Pattern-based requirements development

Jean-Paul Bodeveix, Arnaud Dieumegard, M Filali

Pattern-based requirements development

J.-P. Bodeveix[1], A. Dieumegard[2], and M. Filali[1]

[1] IRIT UPS CNRS , 118 Route de Narbonne , 31062 Toulouse
[2]IRT Saint Exupéry , 118 Route de Narbonne , 31432 Toulouse

Abstract

In the recent years, several high level requirements languages have been proposed. Although these languages are close to natural language, thanks to the use of dedicated patterns [15, 17], a formal semantics is attached to them. In general, modal logics have been used to express the semantics of such languages as for instance, temporal logics: branching time, linear time, temporized or not. In this paper, we are interested by the preliminary steps of the development of safety critical systems. We investigate how patterns could be used in order to generate refinements automatically. One of our main concerns is to produce Event-B machines such that the user can refine them further.

Keywords: High level requirement, refinement, development, certification, reactive system.

1 Introduction

In this paper, we are interested by the preliminary steps of the development of safety critical systems. For this purpose, several high level requirements languages have been proposed. Although these languages are close to natural language, thanks to the use of dedicated patterns [15, 17], a formal semantics is attached to them. In general, modal logics have been used to express the semantics of such languages as for instance, temporal logics: branching time, linear time, temporized or not. These high level requirements have been used for several purposes. In the context of model checking, they have been used to check a posteriori that a model has the required properties [1]. In the context of controller synthesis, illegal behaviours are statically pruned [23]. Last, in the context of runtime verification, a monitor [25, 20] is derived from specific high level requirements. Through the dynamic observation of the system, this monitor prevents illegal behaviours.

We investigate how patterns could be used in order to generate refinements automatically. One of our major concern is to produce Event-B machines such that the user can refine them further. Indeed, we propose temporal/timed and resource allocation patterns. These patterns are intended to take into account requirements incrementally. Successive refinements support this process in a safe way. Our objective here is to automatically generate Event-B models by formally expressing the requirements of the system. Furthermore, we analyse how the proposed approach can for instance make easier the use of methodology based development [22]. The rest of the paper is organized as follows: Section 2 presents our working example together with its requirements. After a brief overview of Event-B in Section 3, Section 4 presents our refinement based approach. Through a case study, Section 5 presents additional refinements. After reviewing some related works in Section 6, Section 7 concludes with some future directions of research.

2 Working example and system requirements

Throughout this document, we illustrate our approach with the Automatic Rover Protection system (ARP) of the IRT-Saint Exupéry[1] case study TwIRTee. TwIRTee is the Three Wheeled Integrated Rover Testbench for Engineering Evaluation used as the demonstrator of several projects conducted within IRT

[1]Institut de Recherche Technologique Saint Éxupéry

REQ-MIS-1	The mission of a robot is defined by an oriented path of the environment map.
REQ-MIS-2	To execute a mission means to move the robot in compliance to the path of the mission.

Table 1: Mission requirements

REQ-FUN-1	The robot shall ensure collision avoidance with as few deceleration actions as possible.
REQ-FUN-2	The robot shall always move as fast as possible in compliance with the safety constraints.
REQ-FUN-3	The robot shall always move according to its mission.
REQ-FUN-4	The robot shall move according to a speed bounded by the values 'Min_Speed' and 'Max_Speed'.
REQ-FUN-5	The robot shall adapt its speed in order to be able to stop at any time within its booked edges.

Table 2: Robots functional requirements

in Toulouse. It is used to evaluate new methods and tools in the domain of hardware/software co-design, virtual integration, and application of formal methods for the development of equipment. TwIRTee's architecture, software, and hardware components are representative of a significant family of aeronautical, spatial and automotive systems [11]. A robot moves on a topology and performs a mission. A mission is defined by a start time and an ordered set of waypoints to be passed-by. Missions are planned off-line and transmitted to the robot by a supervision station. To go from the first waypoint to the last, the robot moves on a track that is materialized by a grey line on the ground.

