Automatic generation of proof obligations parameterised by domain theories implementation in Event-B : The EB4EB Framework

Peter Riviere

RÉPUBLIQUE FRANÇAISE
Liberté Égalité Fraternité

Université de Toulouse

Doctorat de l'Université de Toulouse

préparé à Toulouse INP

Génération automatique d'obligations de preuves paramétrée par des théories de domaine dans Event-B : Le cadre de travail EB4EB

Thèse présentée et soutenue, le 7 juin 2024 par

Peter RIVIERE

École doctorale
EDMITT - Ecole Doctorale Mathématiques, Informatique et Télécommunications de Toulouse

Spécialité
Informatique et Télécommunications

Unité de recherche
IRIT : Institut de Recherche en Informatique de Toulouse

Thèse dirigée par
Neeraj SINGH et Yamine AIT AMEUR

Composition du jury
Mme Régine LALEAU, Présidente, Université Paris Est Créteil
M. Frédéric MALLET, Rapporteur, Université Côte d'Azur
M. David DELAHAYE, Rapporteur, Université de Montpellier
M. Michael LEUSCHEL, Rapporteur, Heinrich-Heine-Universität Düsseldorf
Mme Rosemary MONAHAN, Examinatrice, Maynooth University Faculty of Science and Engineering
M. Neeraj SINGH, Directeur de thèse, Toulouse INP
M. Yamine AÏT-AMEUR, Co-directeur de thèse, Toulouse INP

Membres invités
M. Guillaume DUPONT, Toulouse INP

Abstract

Nowadays, we are surrounded by complex critical systems such as microprocessors, railways, home appliances, robots, aeroplanes, and so on. These systems are extremely complex and are safety-critical, and they must be verified and validated. The use of state-based formal methods has proven to be effective in designing complex systems. Event-B has played a key role in the development of such systems. Event-B is a formal system design method that is state-based and correct-by-construction, with a focus on proof and refinement. Event-B facilitates verification of properties such as invariant preservation, convergence, and refinement by generating and discharging proof obligations.

Additional properties for system verification, such as deadlock-freeness, reachability, and liveness, must be explicitly defined and verified by the designer or formalised using another formal method. Such an approach reduces re-usability and may introduce errors, particularly in complex systems.

To tackle these challenges, we introduced the reflexive EB4EB framework in Event-B. In this framework, each Event-B concept is formalised as a first-class object using First Order Logic (FOL) and set theory. This framework allows for the manipulation and analysis of Event-B models, with extensions for additional, non-intrusive analyses such as temporal properties, weak invariants, deadlock freeness, and so on. This is accomplished through Event-B Theories, which extend the Event-B language with the theory's defined elements, and also by formalising and articulating new proof obligations that are not present in traditional Event-B. Furthermore, Event-B's operational semantics (based on traces) have been formalised, along with a framework for guaranteeing the soundness of the defined theorems, including operators and proof obligations. Finally, the proposed framework and its extensions have been validated across multiple case studies, including Lamport's clock case study, read/write processes, the Peterson algorithm, Automated Teller Machine (ATM), autonomous vehicles, and so on.

Résumé

De nos jours, nous sommes entourés de systèmes critiques complexes tels que les microprocesseurs, les trains, les appareils intelligents, les robots, les avions, etc. Ces systèmes sont extrêmement complexes et critiques en termes de sûreté, et doivent donc être vérifiés et validés. L'utilisation de méthodes formelles à états s'est avérée efficace pour concevoir des systèmes complexes. Event-B a joué un rôle clé dans le développement de tels systèmes. Event-B est une méthode formelle de conception de systèmes à états avec une approche correcte par construction, qui met l'accent sur la preuve et le raffinement. Event-B facilite la vérification de propriétés telles que la préservation des invariants, la convergence et le raffinement en générant des obligations de preuve et en permettant de les décharger.

