) (not (at ?from ?tile)) (not (empty ?to))))
(**:action** move_up
:parameters (?from ?to - position ?tile - tile)
:precondition (and (neighbor_up ?from ?to) (at ?from ?tile) (empty ?to))
:effect (and (at ?to ?tile) (empty ?from) (not (at ?from ?tile)) (not (empty ?to))))
(**:action** move_down
:parameters (?from ?to - position ?tile - tile)
:precondition (and (neighbor_down ?from ?to) (at ?from ?tile) (empty ?to))
:effect (and (at ?to ?tile) (empty ?from) (not (at ?from ?tile)) (not (empty ?to)))))

Listing 11: Domain construct of the second modeling

5.2.2 Problem construct

Listing 12 provides the same instance of the Sliding puzzle game dealt with in the first modeling. It is almost identical to the one provided by the Listing 10 except the *neighbor* initialization replaced by *neighbor_left*, *neighbor_right*, *neighbor_up* and *neighbor_down*.

(**define** (**problem** p-sliding-puzzle)
(**:domain** n-sliding-puzzle)
(**:objects** p_1_1 p_1_2 p_1_3 p_2_1 p_2_2 p_2_3 p_3_1 p_3_2 p_3_3 – position t_1 t_2 t_3 t_4 t_5 t_6 t_7 t_8 – tile)

```
(:init (empty p_1_2)
  (at p_1_1 t_4) (at p_1_3 t_8) (at p_2_1 t_6) (at p_2_2 t_3) (at p_2_3 t_2) (at p_3_1 t_1) (at p_3_2 t_5) (at p_3_3 t_7)
  (neighbor_left p_1_1 p_1_2) (neighbor_right p_1_2 p_1_1) (neighbor_left p_1_2 p_1_3) (neighbor_right p_1_3 p_1_2)
  (neighbor_left p_2_1 p_2_2) (neighbor_right p_2_2 p_2_1) (neighbor_left p_2_2 p_2_3) (neighbor_right p_2_3 p_2_2)
  (neighbor_left p_3_1 p_3_2) (neighbor_right p_3_2 p_3_1) (neighbor_left p_3_2 p_3_3) (neighbor_right p_3_3 p_3_2)
  (neighbor_up p_1_1 p_2_1) (neighbor_down p_2_1 p_1_1) (neighbor_up p_1_2 p_2_2) (neighbor_down p_2_2 p_1_2)
  (neighbor_up p_1_3 p_2_3) (neighbor_down p_2_3 p_1_3) (neighbor_up p_2_1 p_3_1) (neighbor_down p_3_1 p_2_1)
  (neighbor_up p_2_2 p_3_2) (neighbor_down p_3_2 p_2_2) (neighbor_up p_2_3 p_3_3) (neighbor_down p_3_3 p_2_3))
(:goal (and
     (at p_1_1 t_1) (at p_1_2 t_2) (at p_1_3 t_3) (at p_2_1 t_4) (at p_2_2 t_5) (at p_2_3 t_6) (at p_3_1 t_7) (at p_3_2 t_8))))
```

Listing 12: Problem construct of the second modeling

5.2.3 Plan-solution

We submitted the PDDL description of the second modeling of the Sliding puzzle game (Listing 11 and Listing 12) to the same planner used for the first modeling: WEB PLANNER. The generated solution plan includes the same number of trips (here 25) as the one provided in Section 5.1.3. Such a plan-solution **explains** the direction of the legal movements of the tiles via the names of the operators executed, i.e. *move_right*, *move_left*, *move_up* and *move_down*.

5.3 Comparison

Intuitively, the second model (see Listing 11) **refines** the first model (See Listing 9). **The states** of two models are equivalent. The union of four predicates *neighbor_left*, *neighbor_right*, *neighbor_up* and *neighbor_down* is equal to *neighbor*. And the other two predicates *at* and *empty* remain the same. The four state change **operators** *move_left*, *move_right*, *move_up* and *move_down* of the second model are derived from the generic operator *move* of the first model. But the PDDL language cannot **formally verify** the refinement relationship between the two models.

5.4 Evaluation

The PDDL language is used to describe the states and actions of a planning problem. However, not all the semantic aspects related to the formalization of a state of a planning problem are easy to describe in PDDL. This concerns the intra-atomic and inter-atomic semantic properties. For example, in the planning problem of the Sliding puzzle game treated previously, we want to attach the following property to the predicate *empty*: at any state, one and only one position is empty. Thus, the actions are specified in a totally separate way. Moreover, the basic properties (mainly typing) related to the entities of a planning problem are insufficient to judge the coherence of a PDDL description. More or less elaborated properties should be attached to static and dynamic predicates in particular. This makes it possible to check the consistency of the initial state, the goal state and the actions.

6. Modeling in Event-B

The Event-B method supports the paradigm **correct by construction** (Abrial, 2010). It allows incremental development through successive refinements with mathematical proof. In this section, we will use Event-B for the application of the Sliding puzzle game. First of all, we will propose a refinement strategy for the said application going from a set representation to a predicative representation, at the end, translatable into PDDL. Then, we will develop the adopted strategy step by step.