In a more abstract way, a mission (`REQ-MIS-1` in table 1) can be modelled by a path in a graph where edges represent zones of the track joining two waypoints. A robot shares the edges with several identical robots and moves only according to its mission (`REQ-FUN-3` in table 2). In order to prevent collisions, each of them embeds a protection function (or ARP) whose aim is to maintain some specified spatial and temporal distance between them (`REQ-SAF-1` in table 3). In the version of the system we are using here, the ARP essentially acts by booking edges prior to move on them (`REQ-SAF-1-BOO` in table 3) and by releasing previously booked edges (`REQ-SAF-1-REL` in table 3) after they have been visited.

In order to take the appropriate action, the ARP of a robot only has the following pieces of information

REQ-SAF-1	Any couple of robot in the environment shall be separated by a certain amount of edges. This property shall be ensured autonomously by each robot.
REQ-SAF-1-BOO	The robot shall book edges before moving to them.
REQ-SAF-1-BOO-v1	The robot shall book the next edge.
REQ-SAF-1-BOO-v2	The robot shall book N edges in advance.
REQ-SAF-1-BOO-v3	The robot shall book its whole mission before starting it.
REQ-SAF-1-REL	The robot shall release already visited edges.
REQ-SAF-1-REL-v1	The robot shall release previously booked edges after each movement.
REQ-SAF-1-REL-v2	The robot shall release previously booked edges after N movements.
REQ-SAF-1-REL-v3	The robot shall release previously booked edges after acomplishing a mission.

Table 3: Robots safety requirements

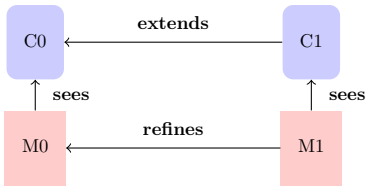


Figure 1: Event-B development step

available: the topology, the position of all other robots transmitted by a centralized supervision station, and its own attitude, position, and speed. The requirements presented here will be used throughout the paper. It is additionally worth noting that it is on purpose that we provide in table 3 multiple alternative detailed requirements (v1, v2, and v3) related to booking and release of edges. These represent variants of possible implementations solutions for the main safety requirement.

3 A brief overview of Event-B

The Event-B method allows the development of correct by construction systems and software [3]. It supports a formal development process based on a refinement mechanism with mathematical proofs. Figure 1 illustrates a refinement step where a machine M0 using a context C0 (the **sees** edge) is refined (the **refines** edge) by a machine M1 using an extension C1 of C0 (the **extends** edge). Contexts define abstract data types through sets, constants and axioms while machines define symbolic labelled transition systems. The state of a transition system is defined as the value of machine variables. Labelled transitions are defined by events specifying the new value of variables while preserving invariants. Moreover, the **theorem** clause expresses facts that should be satisfied. Proof obligations for wellformedness, invariant preservation and theorems are automatically generated by the Rodin tool [24]. They can be discharged thanks to automatic proof engines (CVC4, Z3 ...) or through human-assisted proofs.

3.1 Notations

For the most part, Event B uses standard set theory and its usual set notation. As a matter a fact, in Event-B, arrays and functions are both considered as sets of couples. Some notations are specific to Event B :

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

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

3.2 Specification and refinement of the working example

As a preliminary example for section 4, we consider the problem of robots which have to achieve a mission over a set of zones connected through a given topology specified as the neighbourhood relation over these zones. A mission is specified by an initial zone and a linear and continuous trajectory respecting the topology. A zone is exclusive and cannot be preempted. We elaborate such a specification progressively

through two steps: first, we take into account that the robots have to move according to a topology. Then, we refine this specification by taking into account that robots have an exclusive access to zones.

The static description of zones and missions is given by the context `Topology`. `Topology` introduces `ZONE` and `ROBOT` as abstract **sets**. Then, the initial location of each robot is given by the **constant** `init`: a total injection from `ROBOT` to `ZONE`. The constant relation `edge` specifies the topology of the zones: two zones are in relation if one goes from one to another in one edge. The axiom `@edge_zones_ty` introduces the topology as a relation. The axiom `@edge_irrefl`[2] expresses that a move occurs between two different nodes. The mission of each robot is specified by `mission`. The axiom `mission_ty` specifies, through the use of a partial function, that from each reached location, that a robot moves possibly to another location deterministically. The axioms `mission_WD, mission_connexity` specify respectively the corresponding properties.