Certaines propriétés additionnelles du système, telles que l'absence d'interblocage, l'atteignabilité ou encore la vivacité, doivent être explicitement encodées et vérifiées par le concepteur, ou formalisées à l'aide d'une autre méthode formelle. Une telle approche pénalise la réutilisabilité des modèles et des techniques, et peut introduire des erreurs, en particulier dans les systèmes complexes.

Pour pallier cela, nous avons introduit un *framework* réflexif EB4EB, formalisé au sein de Event-B. Dans ce cadre, chacun des concept d'Event-B est formalisé comme un objet de première classe en utilisant la logique du premier ordre (FOL) et la théorie des ensembles. EB4EB permet la manipulation et l'analyse de modèles Event-B, et permet la définition d'extensions afin de réaliser des analyses supplémentaires non-intrusives sur des modèles, telles que la validation de propriétés temporelles, l'analyse de la couverture d'un invariant, ou encore l'absence de blocage. Ce framework est réalisé grâce aux théories d'Event-B, qui étendent le langage d'Event-B avec des éléments définis dans des théories, et aussi en formalisant de nouvelles obligations de preuves, qui ne sont pas présentes initialement dans Event-B. De plus, la sémantique opérationnelle d'Event-B (basée sur les traces) a été formalisée, de même qu'un cadre qui sert à garantir la correction des théorèmes définis, y compris les opérateurs et les obligations de preuve. Enfin, le cadre proposé et ses extensions ont été validés dans de multiples études de cas, notamment l'horloge de Lamport, le problème du lecteur/rédacteur, l'algorithme de Peterson, les distributeurs automatiques de billets (DAB), les véhicules autonomes, etc.

Contents

List of Tables

List of Figures

List of Listings

Part I

Introduction

Chapter 1

Introduction

1.1 Context

Complex information systems include several components and actors such as hardware, software, physical plants, communication devices, humans, surrounding environment and so on. These systems can be data-driven, distributed, centralised, reactive, real-time, etc. The design of such systems relies on several methods, techniques, tools, standards, processes, and so on to ensure their quality. In particular, when these systems are critical, more attention and confidence in their design are required. The advancement of formal methods has demonstrated their effectiveness in assisting with the design phase of critical systems by providing powerful modelling languages and associated verification and validation techniques.

Among the proposed formal methods, we are particularly interested in state-based formal methods. These methods have demonstrated the ability to model and reason about complex systems in order to establish properties that reflect the modelled requirements. They have been particularly effective in ensuring system safety by verifying safety and other behavioural properties such as liveness. These approaches manipulate states and their evolution while allowing properties to be expressed and reasoned about. Several state-based methods have been designed supporting various aspects of system modelling, validation and/or verification and addressing functionality, safety, real-time, reliability, security, etc. Each of these methods are efficient in their *"area of expertise"*. As a result, different formal methods may be set up to address specific requirements for a single system.

The use of different formal methods in a multi-modelling setting may result in semantic mismatches and thus different interpretations.

1.2 The addressed problem

The introduction of semantic mismatches as a result of different modelling languages is a key issue. Several proposals for avoiding or reducing such semantic mismatches have been proposed for many formal methods.

Our work focuses on addressing the issue of reducing heterogeneity that arises from the use of multiple formal modelling languages and verification techniques.

Below, we provide an overview of the different approaches we have identified.

- *Multi-modelling.* A collection of formal models is presented, each one focusing on a specific aspect of the designed system. Informal arguments are included to ensure that the desired properties are related to the same system. Challenges may arise when a rely guarantee reasoning is performed between different models. In this instance, no formal proof of correctness is offered, only informal arguments are provided.

- *Ad hoc transformations.* In this case, certain models are transformed into another model using a target modelling language. The transformation process is informal and, consequently, not certified.

- *Model transformations.* Here, model transformations are discussed. Tools, typically in the form of plug-ins, are provided for transforming a model provided in a source modelling language into another model in a target modelling language. There are numerous tools available in the field of Model-Driven Engineering (MDE) for performing model transformations, like ATL [Jouault et al., 2006], Kermeta [Jézéquel et al., 2009] or Viatra [Csertán et al., 2002] In this case, semantic mismatch is addressed by either certifying each model produced by the transformation or certifying the transformation itself.