6.1 The adopted refinement strategy

The refinement strategy adopted for the Sliding puzzle game includes:

1. Initial abstract model. Such a model is intuitively equivalent, in terms of operational semantics, to the first PDDL model introduced in Section 5.1. However, this model is simple to understand and has mathematically proven invariant and vivacity properties.

2. First refinement. Consideration of four directions. This first refinement will give rise to a model that is semantically equivalent to the PDDL model introduced in Section 5.2. However, the refinement relationship between the two abstract and refined models is proven mathematically within the Rodin platform supporting Event-B.

3. Second refinement. Location of the empty square in the grid. The objective of this step is to remove the non-determinism related to the parameters introduced in the ANY clause of each change of state operator. Technically, the WITH clause is used to obtain events without parameters.

4. Data refinement. From a set representation to a predicative representation. This is a refinement of data to move from a set representation of the state of a planning problem to a predictive representation that can be translated into a PDDL using simple rules. To facilitate the correction of this data refinement (discharge of Obligations of Proof), we have introduced several intermediate models based on hierarchical refinement schemes. The ultimate Event-B model obtained includes a state (Event-B constants and variables) fully described using a predicative representation (See Section 3.2).

6.2 Initial Abstract Model

6.2.1 Presentation

The 9 square (3×3) Sliding puzzle game is modeled in Event-B by a context called *problem* and a machine called *T1* (See Listing 13). The state of this game is modeled by the variable *grid* of bijective function type whose starting set is a Cartesian product 1..3×1..3 and the ending set is from 0..8. Both initial and goal states are considered as two constants *initial_state* and *goal_state* introduced in the *problem* context. The *T1* machine contains two events INITIALISATION and *goal*, respectively, to initialize *grid* to *initial_state* and to see if *grid* is a goal state, i.e. coincides with *goal_sate*. The *move* event allows calculating all the following situations of the Sliding puzzle game starting from the current situation in *grid*. It is a non-deterministic event with four local parameters:

– *row* and *column* enabling to locate the empty square in grid (see *grd6*, *move* event, machine *T1*, Listing 13),
– *r* and *c* allowing to designate the chosen neighbor of the empty square in the grid.

CONTEXT problem
CONSTANTS initial_sate,goal_state
AXIOMS
axm1: initial_state∈1..3×1..3⇸⇸0..8
axm2: goal_state∈1..3×1..3⇸⇸0..8
axm3: initial_state={1↦1↦4,1↦2↦0,1↦3↦8,2↦1↦6,2↦2↦3,2↦3↦2,3↦1↦1,3↦2↦5,3↦3↦7}
axm4: goal_state={1↦1↦1,1↦2↦2,1↦3↦3,2↦1↦4,2↦2↦5,2↦3↦6,3↦1↦7,3↦2↦8,3↦3↦0}
END

MACHINE T1
SEES problem
VARIABLES grid
INVARIANTS
inv1: grid∈1..3×1..3⇸⇸0..8
DLF: ∃r,c,row,column·(r∈1..3∧c∈1..3∧row∈1..3∧column∈1..3∧((r=row+1∧c=column)∨(r=row−1∧c=column)∨
(r=row∧c=column+1∨(r=row∧c= column−1))∧grid(row↦column)=0) Theorem
EVENTS
INITIALISATION ≙
act1: grid:∈1..3×1..3⇸⇸0..8
END
move ≙
ANY r,c,row,column
WHEN
grd1: r∈1..3
grd2: c∈1..3
grd3: row∈1..3
grd4: column∈1..3
grd5: (r=row+1∧c=column)∨(r=row−1∧c=column)∨ (r=row∧c=column+1)∨(r=row∧c=column−1)
grd6: grid(row↦column)=0
THEN
act1: grid:=grid<+{row↦column↦grid(r↦c),r↦c↦grid(row↦column)}
END
goal ≙
WHEN
grd1: grid=goal_state
THEN
skip
END
END

Listing 13: Initial abstract model of the Sliding puzzle game

The *grd5* guard of the *move* event models the four potential neighbors of the empty square (*row*, *column*) of the grid *grid*. Such a predicate can have at most four solutions. The effect of the *move* event is modeled by the *act1* action. This action exchanges the contents of two cells (*row*, *column*) and (*r*, *c*) of the grid *grid*. The square designated by (*r*, *c*) is the chosen square if there are several solutions. The *T1* machine is correct with respect to the Event-B theory concerning the respect of the invariance properties. In fact, the INITIALISATION event establish the invariant (*inv1*) and the *move* event preserves the invariant (*inv1*) of this machine. In addition, the obligation of proof DLF of theorem type (THM) is discharged by the proofs of the Rodin platform. This ensures that the *T1* machine is not blocked. Such a liveliness property ensures that the *move* event is always applicable on the current state provided by the grid.