The machine `Move` describes the dynamics of the robot. The state of the system is represented by the variable `location` containing the position of each robot. It is a function over the domain `ROBOT`. The dynamics is specified through two events: the `INITIALISATION` where `location` is initialized by the constant `init`, the `Next` event makes any robot `r` moves to its next location according to its mission. Its guard, labelled by `@r` ensures that the current location `location(r)` has a successor in the mission of `r`.

Remark. At this level, exclusion is not ensured since a zone can be occupied by more than one robot. The no preemption property is ensured since once a robot `r` occupies `location(r)`, this location can only be changed by the `Next` move of `r`.

```
context Topology
sets ZONE // cross point and its neighborhood
     ROBOT
constants init edge mission
axioms
  @finite_Z finite (ZONE)
  @init init ∈ ROBOT ⤖ ZONE
  @edge_zones_ty edge ∈ ZONE ↔ ZONE
  @edge_irrefl edge ∩ id = ∅
  @mission_ty mission ∈ ROBOT → (ZONE ⇸ ZONE)
  @mission_init ∀r· init(r) ∈ dom(mission(r))
  @mission_WD ∀ r · mission(r) ⊆ edge
  @connexity ∀r· ∀cl· init(r) ∈ cl ∧ mission(r)[cl] ⊆ cl
                    ⇒ dom(mission(r)) ⊆ cl
end
```

```
machine Move sees Topology
variables location
invariants
  @position_ty location ∈ ROBOT → ZONE
events
  event INITIALISATION
    then
      @position location := init
  end
  event Next
    any r where
      @r location(r) ∈ dom(mission(r))
    then
      @m location(r) := mission(r)(location(r))
  end
end
```

Figure 2: Event-B context and machine

Listing 1: Event-B refinement

```
machine Security refines Move sees cSecurity

variables location

invariants
  @excl location ∈ ROBOT ⤖ ZONE

events
  event INITIALISATION extends INITIALISATION end

  event Next extends Next
  when
    @excl mission(r)(location(r)) ∉ location[ROBOT \{r}]
  end
end
```

Listing 1 illustrates an Event-B refinement named `Security` of the preceding machine `Move`. Through this refinement, we take into account that locations are exclusive. The *invariant* `@excl` expresses exclusion through the injectivity of `location`. Moreover, this refinement introduces, through the keyword **extends**, the `Next` event as a superposition [6] to the event `Next` of the `Move` machine. Thanks to the guard `excl`, we ensure that locations remain exclusive and that no preemption occurs. Actually, the guard ensures that the robot will move to a free location.
In Event-B, the proof obligations enforce a weak refinement semantics [21].

[2] In Event-B, `id` is the identity relation.

4 Pattern-based refinement of Event-B machines

The Event-B methodology consists in capturing requirements incrementally through a sequence of *horizontal* refinements [3]. Our goal consists in giving an assistance to this process by making the generation of the refined machine automatic given a well formatted requirement expressed using structured natural language. Such language is often used to write more consistent, unambiguous and verifiable requirements [19]. This is the case for example in Boeing requirements management process [8] as well as in the Stimulus approach for requirements engineering [16]. It is interesting to remark that both latter approaches insist on the importance of formalizing requirements. Moreover, the Stimulus approach offers means to debug and validate requirements. Last, it has also been recognised that formal methods are well suited to capture requirements [2] in an incremental way. In this paper we investigate pattern-based textual requirements from which eventually one could build a formal model of his requirements and assess their soundness.

We have considered two kinds of patterns: temporal/timed patterns and resource management patterns. The first ones allow to declare several kinds of possibly timed precedence constraints. The second ones introduce events to allocate and free resources together with an invariant stating that required resources are always booked. Since a machine can be the source of further refinements, the generated machine should be human readable. For this purpose, requirements are expressed through *annotated* patterns so that names (of new state variables or events) are provided by the user. Moreover, labels of introduced guards and invariants provide traceability data between the generated model and the source pattern.