- *Certified model transformations.* They align with the earlier example where the transformation generates a certificate that guarantees semantic preservation. Typically, this approach necessitates an additional formal framework like a proof assistant to help with certificate generation. In this scenario, the semantics of both the source and target modelling languages are formally defined. An illustrative and widely recognised example is CompCert [Leroy et al., 2016], which validates the conversion of C code into assembly language.

- *Exportation in a unified modelling language.* In this case, the semantics of the modelling language are defined in a unified framework. The models are exported as instances within this unified modelling language. Exporting or instantiating various formal models within a unified framework or metamodel that expresses their semantics. This method involves formalising each modelling language and transformation within this unified framework. For examples, Unifying Theory of Programming (UTP) [He & Hoare, 1998] and COQ [Bertot & Castéran, 2010] can be used for this purpose. In [Woodcock & Cavalcanti, 2004], the authors have used UTP to formalise Z [Spivey, 1985] and CSP [Hoare, 1978].

- *Extension of the modelling language.* The method involves creating extensions of the formal modelling language that enable the description of concepts and properties, along with a verification process to validate these

properties. Of course, these extensions must ensure the coherence of the original semantics of the extended formal modelling language.

Each of the approaches described above aims to create a framework that enables the modelling and verification of complex systems in a unified environment. Each approach has its advantages and disadvantages. Indeed, ensuring the certification of the transformation/export process and receiving feedback from the model analyses are key requirements when dealing with formal verification.

1.3 Our proposal

Our work focuses on refinement and proof based development of complex systems using the Event-B method. We have chosen the Event-B method as it supports system development and checking of safety and functionality together with some behavioural properties.

During our experiments with this formal method, we encountered many situations in which it was impossible, or too complicated to model specific properties using Event-B, such as temporal logic properties or domain specific properties. Therefore, we needed to bridge the gap between Event-B and other formal modelling languages, allowing us to express and verify related properties.

Following the various approaches described in the previous section, we propose a hybrid approach. It relies on both *extending the modelling language* and *exporting models to a unified modelling language*. Certainly, we utilise the concept of *extensions* in Event-B to enhance the formal expression and verification capabilities that are not present in the native Event-B. Additionally, we establish a meta-model of Event-B using Event-B itself (known as reflexive modelling) where Event-B models are *exported* as instances of this meta-model. The definition of Event-B extension/export is possible since Event-B,

- is grounded in set theory and first order logic (FOL), which are useful for modelling the core concepts of systems,
- offers meta-modelling capabilities thanks to the availability of set theory and FOL. It also proposes the capability to define additional algebraic theories,
- allows for both *deep* and *shallow* modelling capabilities.

1.4 Our contribution

In order to support the capability of handling the verification of properties that are not explicitly formalised in Event-B, we have developed a framework allowing to

- support the explicit formalisation of various properties, including their associated semantics using traces.

- automatically generate proof obligations to ensure newly defined properties are valid,
- check the consistency of the proposed formalisation for Event-B via the definition of its trace-based semantics.

By leveraging algebraic theories, the construction of this framework led to the creation of a series of formalised components and services in the following manner:

- First, a meta-model is presented that formalises all Event-B features, such as states, events, proof obligations (invariant, feasibility, etc.).
- A model of traces is provided to formalise the trace-based semantics of Event-B, along with a generic template for proving the soundness of both Event-B and any defined extensions.
- Two different modelling techniques are used to support the formalisation of Event-B models in the designed framework:
  - *Deep modelling*, where an Event-B machine is represented by a context instantiating the above defined meta-model,
  - *Shallow modelling*, where an Event-B machine is defined as a refinement of a generic Event-B machine.
- An annotation mechanism is used to link a meta-model of externally defined semantic features with the core Event-B meta-model. This mechanism provides the ability to analyse Event-B models that incorporate these additional semantic features.

It is worth noting that for each of the exports and extensions we defined, a proof of consistency is provided to certify the correctness of these definitions. Furthermore, all the developments we have made are supported by the Rodin platform and are included in the appendices to this thesis.