6.2.2 Discussion

Intuitively, the first PDDL model (see Listing 9) and the initial abstract model in Event-B (see Listing 13) are behaviorally equivalent. Indeed, these two models produce the same movements (or transitions) labelled by the action *move*. However, these two models have two different states. The Event-B model proposes a state with a single variable *grid* of bijective function type. This variable naturally translates the *grid* as a two-dimensional matrix. In addition, the *grid*'s bijective character ensures the non-redundancy of the tiles and one and only one empty square. Finally, the notion of neighborhood of the empty square is naturally modeled by reasoning in relation to the empty square (See *grd5* of the *mov*e event, Listing 13). On the other hand, the PDDL model contains a state with three variables of a predicate type *at*, *neighbor* and *empty* (See Listing 9). Such variables respectively **codify** the content of the grid, the potential neighbors and the empty square. The properties of the *grid* variable of the Event-B model cannot be deduced logically from the variables forming the state of the PDDL model. In contradiction to the *move* action of the PDDL model, the *move* event of the Event-B model is considered **formally correct** - by mathematical proof - with respect to the invariant properties of the Event-B model. Similarly, **the absence of the blockage** of the Event-B model is mathematically proven. This is not possible for the PDDL model. In the Event-B model, it is certain **a priori** that its state space contains **valid** states and does not contain states without successors. This is not necessarily true for the PDDL model.

6.3 First refinement: consideration of the four direction

6.3.1 Presentation

The *move* event of the initial abstract model can move the empty playing square of the Sliding puzzle in one of four directions: left, right, up or down. This is modeled by the *grd5* guard of the *move* event (See Listing 13). In this first refinement, we will **explain** the direction of each move made on the grid of Sliding puzzle game. To do this, we decompose the *move* event into four events *move_left*, *move_right*, *move_up* and *move_down* according to the *grd5* guard of the abstract event *move* (See Figure 2).

The *T2* machine (see Listing 14) is considered to be correct. Indeed, the refinement relationship between *T1* (see Listing 13) and *T2* is proven correct by discharging proof obligations associated with *T2*. Like the *T1* machine, the *T2* machine is non-blocking thanks to the *DLF* theorem (See Listing 14).

MACHINE T2
REFINES T1
SEES problem
VARIABLES grid
INVARIANTS
inv1: grid∈1‥3×1‥3⇸0‥8
DLF: ∃r,c,row,column·(r∈1‥3∧c∈1‥3∧row∈1‥3∧column∈1‥3∧((r=row+1∧c=column∧grid(row↦column)=0)∨(r=row−1∧
c=column∧grid(row↦column)=0)∨(r=row∧c=column+1∧grid(row↦column)=0)∨(r=row∧c=column−1∧
grid(row↦column)=0))) Theorem
EVENTS
INITIALISATION ≙
act1: grid:=initial_state
END
move_left ≙
REFFINE move
ANY r,c,row,column
WHEN
grd1: r∈1..3
grd2: c∈1..3
grd3: row∈1..3
grd4: column∈1..3

```
 grd5: r=row∧c=column−1
 grd6: grid(row↦column)=0
 THEN
 act1: grid≔grid<+{row↦column↦grid(r↦c),r↦c↦grid(row↦column)}
 END
move_right ≙
REFFINE move
ANY r,c,row,column
WHEN
 grd1: r∈1..3
 grd2: c∈1..3
 grd3: row∈1..3
 grd4: column∈1..3
 grd5: r=row∧c=column+1
 grd6: grid(row↦column)=0
 THEN
 act1: grid≔grid<+{row↦column↦grid(r↦c),r↦c↦grid(row↦column)}
 END
move_up ≙
REFFINE move
ANY r,c,row,column
WHEN
 grd1: r∈1..3
 grd2: c∈1..3
 grd3: row∈1..3
 grd4: column∈1..3
 grd5: r=row-1∧c=column
 grd6: grid(row↦column)=0
 THEN
 act1: grid≔grid<+{row↦column↦grid(r↦c),r↦c↦grid(row↦column)}
 END
move_down ≙
REFFINE move
ANY r,c,row,column
WHEN
 grd1: r∈1..3
 grd2: c∈1..3
 grd3: row∈1..3
 grd4: column∈1..3
 grd5: r=row+1∧c=column
 grd6: grid(row↦column)=0
 THEN
 act1: grid≔grid<+{row↦column↦grid(r↦c),r↦c↦grid(row↦column)}
 END
goal ≙
 extended
WHEN
 grd1: grid=goal_state
THEN
 skip
END
END
```

Listing 14: First refinement-four directions explained

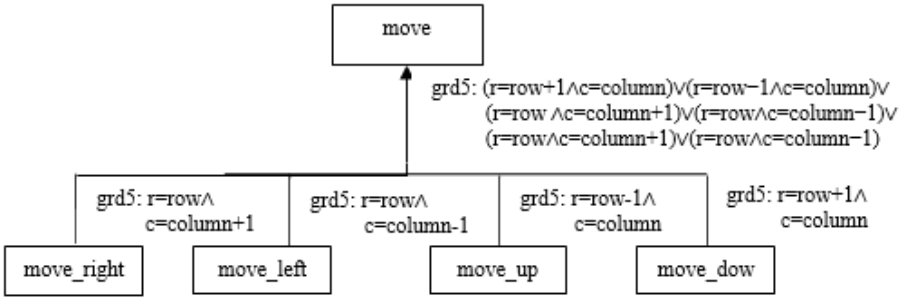

Figure 2: Decomposition of the abstract event move

6.3.2 Discussion

Intuitively, the second PDDL model (see Listing 11) and the refined *T2* model (see Listing 14) are behaviorally equivalent. However, *T2* refined model is formally obtained starting from *T1* thanks to the semantics of the refinement relation (REFINES) in Event-B. Thus, *T2* keeps the "good" properties of *T1* (See Section 6.2.1). On the other hand, there is no rigorous formal relationship between the two PDDL models: first model (see Listing 9) and second model (See Listing 11).