4.1 Temporal and timed patterns

The patterns we propose for event ordering are close to Dwyer's ones [15]. The usual way to integrate these patterns in a tool is to consider their temporal logic semantics (mostly LTL-based) and to call an LTL to Büchi tool to get an automaton. Then, the product of the system and the automaton is analysed. This analysis can be done statically to check some system properties. It can also be done dynamically to enforce some system properties. Here, patterns are used at design time to help taking into account system requirements and eventually build correct by construction systems.

We envision a tooled method that would take as input an Event-B model and a set of specifications corresponding to a fixed number of (timed) LTL formulas. In order to facilitate the design of correct refinement patterns, we reuse LTL to Buchi-automata translators. Thus, specification patterns are first expressed as an LTL formula enriched by propositions marking timing constraints. The SPOT tool [14] generates a Buchi automaton which is in turn rewritten manually as an Event-B machine. The refined machine is obtained by superposing the original and the synthesized machine [26].

4.1.1 Temporal patterns

The managed temporal patterns have a semantics expressed as an LTL formula of which associated Buchi automaton has few states. The user should be able to introduce names for these states using dedicated annotations of the pattern. For example, the automaton associated to a precedence constraint has two states which can be represented by a Boolean variable. This variable can either be already present in the system (leading to the generation of a proof obligation) or should be created. This principle is illustrated by the following example consisting in one of the Dweyer's precedence patterns:

new event request **precedes event** authorization (**using new status** authorization_req)

which specifies that the event `request` should be added to the model (the `new` keyword) and that the existing event `authorization` should be preceded by the newly introduced event. Precedence control is managed by the newly introduced variable `authorization_req`. It will be set by `request` and checked by `authorization`, thus strengthening its guard.

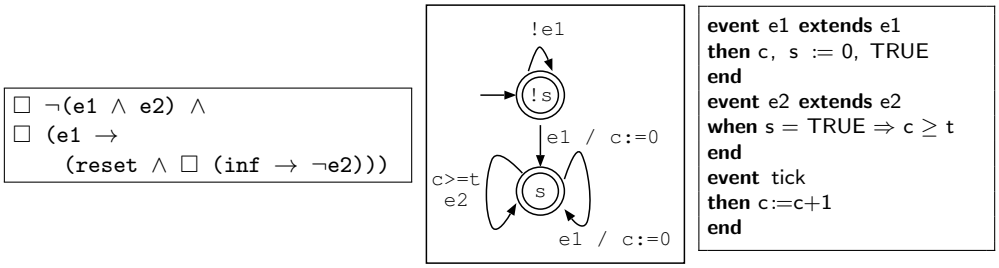
The semantics of such a pattern is defined by an event-based LTL formula [9] having the following structure: $\Box((e_2 \vee \textbf{init}) \rightarrow \mathbf{X}\ (\neg e_2\ \mathbf{W}\ e_1))$ which leads to a two-state Buchi automaton and finally to the following Event-B machine refinement pattern. The two states are distinguished in Event-B by the value of status variable `s`. If `s` is false, `e2` has not occurred and `e1` is forbidden. If `e2` is fired, `e1` can occur once. Here, `e1` is a newly introduced event while `e2` is supposed to already exist. Its guard is

strengthened and its action resets the control variable **s** thanks to the **extends** declaration.

Each occurrence of e_2 should be preceded by a e_1.		
new event e_1 **precedes each event** e_2 (**using new status** s)		
event INITIALISATION **extends** INITIALISATION **then** s := FALSE **end**	**event** e_1 **then** s := TRUE **end**	**event** e_2 **extends** e_2 **when** s = TRUE **then** s := FALSE **end**

4.1.2 Timed pattern

Timed patterns are derived from Dweyer patterns by adding some timing constraints. They have a Metric Temporal Logic (MTL[18]) semantics, which is a timed extension of LTL where temporal operators can be given a minimal/maximal duration. Contrary to LTL, to the best of our knowledge, no tool is available to transform MITL formulas to timed automata. Thus, this transformation is done partially manually by the plugin developer for each of the considered patterns. Timed automata are ultimately transformed into Event-B machines equipped with a **tick** event managing the advance of time. As an example, consider the following timed pattern: *no e2 after e1 during t time units* which can be specified in MTL as $\Box(e_1 \rightarrow \Box_{<t}\neg e_2)$, or in Timed Propositional Temporal Logic [4, 7], introducing reset quantification, as $\Box(e_1 \rightarrow c \cdot\ \Box(c < t \rightarrow \neg e_2))$ to make explicit the use of clocks. The following figure illustrates the steps leading to the construction of the refined machine generator. We make explicit the management of time by introducing two propositions: **reset** for the clock c being 0 and **inf** for the clock c being less that t. We also make explicit the fact that events exclude each other, which lead to the LTL formula. Then we apply an LTL2BA tool [14] to obtain a Buchi automaton, transformed into a timed automaton by replacing **reset** and **inf** by **c:=0** and **c<t**. The automaton is then superposed to the existing Event-B model thanks to the **extends** construct.