1.5 Organisation of the manuscript

The following chapters of this thesis describe the specifics of our contribution. We have grounded our thesis manuscript in a collection of research papers published since the start of our thesis work. The chapters are arranged in chronological order to reflect the work we completed. Each chapter wraps up one or more published research papers relevant to the described contribution. In addition to the published papers, an overview and assessment are provided. They enable the integration of the various contributions.

This thesis is structured as follows.

In Chapter 2, we present the foundation of our research, which is the meta-model formalised as an Event-B theory of core Event-B. We discuss the trace-based semantics of Event-B and provide the essential theorems for consistency.

This chapter synthesises the content of the published papers [Riviere et al., 2022a, 2022b, 2023b].

In Chapter 3, we demonstrate the ability to conduct non-intrusive Event-B model analyses using Event-B without additional semantic features. These analyses are outlined by either proposing new proof obligations or manipulating the structure of the Event-B models. This chapter encapsulates the content of the published paper [Riviere et al., 2023c].

In Chapter 4, we present the initial extension we developed for Event-B to address temporal logic properties. We formalise a collection of operators for temporal logic expressions and proof obligations within a theory that extends the meta-model theory of core Event-B. This chapter compiles the content of the published papers [Riviere et al., 2023a].

Chapter 5 introduces the second type of extension, which enables the annotation of Event-B models by linking them to externally defined domain knowledge features, particularly ontologies. This chapter summarises the contents of the published papers [Mendil et al., 2022].

Finally, Chapter 6 introduces an alternative, reusable method for proving invariant proof obligations. This approach emerged from our experiments and research into defining Event-B proof obligations in the meta-model. It entails splitting the proof of these obligations into two parts: one at the theory level and one at the model level, thereby reducing the complexity of the proof on the model side. This chapter covers the contents of the published papers [Aït Ameur et al., 2022].

Chapter 7 provides a conclusion and set of perspectives.

Finally, every model presented in the papers have been developed and proved on the Rodin platform, and are included as appendices, at the end of this manuscript.

1.6 List of Published Paper:

1.6.1 Journal

1. Riviere, P., Singh, N. K., & Aït Ameur, Y. [2022b]. Reflexive Event-B: Semantics and Correctness the EB4EB Framework. *IEEE Transactions on Reliability*, 1–16

2. Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2024,(Submitted)]. Extending the EB4EB framework with parameterised events

1.6.2 International Conference

1. Riviere, P., Singh, N. K., & Aït Ameur, Y. [2022a]. EB4EB: A Framework for Reflexive Event-B. *International Conference on Engineering of Complex Computer Systems, ICECCS 2022*, 71–80

2. Aït Ameur, Y., Dupont, G., Mendil, I., Méry, D., Pantel, M., Riviere, P., & Singh, N. K. [2022]. Empowering the Event-B Method Using External Theories. *IFM*, *13274*, 18–35

3. Mendil, I., Riviere, P., Aït Ameur, Y., Singh, N. K., Méry, D., & Palanque, P. A. [2022]. Non-intrusive annotation-based domain-specific analysis to certify event-b models behaviours. *29th Asia-Pacific Software Engineering Conference, APSEC*, 129–138

4. Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2023c]. Standalone Event-B models analysis relying on the EB4EB meta-theory. *International Conference on Rigorous State Based Methods, ABZ 2023*

5. Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2023a]. Formalising liveness properties in Event-B. *NASA Formal Methods 2023*

6. Ferrarotti, F., Rivière, P., Schewe, K.-D., Singh, N. K., & Aït Ameur, Y. [2024]. A complete fragment of LTL(EB). *13th International Symposium on Foundations of Information and Knowledge Systems FoIKS 24*

7. Riviere, P., Kobayashi, T., Singh, N. K., Ishikawa, F., Aït Ameur, Y., & Dupont, G. [2024,(Submitted)]. On-the-Fly Proof-Based Verification of Reachability in Autonomous Vehicle Controllers Relying on Goal-Aware RSS