6.4 Second refinement: location of the empty square in the grid

Due to the bijective nature of the *grid* variable coming from the *T2* ascending machine, the empty square is designated by a single pair *(r_v, c_v)∈1..3×1..3*.Thus, the two local parameters *row* and *column* of four events *move_right*, *move_left*, *move_up* and *move_down* can be replaced respectively by the two state variables *r_v* and *c_v* introduced by the refined machine *T3* (See Listing 15, the WITH clause of the four events). The *inv*3 invariant of the *T3* machine always guarantees that the empty square is indeed identified by (*r_v, c_v).* Similarly, the two other local parameters *r* and *c* designating the target square of the four events of *T2* are replaced by two expressions using *r_v* and *c_v* respectively (See the WITH clause of the relevant events in Listing 15).

The obligations of proof associated with the *T3* machine are proven. This formally verifies that *T3* refines *T2*. In addition, the *DLF* considered as *T3*'s theorem is discharged. Thus, like *T1* and *T2*, *T3* is non-blocking.

MACHINE T3
REFINES T2
SEES problem
VARIABLES grid,r_v,c_v
INVARIANTS
inv1: r_v∈1..3
inv2: c_v∈1..3
inv3: grid(r_v↦c_v)=0
DLF: (c_v-1∈1..3)∨(c_v+1∈1..3)∨(r_v1∈1..3)∨(r_v+1∈1..3) Theorem
EVENTS
INITIALISATION ≙
act1: grid:=initial_state
act2: r_v:=2
act3: c_v :=2
END
move_left ≙
REFFINE move_left
WHEN
grd1: c_v-1∈1..3
WITH
row: row=r_v
column: column=c_v
r: r=r_v
c: c=c_v-1
THEN
act1: grid≔grid<+{r_v↦c_v↦grid(r_v↦c_v−1),r_v↦c_v−1↦grid(r_v↦c_v)}
act2: c_v≔c_v−1
END
move_right ≙

REFFINE move_right
WHEN
grd1: c_v+1∈1..3
WITH
row: row=r_v
column: column=c_v
r: r=r_v
c: c=c_v+1
THEN
act1: grid≔grid<+{r_v↦c_v↦grid(r_v↦c_v+1),r_v↦c_v+1↦grid(r_v↦c_v)}
act2: c_v≔c_v−1
END
move_up ≙
REFFINE move_up
WHEN
grd1: r_v-1∈1..3
WITH
row: row=r_v
column: column=c_v
r: r=r_v-1
c: c=c_v
THEN
act1: grid≔grid<+{r_v↦c_v↦grid(r_v-1↦c_v),r_v-1↦c_v↦grid(r_v↦c_v)}
act2: r_v≔r_v−1
END
move_down ≙
REFFINE move_down
WHEN
grd1: r_v+1∈1..3
WITH
row: row=r_v
column: column=c_v
r: r=r_v+1
c: c=c_v
THEN
act1: grid≔grid<+{r_v↦c_v↦grid(r_v+1↦c_v),r_v+1↦c_v↦grid(r_v↦c_v)}
act2: r_v≔r_v+1
END
goal ≙
extended
WHEN
grd1: grid=goal_state
THEN
skip
END
END

Listing 15: Locating the empty square in the grid

6.5 Data refinement

The proposed data refinements are intended to move from the set representation of the *T3* machine state (see Listing 16, part a) to the desired predicative representation (See Listing 16, part b). To achieve this, we have proposed the following data refinement steps:

Step 1: Introduction of two business concepts: *POSITION* and *TILE*. These two concepts are modeled by two sets defined by extension. In the same way, we have modeled as predictive constants the left, right, top and bottom neighbors.

Step 2: Refinement of a variable of type a bijective function to a total function type by keeping the same signature.

Step 3: Refinement of a set- variable of type a total function to a predictive variable of type a total function with a BOOL type input set.

Step 4: Introduction of two predicative variables *at* and *empty* (See Listing 16, part b).

Each data refinement step contains an abstract level, a refined level and a gluing invariant. The latter allows to logically linking the refined level to its abstract counterpart. The formal correction of the refinement relationship between the two levels (abstract and refined level) is based on the intuition of the proposed gluing invariant. Technically, the data refinement steps proposed above remove abstract variables and introduce new ones. Similarly, abstract events are refined according to the new variables introduced and the recommended gluing invariant.

VARIABLES grid, r_v, c_v
INVARIANTS I
inv1: grid∈1‥3×1‥3⇸⇸0‥8
inv2: r_v∈1‥3
inv3: c_v∈1‥3
inv4: grid(r_v↦c_v)=0

Listing 16: Part a, Set representation

VARIABLES at,empty
NVARIANTS
inv1: at∈POSITION×(TILE\{t0})→BOOL
inv2: empty∈POSITION→BOOL

Listing 16: Part b, Ultimate predicative Representation

6.6 The ultimate Event-B model

The ultimate Event-B model obtained is given by the Listing 17 by keeping only the typing properties of the variables forming the state of this model, namely *at* and *empty*. Such a model has two contexts *position_contex* and *neighbor* and a *T8* machine. The context *position_contex* locates the definition of two business concepts *POSITION* and *TILE* which are considered as two sets defined by extension. As for the *neighbor* context, it locates the static predicates modeling respectively the left, right, top and bottom neighbors. Finally, the *T8* machine contains a state formed by predictive variables and state change operators. The ultimate flattened Event-B model is systematically translatable to a PDDL model.