4.2 Resource management patterns

We propose a family of resource management patterns that vary depending on the number of events used to manage the resources. The pattern applies to a machine **M0** which declares some state variables **st** and events **ev**. Events are supposed to take an **Agent** a as a parameter. A resource management pattern is then written. The set of agents **Agent** is declared to have exclusive access to individual resources given by the **s_rsc** expression supposed to be partial function from agents to resources of some set R or to pool of resources given by the **p_rsc** relation. Several resources or pool of resources of several types could be allocated to an agent. To make the presentation of the pattern simpler, we suppose that resources belong to a unique type R. The pattern introduces a partial function **booked** from resources to agents used to ensure exclusive access. Events are declared for granting resources needed by the post-state of each of the events **ev**. Events that are not mentioned by the pattern should not update variables on which depend the resources declarations. Resource freeing is performed by a unique event named **release**.

```
machine M0
 variables st ···
axioms
  theorem @r1 s_rsc ∈ Ag ⇸ R // single
  theorem @r2 p_rsc ∈ Ag ↔ R // pool
events
  event INITIALISATION ··· end
  event ev
    any a where G_ev(a,st) then A_ev(a,st) end
  ···
end
```

```
for a : Agent
  @E1 exclusion on single R given by s_rsc
  @E2 exclusion on pool R given by p_rsc
    (using new mapper booked init init)
  granted by new event grant_ev for event ev
    (using new status ready_ev)
  released by new event release
```

The resulting machine (Figure 3) introduces two new variables: `booked` maps resources to agents to which they are exclusively linked and `ready_ev` contains the set of agents allowed to perform the event `ev`. The machine contains two new events: `grant_ev` manages resource allocation for post-states of events `ev`, and `release` which disposes resource allocations that are not necessary in the current state. The event `grant_ev` is fired when the `ev` guard is satisfied and resources can be allocated exclusively. This is expressed by introducing a resource mapping parameter `rsc` containing requirements for all possible post-states of `ev` while keeping the resource mapping `booked` ∪ `rsc` functional. Agents ready to perform `ev` are added to the `ready_ev` set. The event `release` frees unnecessary resources, i.e. resources that are not required by the current state. The pattern thus uses the [_] notation to specify that the weakest precondition [13] of the action part of the corresponding event should be computed. It has to be noted that events may be non-deterministic, in which case resources for all of the possible post-states are allocated.

```
machine M1 refines M0
 variables st booked ready_ev
 invariants
  @b booked ∈ R ⇸ Agent
  @r ready_ev ⊆ Agent
  @E1 s_rsc⁻¹ ⊆ booked
  @E2 p_rsc⁻¹ ⊆ booked
  @ready_ev ∀a· a∈ ready_ev ⇒ G_ev(a,st)
  @E1_ev ∀a· a ∈ ready_ev ⇒
    [A_ev(a,st)]( s_rsc⁻¹ ⊆ booked)
  @E2_ev ∀a· a ∈ ready_ev ⇒
    [A_ev(a,st)]( p_rsc⁻¹ ⊆ booked)
events
  event INITIALISATION extends INITIALISATION
  then
    @bnr_i booked,next_ready := init ,∅
  end

  event grant_ev
  any a rsc where// grant for any post−state of ev
    @a a ∈ Agent
    @G G_ev(a,st)
    @E1 [A_ev(a,st)]( s_rsc⁻¹ ⊆ rsc)
    @E2 [A_ev(a,st)]( p_rsc⁻¹ ⊆ rsc)
    @rsc booked ∪ rsc ∈ R ⇸ Agent
  then
    @booked booked := booked ∪ rsc
    @ready ready_ev := ready_ev ∪ {a}
  end

  event release
  any rsc where
    @a1 ran(s_rsc) ∩ rsc = ∅
    @a2 ran(p_rsc) ∩ rsc = ∅
  then
    @b booked := rsc ⩤ booked
    @r ready_ev := ∅
  end

  event ev refines ev
  any a where
    @r a ∈ ready_ev
  then
    @a A_ev(a,st)
    @nr ready_ev := ∅ // reallocate before next ev
  end
end
```