1.6.3 Workshop

1. Riviere, P., Singh, N. K., & Aït Ameur, Y. [2021]. Data-types definitions: Use of Theory and Context instantiations Plugins. *9th Rodin User and Developer Workshop collocated with the ABZ 2021 Conference*, 1–6

2. Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2023b]. Proof automation for Event-B theories. *10th Rodin User and Developer Workshop collocated with the ABZ 2023 Conference*

Part II

Contribution

Chapter 2

The Reflexive Framework EB4EB

Overview

This chapter introduces the EB4EB framework, which is based on meta-modelling techniques inspired by reflexive modelling concepts. This framework allows us to manipulate Event-B components, as well as extend the core of the Event-B language by defining new meta theories, such as the introduction of new proof obligations, LTL and CTL properties, domain specific properties, and so on. This framework consists of datatypes, operators, theorems, and proof-rules that are defined using algebraic definitions derived from the Event-B book. In addition, each defined operator is equipped with *well-defined* conditions to ensure the correct usages of the defined operators. The trace-based semantics preserves the operational soundness of each defined operator. This framework is developed in the Event-B modelling language using the *Theory plugin* and Rodin IDE. A set of new tactics is defined to assist proving.

Associated papers of this chapter:

- Riviere, P., Singh, N. K., & Aït Ameur, Y. [2022a]. EB4EB: A Framework for Reflexive Event-B. *International Conference on Engineering of Complex Computer Systems, ICECCS 2022*, 71–80
- Riviere, P., Singh, N. K., & Aït Ameur, Y. [2022b]. Reflexive Event-B: Semantics and Correctness the EB4EB Framework. *IEEE Transactions on Reliability*, 1–16
- Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2023b]. Proof automation for Event-B theories. *10th Rodin User and Developer Workshop collocated with the ABZ 2023 Conference*

Reflexive Event-B: Semantics and Correctness The EB4EB framework

Peter Rivière, Neeraj Kumar Singh, Yamine Aït-Ameur

INPT-ENSEEIHT/IRIT
University of Toulouse, Toulouse, France
{peter.riviere, neeraj.singh, yamine}@enseeiht.fr

The Event-B method enables correct by construction modelling of systems. It relies on set theory and first-order logic, to describe a series of refined system models expressed as a set of events modifying state variables. Invariants and theorems are introduced to express system properties submitted to the proof system associated to Event-B. While Event-B has proven its efficiency for the proof of this type of properties, it does not offer powerful means allowing the explicit description of properties other than safety and specific forms of reachability. Checking other properties like deadlock-freeness, liveness or event scheduling, etc. requires ad hoc modelling techniques and external tools such as model checkers or other proof systems. This paper presents *EB4EB*, a new modelling framework offering the capability to introduce formally defined Event-B extensions, in particular new proof obligations corresponding to new properties. It is based on meta-modelling techniques. It includes a theory (a meta-theory) modelling Event-B and offers means for explicit manipulation of Event-B features and an extension mechanism to explicitly formalise and prove other properties. This reflexive framework relies on a trace-based semantics of Event-B and introduces a set of Event-B theories defining data types, operators, well-defined conditions, theorems and proof rules to define Event-B constructs and their semantics. Deep and shallow instantiation mechanisms are set up to instantiate the obtained meta-theory. The EB4EB framework and its instantiation mechanisms are developed in Event-B using the Rodin platform ensuring correctness and internal consistency of the defined theories. Lamport's clock example, instantiating EB4EB in both shallow and deep mechanisms, is used to evaluate the proposed approach.

2.1 Introduction

Metamodelling is a standard approach in software engineering for describing abstractions of models and properties, as well as performing analysis to guarantee the quality of the developed models, rules, operations, and constraints. This approach is widely used in the field of model-driven engineering. Formal methods also offer frameworks to support meta-modelling facilities through the development of meta-theories, axiomatising metamodels, to represent higher-level reasoning concepts used in the specification, development, and verification of complex systems [Bertot & Castéran, 2010; Fallenstein & Kumar, 2015; Muñoz & Rushby, 1999; Sozeau et al., 2020].