MACHINE T8
REFINES T7
SEES position_context, neighbor
VARIABLES at,empty
I**NVARIANTS**
inv1: at∈POSITION×(TILE\{t0})→BOOL
inv1: empty∈POSITION→BOOL
DLF: (∃from,to,tile·(from∈POSITION∧to∈POSITION∧empty(to)=TRUE∧tile∈(TILE\{t0})∧at(from↦tile)=TRUE∧neighbor_left(to↦from)=TRUE))∨(∃from,to,tile·(from∈POSITION∧to∈POSITION∧empty(to)=TRUE∧tile∈(TILE\{t0})∧at(from↦tile)=TRUE∧neighbor_right(to↦from)=TRUE))∨(∃from,to,tile·(from∈POSITION∧to∈POSITION∧empty(to)=TRUE∧tile∈(TILE\{t0})∧at(from↦tile)=TRUE∧neighbor_up(to↦from)=TRUE))∨(∃from,to,tile·(from∈POSITION∧to∈POSITION∧empty(to)=TRUE∧tile∈(TILE\{t0})∧at(from↦tile)=TRUE∧neighbor_right(to↦from)=TRUE))
EVENTS
INITIALISATION ≙
act1: at:=((POSITION×(TILE\{t0}))×{FALSE})<+({p_1_1↦t1,p_1_2↦t4,p_1_3↦t5,p_2_1↦t6,p_2_3↦t3,p_3_1↦t7, p_3_2↦t8,p_3_3↦t2}×{TRUE})
act2: empty:=(POSITION×{FALSE})<+({p_2_2}×{TRUE})
END
move_left ≙
REFFINE move_left
ANY from,tile,to
WHEN
grd1: from∈POSITION∧to∈POSITION
grd2: empty(to)=TRUE∧tile∈(TILE\{t0})
grd3: neighbor_left(to↦from)=TRUE
grd4: at(from↦tile)=TRUE
THEN
act1: at:=at<+{(to↦tile)↦TRUE,(from↦til)↦FALSE}
act2: empty:=empty<+{to↦FALSE,from↦TRUE}
END
move_right ≙
REFFINE move_right
ANY from,tile,to
WHEN
grd1: from∈POSITION∧to∈POSITION
grd2: empty(to)=TRUE∧tile∈(TILE\{t0})
grd3: neighbor_right(to↦from)=TRUE

```
 grd4: at(from↦tile)=TRUE
THEN
 act1: at:=at<+{(to↦tile)↦TRUE,(from↦til)↦FALSE}
 act2: empty:=empty<+{to↦FALSE,from↦TRUE}
END
move_up ≙
REFFINE move_up
ANY from,tile,to
WHEN
 grd1: from∈POSITION∧to∈POSITION
 grd2: empty(to)=TRUE∧tile∈(TILE\{t0})
 grd3: neighbor_up(to↦from)=TRUE
 grd4: at(from↦tile)=TRUE
THEN
 act1: at:=at<+{(to↦tile)↦TRUE,(from↦til)↦FALSE}
 act2: empty:=empty<+{to↦FALSE,from↦TRUE}
END
move_down ≙
REFFINE move_down
ANY from,tile,to
WHEN
 grd1: from∈POSITION∧to∈POSITION
 grd2: empty(to)=TRUE∧tile∈(TILE\{t0})
 grd3: neighbor_down(to↦from)=TRUE
 grd4: at(from↦tile)=TRUE
THEN
 act1: at:=at<+{(to↦tile)↦TRUE,(from↦til)↦FALSE}
 act2: empty:=empty<+{to↦FALSE,from↦TRUE}
END
goal ≙
REFFINE goal
WHEN
 grd1: at=((POSITION×(TILE\{t0}))×{FALSE})<+({p_1_1↦t1,p_1_2↦t2,p_1_3↦t3,p_2_1↦t4,p_2_2↦t5,p_2_3↦t6,
        p_3_1↦t7,p_3_2↦t8}×{TRUE})
 grd2: empty=(POSITION×{FALSE})<+({p_3_3}×{TRUE})
THEN
 skip
END
END
```

Listing 17: The ultimate Event-B model flattened of Sliding puzzle game

In our previous works, we established an approach to translating from Event-B to PDDL and developed an exploratory Event-B2PDDL prototype. The PDDL model of the Sliding puzzle game from the ultimate flattened Event-B model is identical to the second PDDL model introduced in Section 5.2. However, we underline a major advantage in favor of the PDDL model from the ultimate Event-B model: it is correct by construction under the assumption that the translation from Event-B to PDDL is correct. Given that the ultimate Event-B model is based on a subset of Event-B that is very close to the PDDL language, this assumption is rather realistic.