Figure 3: Refinement pattern for resource management

The generic pattern above can be instantiated for our example to take into account the no collision requirement `REQ-SAF-1` and the booking procedure requirement `REQ-SAF-1-BOO` specified in section 2. The resource release event is left non-deterministic but could be constrained to follow some protocol to conform with the given requirement, for example `REQ-SAF-1-REL v2`.

for r : ROBOT
@excl: **exclusion on single** ZONE **given by** location (**using new mapper** booked)
granted by new event book **for event** move (**using new status** ready)
released by new event release

5 Case study

In this section we briefly describe additional refinements to be implemented on our model of the ARP system. Each refinement is related to requirements and provide additional information on the properties of the expected related patterns.

Alternative resource management refinements. The previous pattern instantiation generates a `grant_ev` event booking resources given by the `location` variable. By modifying only this last variable of the pattern it is possible to evaluate design variations for booking resources: `Static Booking` where the whole mission is booked, the robot performs it and then frees the mission zones (requirements `REQ-SAF-1-BOO-v3` and `REQ-SAF-1-REL-v3` in table 3); `Dynamic Booking` where the next zone in the mission is booked, the robot moves on it and then frees the zone that was just left (requirements `REQ-SAF-1-BOO-v1` and `REQ-SAF-1-REL-v1` in table 3); `Temporal Booking` where zones in the mission are booked for some time range, the robot moves to the next zone and then frees the zone that was just left (requirements `REQ-SAF-1-BOO-v2` and `REQ-SAF-1-REL-v1` in table 3); `Temporal booking with regular release` where zones in the mission are booked for some time range, the robot moves to the next zone and releases zones at regular time interval (requirements `REQ-SAF-1-BOO-v2` and `REQ-SAF-1-REL-v2` in table 3). Additional elements needs to be provided on the pattern in order to automatically implement the variations of the `release` event. It is indeed necessary to strengthen the guards of the event and specify additional constraints for the `rsc` variable. This can be done either by extending the resource management pattern with the release strategy or through another pattern.

Move refinements. The `Next` event can also be refined to take into account parameters of the robots movement as for example the speed of the robot. We propose to introduce the speed of the robot through two distinct refinements of the `Next` event taking into account requirements `REQ-FUN-4` and `REQ-FUN-5` in table 2 constraining the value of a new variable: `Next_Nominal`: in this event, booking has been done and it is possible to move to the zone afterward; `Next_Warning`: in this event, booking has been done but it is not possible to move to the zone afterward. The possible values of the `speed` variable are introduced as a constant enumeration: {`Min, Inc, Dec, Max`} for respective Minimal value, Increasing value, Decreasing value, and Maximal value. These values represent an abstraction of the possible robot instantaneous speed and acceleration.

Refinements composition. As the `Speed` refinement does not have common variables with the `Booking` and `Release` events, it is easy to build a final Event-B model comprising both refinements by following the composition/decomposition approach [27]. Thus, the actual system can be synthesized by selecting specific refinements within the two branches. This is allowing for a software product line based approach to formal refinements where variation points are expressed at the pattern level allowing for a better management of their complexity. However, we should mention, that in real cases, resource booking may depend on the speed. Indeed, we have a dependent composition which deserves further study.

6 Related works

It is of common knowledge in formal refinement that rework of the requirement document is necessary [28] (additional citations can be provided if necessary). It leads to the expression of the requirements as a bigger number of smaller and simpler requirements. The authors advocate for the ordering of these new requirement (leading to the creation of a refinement strategy) and their application in refinements. Our approach is to automate the refinement creation by expressing new requirements using patterns. It may help in the scalability of the approach proposed in [28] by focusing the effort on the creation of the refinement strategy and the expression of the requirements instead of the writing of the formal refinement per say.