Event-B [J.-R. Abrial, 2010] enables correct by construction modelling of systems. It relies on set theory and first-order logic (FOL), to describe a series of refined system models expressed as a set of events modifying state variables. Invariants and theorems are introduced to express system properties submitted to the proof system associated to Event-B. An integrated development environment (IDE), Rodin [J.-R. Abrial et al., 2010], enables model development as well as the automatic generation of proof obligations. The associated proof process ensures system consistency thanks to the proof system it supports. Rodin has been extended with several plugins including composition/decomposition [Silva & Butler, 2012], Theory plug-in [J.-R. Abrial et al., 2009; M. Butler & Maamria, 2010], code generation [Fürst et al., 2014; Méry & Singh, 2011] and so on. In particular, the theory plugin [J.-R. Abrial et al., 2009; M. Butler & Maamria, 2010] enables to extend the core concepts of Event-B by defining new data types, theories, and operators that can be used in Event-B models. In addition to the classical theories for lists, trees, graphs and reals, several other theories have been developed to support complex constructs like continuous features [Dupont et al., 2020, 2021] or domain knowledge ontologies [Mendil, Aït Ameur, et al., 2021; Mendil, Ameur, et al., 2021].

For checking system consistency and refinement, Event-B and its Rodin IDE rely on induction and provide automatically generated proof obligations for invariant preservation, variant progress, events feasibility, proof theorems, guard strengthening, refinement, and so on. To check additional properties such as deadlock freeness, liveness, reachability, event scheduling, and domain-specific properties, the designer must provide an adhoc Event-B model based on the core Event-B features or must rely on other modelling tools, such as model checkers and external interactive theorem provers. Indeed, there is no mechanism for encoding and reasoning on Event-B trace semantics. Moreover, Event-B does not offer the capability to manipulate Event-B concepts explicitly to formalise properties in a generic an reusable setting. *Therefore, performing advanced reasoning level by introducing new, reusable and automatically generated POs for any designed model is not yet possible.*

Our objective is to define a novel framework based on a reflexive formalisation, using Event-B, of a meta-theory allowing to manipulate Event-B concepts. This theory is enriched by new concepts allowing to formalise and generate new proof obligations formalising advanced and reusable reasoning mechanisms.

This paper extends our work presented in [Riviere et al., 2022a]. Our primary contribution is to present an EB4EB framework based on meta modelling concepts, including trace semantics of the core Event-B, for explicitly manipulating Event-B features and extending its reasoning mechanism to support other properties. In order to express the Event-B core modelling constructs, trace-based semantics, and new proof obligations for the EB4EB framework, a set of theories including data types, operators, well-defined conditions, theorems, and proof rules is developed. In addition, these theories enable manipulation of static and dynamic concepts of Event-B features as well as defining new proof obligations to support a *reusable* (defined once and for all) advanced reasoning level. Two instantiation mechanisms, deep and shallow, associated to these generic theories,

are introduced to exploit this correct by construction framework to support new Event-B model analyses. The formalised trace-based semantics is used to prove the soundness of the defined models analyses associated to the native and new generated Event-B POs allowed by EB4EB. Finally, we evaluate our approach on Lamport's clock example.

This paper is organised as follows. Section 2.2 presents Event-B modelling concepts, including refinement and proof obligations. Section 2.3 describes reflexive concepts, related work, and the EB4EB framework. The core concepts of Event-B are described in Sections 2.4 and 2.5 of the EB4EB framework. Section 2.6 describes the trace semantics and the correctness of proof obligations is provided in Section 2.7. Deep and shallow embeddings are described in Section 2.8. Section 2.9 describes Lamport's clock example, which is used to describe the application of EB4EB framework by applying the deep and shallow embeddings in Section 2.10. Section 2.11 presents the EB4EB reasoning mechanism, including a new set of proof obligations. The proof process related to the development of the clock model and deadlock reasoning extension is described in Section 2.12. Finally, Section 2.13 concludes the paper and discusses future work.