6.7 Proof

In Event-B, the correction of Event-B models is based on correction criteria formalized by Proof Obligations (POs). Such POs must be discharged by the automatic and interactive provers of the Rodin platform. Table 1 summarizes the POs linked to the different Event-B models of the Sliding puzzle game application. We have used the external provers of the RODIN platform, in particular the SMT provers and the ProB (Krings et al., 2015) counter-prover, in order to discharge the POs in an interactive way. Moreover, we introduced lemmas in order to discharge the *DLF* (DeadLock Free) theorems associated to the different machines. Thus, all the machines forming the Sliding puzzle application are non-blocking. This allows access to all the attainable states of the state space related to the Sliding puzzle game. The cyclic character of the Sliding puzzle state space (for example, *move_left* followed by *move_right*) fully justifies the absence of blocking of Event-B models with Sliding puzzle game. The static predicates *neighbor_left*, *neighbor_right*, *neighbor_up* and *neighbor_down* are modeled by Event-B constants. The initialization of these constants is the responsibility of the modeler. In order to reduce the risk of error, we have established properties relative to these

constants considered as theorems. In PDDL, the initialization of static predicates (see Listing 10 and Listing 12) is a delicate and tedious problem leading to errors that are difficult to detect.

Machine	POs	Automatic POs	Interactive POs	not discharged POs
T1	6	1	5	0
T2	10	4	6	0
T3	45	36	9	0
T4	86	19	67	0
T5	16	1	15	0
T6	29	10	19	0
T7	54	8	46	0
T8	42	9	33	0

Table 1: POs associated to Event-B models of Sliding puzzle game

6.8 Animation

The ProB tool (Krings et al., 2015) allows animating Event-B models proven by the Rodin platform proofs. Such an animation is considered as a certain "execution" of the Event-B models. Indeed, the ProB animator is equipped with a constraint solver capable of establishing solutions for event guards. The triggering of a triggerable event chosen by the modeler leads to the execution of the action associated with this event by ProB. Knowing that a triggerable event is an event whose guard is satisfied. Thus, we have successfully used ProB's animator on the different models of the Sliding puzzle game application, from the initial abstract model (*T1*) to the ultimate model (*T8*) through the intermediate models *T2*, *T3*, *T4*, *T5*, *T6* and *T7*. We replayed the same scenarios on these different models.

7. Process Combining Event-B and PDDL for Automatic Planning

Here, we advocate a process of developing planning problems by combining Event-B and PDDL. The Event-B method is used to develop Event-B models for planning problems. Such models are correct by construction and valid using the proof and validation tools offered by the Rodin platform supporting Event-B. The ultimate model from the Event-B refinement chain is translatable into PDDL. The planners associated with PDDL are used to automatically produce plan-solutions related to the planning problems initially described by Event-B.

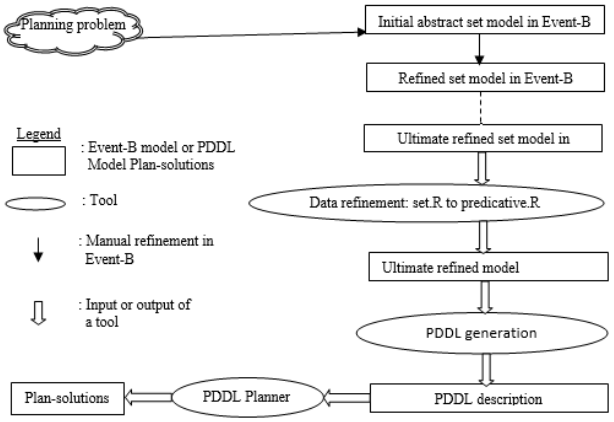

Figure 3: Process coupling Event-B and PDDL

The development process coupling Event-B and PDDL to specify and solve planning problems is described in Figure 3. Such a process combines manual refinement and automatic refinement in Event-B. The manual refinement ultimately produces the ultimate set-model. Depending on the planning problems, this manual refinement could involve, in our experience, the decomposition of an event (one to many see Section 6.3), the reinforcement of guards and the enrichment of the state (addition of variables and/or invariant properties). As for automatic refinement, it concerns the refinement of data allowing going from a set representation to a predicative representation. In particular, the automatic data refinement tool shall solve the problem of rewriting events following the introduction of a predictive variable and the gluing invariant. For reasons related to the automatic discharging of

proof obligations, this tool must work step by step taking into account the types of variables and set -constants. Our development process coupling Event-B and PDDL provides a tool for generating PDDL code from the ultimate refined Event-B predicative model. We have already developed an exploratory MDE Event-B2PDDL prototype. We intend to revisit this prototype in order to make it a software engineering tool worthy of its name. To achieve this, it is necessary to identify and justify the subset of Event-B translatable into PDDL. Like the formal method B which proposes a subset of B called B0 translatable in an imperative programming language like C and Ada, the subset of Event-B translatable in PDDL will be called Event-B0. This is being finalized. In addition, Event-B0 is designed to allow the bilateral passage between Event-B and PDDL. The transformation of PDDL to Event-B0 (and therefore Event-B) promotes formal verification of PDDL descriptions. Finally, our Event-B and PDDL coupling process is reusing known and recognized tools in the field of automatic planning, namely planners accepting PDDL descriptions.