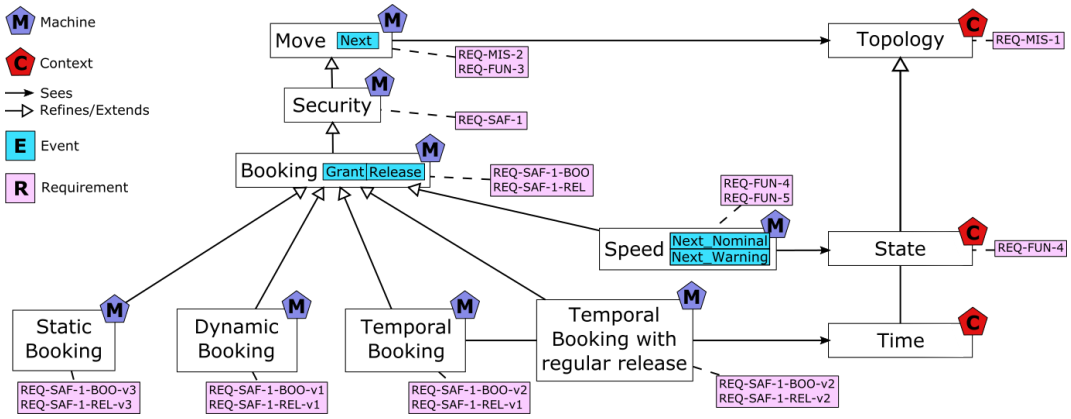


Figure 4: First refinements steps for the ARP Event-B model

Operationalisation of formalized requirements have been studied for example in the work from Aziz et Al [5]. They propose to rely on a goal-oriented requirement engineering approach where requirement are expressed using formal patterns for a limited amount of constructs. Finally, they automatically derive Event-B models from the requirements and complete the generated machine for verification of the correct modelling of the requirements by the generated machine. Our approach is to integrate the use of patterns directly in the development process by relying on pre-existing Event-B models and formal requirements to automatically generate correct refinements and complete the generation.

7 Conclusion and future works

Our test bed TwIRTee has convinced us that automatic rover protection systems deserve dedicated frameworks. With respect to requirements, the pattern approach seems promising especially when it is applied together with formal methods. This paper has investigated such an approach through the refinement-based method Event-B. More precisely, we have been concerned by temporal/timed and resource management patterns.

In order to enhance this practice, it seems to us interesting to investigate also approaches based on ontologies and "natural language" [12]. Concerning the technical part, handling modalities combining spatial and temporal (timed) concerns and quantified variants of modal logics should give better semantic foundations. With respect to our work, combining temporal/timed and resource management patterns looks promising. Last, from the side of the implementation, the link with elaborated allocation techniques [10] should pave the way towards effective code generation.

References

[1] N. Abid, S. Dal-Zilio, and D. L. Botlan. Real-time specification patterns and tools. In *FMICS 2012, Paris, France, August 27-28, 2012. Proceedings*, pages 1–15, 2012.

[2] J.-R. Abrial. Formal methods in industry: Achievements, problems, future. In *Proceedings of the 28th International Conference on Software Engineering*, ICSE '06, pages 761–768, New York, NY, USA, 2006. ACM.

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

[4] R. Alur and T. A. Henzinger. A really temporal logic. *J. ACM*, 41(1):181–204, 1994.

[5] B. Aziz, A. Arenas, J. Bicarregui, C. Ponsard, and P. Massonet. From goal-oriented requirements to Event-B specifications. In *First Nasa Formal Method Symposium (NFM 2009)*, 2009.

[6] R. Back and K. Sere. Superposition refinement of reactive systems. *Formal Asp. Comput.*, 8(3):324–346, 1996.

[7] P. Bouyer. Model-checking timed temporal logics. *Electr. Notes Theor. Comput. Sci.*, 231:323–341, 2009.

[8] R. S. Carson. Implementing structured requirements to improve requirements quality. *INCOSE International Symposium*, 25(1):54–67, 2015.