8. The Event-B2PDDL tool

Our Event-B2PDDL tool takes as input a reduced Event-B model that is translatable into PDDL and outputs a PDDL description acceptable to planners. Event-B2PDDL is based on simple intuitive rules allowing the systematic translation of Event-B elements to PDDL elements. Event-B2PDDL is made according to MDE technology.

8.1 Event-B to PDDL transformation rules

The PDDL description from the Event-B2PDDL tool has two domain and problem constructions (see section 2.2). Thus, in our previous works, we have respectively established rules allowing the translation of Event-B elements related to the planning domain and the planning problem. The translation rules for Event-B elements related to the planning domain concern: the translation of abstract sets, constants, Boolean constants or variables, Boolean functions, events and formulas. As an example, Table 2 groups together the translations of Event-B formulas retained in PDDL.

Event-B formulas	**PDDL formulas**
P ∧ Q	(and P Q)
P ∨ Q	(or P Q)
P ⇒ Q	(imply P Q)
¬P	(not P)
∀z.P⇒ Q	(forall (?z) (imply P Q))
∃z.P∧ Q	(exists (?z) (and P Q))
E = F	(= E F)
E /= F	(not(= E F))
b:=TRUE	(b)
b:=FALSE	(not(b))
f(x):=TRUE	(f ?x)
f(x):=FALSE	(not(f ?x))
f:=f <+{ x|->TRUE, y|->FALSE }	(and (f ?x) (not(f ?y)))

Table 2: Translation of Event-B formulas retained in PDDL

The rules for conveying Event-B elements related to the planning problem concern the translation of the constants linked to the sets defined by enumeration and the translation of two INITIALISATION and GOAL events. For example, Table 3 gives the translation scheme of the Event-B constants for enumeration of the elements of a set.

Event-B	**PDDL**
CONSTANTS cst1, cst2, …, cstn AXIOMS axm1 : partition(TYPE, {cst1}, {cst2}, …, {cstn})	(:objects cst1 cst2 … cstn -TYPE)

Table 3: Schemes of translations of constants

8.2 Translation automation from Event-B to PDDL

Using both MDE Xtext and Xtend tools, we developed the Event-B2PDDL tool based on the simple transformation rules. The Xtext tool allowed us to design a DSL for our input language: A reduced Event-B with only those constructions that are taken into account by the transformation. Transformation and generation of PDDL code is programmed in Xtend.

9. Related Work

There exist several Integrated Development Environments (IDE) supporting PDDL exist (Magnaguagno et al., 2017; Strobel and Kirsch, 2014; Muise and Lipovetzky, 2020). Such environments provide functionality for editing PDDL descriptions (both Domain and Problem parts), lexical-syntax checking of PDDL text, generating plans using a planner that accepts PDDL descriptions, and viewing the state space associated with the planning problem described by PDDL. The latter functionality allows, among other things, to provide information to the user related to the "**execution**" of PDDL actions. This allows the user to detect errors related to the specification of a PDDL action such as incorrect precondition, incorrect post-condition and incorrect precondition and post-condition (Tapia et al., 2015). In addition, the visualization of the state space makes it possible to explain the behavior of the PDDL planner in order to find a solution plan. This reduces the opacity of the PDDL planners. In fact, these planners are used as black boxes when there is no visualization.

The automatic planning community has developed many planners that accept PDDL (McDermott et. al., 1998) descriptions. These software tools, based on state space and planning graph search and SAT solvers, are rarely or no longer usable by other software platforms such as robotic architectures, web architectures and software engineering architectures. The PDDL4J (Pellier and Fiorino, 2017) toolkit written in Java is factorizing techniques from automatic planning: planning algorithms, planning heuristics, a number of planners, and the syntactic and semantic facilities of planning domain description languages such as PDDL.

The correction of plan-solutions generated by PDDL planners is entrusted to validators (see Section 2.4), but these as complex software are **not certified**. The work described in (Abdulaziz and Lammich, 2018) recommends the use of the proof assistant Isabelle/HOL to formally develop **a certified validator**. To achieve this, it formalizes the PDDL language in HOL.

In more or less completed jobs, automatic scheduling is seen as a model checking problem. For example, the work described in (Hörne and van der Poll, 2008) explores the use of two model-checkers ProB and NuSMV for modeling and solving planning problems. It empirically compares these two model-checkers on five planning problems described by B (for ProB) and BDD (Binary Decision Diagrams) for NuSMV. Similarly, the work described in (Li et al.., 2012) proposes an approach for translating PDDL to CSP# in order to use the PAT model-checker in the planning domain.

Unlike the B-method, the Event-B-method does not have a standard code generator. Indeed, the ultimate Event-B model depends on the targeted code: sequential, concurrent, distributed, etc. The work described in (Méry and Kumar Singh, 2011) allows Event-B to be translated into imperative languages that support sequential programming. The work described in (Siala et al., 2016) provides an approach for translating Event-B to BIP that supports distributed programming.

In general, even a confirmed modeler has a difficulty in carrying out a refinement process based on mathematical proof. To cope with these difficulties, automatic refinement is recommended. The BART tool (Requet, 2008) is used to assist B modelers in the refinement process, especially for B models close to the B0 language. The work described in (Siala et al., 2016) proposes a process allowing the formal decomposition of a centralized Event-B specification. Such a process combines manual and automatic refinement.