[9] S. Chaki, E. M. Clarke, J. Ouaknine, N. Sharygina, and N. Sinha. State/event-based software model checking. In E. A. Boiten, J. Derrick, and G. Smith, editors, *IFM , Canterbury, UK, April 4-7, Proceedings*, volume 2999 of *Lecture Notes in Computer Science*, pages 128–147. Springer, 2004.

[10] K. Chandy and J. Misra. *Parallel Program Design, A Foundation*. Addison-Wesley, 1988.

[11] P. Cuenot, E. Jenn, E. Faure, N. Broueilh, and E. Rouland. An Experiment on Exploiting Virtual Platforms for the Development of Embedded Equipments. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016.

[12] C. Czepa, H. Tran, U. Zdun, T. T. T. Kim, E. Weiss, and C. Ruhsam. Ontology-based behavioral constraint authoring. In R. M. Dijkman, L. F. Pires, and S. Rinderle-Ma, editors, *20th IEEE International Enterprise Distributed Object Computing Workshop, EDOC Workshops 2016, Vienna, Austria, September 5-9, 2016*, pages 1–8. IEEE Computer Society, 2016.

[13] E. W. Dijkstra. *A Discipline of Programming*. Prentice-Hall, 1976.

[14] A. Duret-Lutz, A. Lewkowicz, A. Fauchille, T. Michaud, E. Renault, and L. Xu. Spot 2.0 — a framework for LTL and ω-automata manipulation. In *Proceedings of the 14th International Symposium on Automated Technology for Verification and Analysis (ATVA'16)*, volume 9938 of *Lecture Notes in Computer Science*, pages 122–129. Springer, Oct. 2016.

[15] M. B. Dwyer, G. S. Avrunin, and J. C. Corbett. Patterns in property specifications for finite-state verification. In *Proceedings of the ICSE' 99, Los Angeles, USA, May 16-22,*, pages 411–420, 1999.

[16] B. Jeannet and F. Gaucher. Debugging Embedded Systems Requirements with STIMULUS: an Automotive Case-Study. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, Jan. 2016.

[17] S. Konrad and B. H. C. Cheng. Real-time specification patterns. In *Proceedings of the 27th International Conference on Software Engineering*, ICSE '05, pages 372–381, New York, USA, 2005. ACM.

[18] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Systems*, 1990.

[19] D. Lempia, B. Schindel, T. Hrabik, S. McGill, and M. Graber. Using Visual Diagrams and Patterns for Consistent and Complete Requirements. In *INCOSE International Symposium*, volume 26, pages 415–429. Wiley Online Library, 2016.

[20] M. Leucker and C. Schallhart. A brief account of runtime verification. *J. Log. Algebr. Program.*, 78(5):293–303, 2009.

[21] R. Milner. *Communication and Concurrency*. Prentice-Hall, Upper Saddle River, NJ, USA, 1989.

[22] B. Ouni, P. Gaufillet, E. Jenn, and J. Hugues. Model Driven Engineering with Capella and AADL. In *ERTSS 2016*, pages 680–689, Toulouse, France, Jan. 2016.

[23] J.-B. Raclet, E. Badouel, A. Benveniste, B. Caillaud, A. Legay, and R. Passerone. A modal interface theory for component-based design. *Fundamenta Informaticae*, 108(1-2):119–149, 2011.

[24] `http://www.event-b.org/`.

[25] F. B. Schneider. Enforceable security policies. *ACM Trans. Inf. Syst. Secur.*, 3(1):30–50, 2000.

[26] B. Siala, J. Bodeveix, M. Filali, and M. T. Bhiri. Automatic refinement for Event-B through annotated patterns. In I. V. Kotenko, Y. Cotronis, and M. Daneshtalab, editors, *25th Euromicro International Conference on Parallel, Distributed and Network-based Processing, PDP 2017, St. Petersburg, Russia, March 6-8, 2017*, pages 287–290. IEEE, 2017.

[27] R. Silva and M. Butler. *Shared Event Composition/Decomposition in Event-B*, pages 122–141. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[28] W. Su, J.-R. Abrial, R. Huang, and H. Zhu. From requirements to development: methodology and example. In *Formal Methods and Software Engineering*, pages 437–455. Springer, 2011.