In this work, we intend to profitably use the automatic refinement technique and the generation of PDDL code from a **correct by construction** Event-B ultimate model containing only predictive constants and variables (See Section 7).

10. Conclusions

The planning community has developed languages – in this case PDDL, planners and validators for the description of planning problems and the generation and validation of plan-solutions. However, the reliability of PDDL descriptions is dealt with **a posteriori**. In this work, we proposed a process to combine Event-B and PDDL for the development of planning problems. The formal Event-B method is used **upstream** and PDDL is generated from the ultimate Event-B model, which is the result of a chain of Event-B models linked by the formal refinement relationship in the sense of the Event-B theory. Our process coupling Event-B and PDDL has been successfully applied to the Sliding puzzle game. Currently, we are further equipping our process in accordance with the requirements described in Section 7.

References

Abrial, J. R., 2010. Modeling in Event-B: Systems and software engineering. Cambridge University Press, New York, USA.

Abdulaziz, M., Lammich, P., 2018. A formally verified validator for classical planning problems and solutions. IEEE 30th International Conference on Tools with Artificial Intelligence, https://doi.org/10.1017/CBO9781139195881.

Bibai, J., 2010. Segmentation et Evolution pour la Planification : Le Système DivideAndEvolve. THALES Research and Technology France, Paris Sud University.

Ghallab, M., Nau, D., Traverso, P., 2004. Automated planning: theory & practice. Elsevier.

Howey, R., Long, D., Fox, M., 2004. VAL: Automatic Plan Validation, Continuous Effects and Mixed Initiative Planning using PDDL. In Tools with Artificial Intelligence, ICTAI, https://doi.org/10.1109/ICTAI.2004.120.

Hörne, T., van der Poll, J.A., 2008. Planning as model checking: the performance of ProB vs NuSMV. In: Proceedings of the 2008 annual research conference of the South African Institute of Computer Scientists and Information Technologists on IT research in developing countries: riding the wave of technology.

Haslum, P., Lipovetzky, N., Magazzeni, D., Muise, C., 2019. An introduction to the planning domain definition language. In: Synthesis Lectures on Artificial Intelligence and Machine Learning, https://doi.org/10.2200/S00900ED2V01Y201902AIM042.

Krings, S., Bendisposto, J., Leuschel, M., 2015. Turning failure into proof: The ProB disprover for B and Event-B. In: Proceedings 13th International Conference on Software Engineering and Formal Methods (SEFM 2015), Springer LNCS.

Li., Y., Sun, J., Song Dong, J., Liu, Y., Sun, J., 2012. Translating PDDL into CSP# - the PAT Approach, in: Proceedings of the 17th IEEE International Conference on Engineering of Complex Computer Systems.

McDermott, D., Ghallab, M., Howe, A., Knoblock, C., Ram, A., Veloso, M., Weld, D., Wilkins, D., 1998. PDDL-The Planning Domain Definition Language. Yale Center for Computational Vision and Control, Technical Report CVC TR- 98-003/DCS TR-1165, USA.

Méry, D., Kumar Singh, N., 2011. Automatic code generation from Event-b models. SoICT 2011:179-188, https://doi.org/10.1145/2069216.2069252.

Magnaguagno, M. C., Pereira, R. F., More, M. D., Meneguzzi, F., 2017. WEB PLANNER: A tool to develop classical planning domains and visualize heuristic state-Space search. ICAPS, User Interfaces for Scheduling & Planning (UISP) Workshop.

Muise, C., Lipovetzky, N., 2020. KEPS Book: Planning.Domains. In Knowledge Engineering Tools and Techniques for AI Planning. Springer. 91–105.

Pellier, D., Fiorino, H., 2017. PDDL4J: a planning domain description library for java. Journal of Experimental & Theoretical Artificial Intelligence, https://doi.org/10.1080/0952813X.2017.1409278.

Requet, A., 2008. BART: A tool for automatic refinement. In ABZ conference, https://doi.org/10.1007/978-3-540-87603-8_33.

Roberts, M., Howe, A., 2009. Learning from planner performance. Elsevier, Artificial Intelligence, 173(5-6): 536-561.

Strobel, V., Kirsch, A., 2014. Planning in the Wild: Modeling tools for PDDL. 37th German Conference on Artificial Intelligence, Stuttgart. Germany, ffhal-01691693f.

Siala, B., Bhiri, M.T., Bodeveix, J.P., Filali, M., 2016. An Event-B development process for the distributed BIP framework. ICFEM pages: 313-328.

Siala, B., Bodeveix, J.P., Filali, M., Bhiri, M.T., 2017. Automatic Refinement for Event-B through Annotated Patterns. 25th Euromicro International Conference on Parallel, Distributed and Network-based Processing (PDP), https://doi.org/10.1109/PDP.2017.72.

Tapia, C., San Segundo, P., Artieda, J., 2015. A PDDL-Bassed simulation system. in: Proceedings of the IADIS International Conference Intelligent Systems and Agents, Las Palmas de Gran Canaria, España, ISBN 978-989-8533-39-5. pp. 81-88.

Voisin, L., Abrial, J. R., 2014. The Rodin platform has turned ten. In Abstract State Machines, Alloy, B, TLA, VDM, and Z, https://doi.org/10.1007/978-3-662-43652-3_1.