2.2 Event-B

Event-B [J.-R. Abrial, 2010] is a correct-by-construction method supporting the development of large and complex systems. Its formal modelling language is based on set theory and first-order logic (FOL) and relies on the definition of state variables characterising systems state and a set of events to model state changes. A system model is designed as a series of refined intermediate models starting from an abstract model. The main components of the Event-B modelling language are summarised below.

2.2.1 Event-B Contexts and Machines

`Contexts` (Tables 2.1(a)) describe all the static elements of the models through the definition of *carrier sets* s, *constants* c, *axioms* A and *theorems* T_{ctx}.

`Machines` (Table 2.1(b)) describe model behaviour. It consists of *Variables* x, *Invariants* $I(x)$, *Theorems* $T_{mch}(x)$ and *Variants* $V(x)$. It defines a transition system represented as a set of guarded events *evt* recording state changes using a Before-After Predicates (BAP). Events which decrease the variant are tagged as *convergent* otherwise they are *ordinary*. *Invariants* $I(x)$ and *Theorems* $T_{mch}(x)$ ensure safety properties, while *Variant* $V(x)$ ensures convergence properties for *convergent* events.

- *Refinements.* Refinement (see Table 2.1(c)) enables incremental design by introducing characteristics such as functionality, safety, reachability at different abstraction levels. It decomposes a *machine*, a state-transition system, into a more concrete model, by refining events and variables (simulation relationship). Introduction of gluing invariants preserves already proven properties.

Context	Machine	Refinement
CONTEXT Ctx SETS s CONSTANTS c AXIOMS A THEOREMS T_{ctx} END	MACHINE M^A SEES Ctx VARIABLES x^A INVARIANTS $I^A(x^A)$ THEOREMS $T_{mch}(x^A)$ VARIANT $V(x^A)$ EVENTS EVENT evt^A ANY α^A WHERE $G^A(x^A, \alpha^A)$ THEN $x^A :\mid BAP^A(\alpha^A, x^A, x^{A\prime})$ END END	MACHINE M^C REFINES M^A VARIABLES x^C INVARIANTS $J(x^A, x^C) \wedge I^C(x^C)$ EVENTS EVENT evt^C REFINES evt^A ANY α^C WHERE $G^C(x^C, \alpha^C)$ WITH $x^{A\prime}, \alpha^A : W(x^{A\prime}, \alpha^A, x^A, \alpha^C, x^C, x^{C\prime})$ THEN $x^C :\mid BAP^C(\alpha^C, x^C, x^{C\prime})$ END END
(a)	(b)	(c)

Table 2.1: Global structure of Context, Machines and Refinements

(1)	Theorems (THM)	$A \Rightarrow T_{ctx} A \wedge I^A(x^A) \Rightarrow T_{mch}(x^A)$
(2)	Initialisation (INIT)	$A \wedge G_A(\alpha^A) \wedge BAP^A(\alpha^A, x^{A\prime}) \Rightarrow I^A(x^{A\prime})$
(3)	Invariant preservation (INV)	$A \wedge I_A(x^A) \wedge G_A(x^A, \alpha^A)$ $\wedge BAP^A(x^A, \alpha^A, x^{A\prime}) \Rightarrow I^A(x^{A\prime})$
(4)	Event feasibility (FIS)	$A \wedge I_A(x^A) \wedge G^A(x^A, \alpha^A)$ $\Rightarrow \exists x^{A\prime} \cdot BAP^A(x^A, \alpha^A, x^{A\prime})$
(5)	Variant progress (VAR)	$A \wedge I^A(x^A) \wedge G^A(x^A, \alpha^A)$ $\wedge BAP^A(x^A, \alpha^A, x^{A\prime}) \Rightarrow \ V(x^{A\prime}) < V(x^A)$

Table 2.2: Machine Proof obligations

(6)	Event Simulation (SIM)	$A \wedge I^A(x^A) \wedge J(x^A, x^C) \wedge G^C(x^C, \alpha^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge BAP^C(x^C, \alpha^C, x^{C\prime})$ $\Rightarrow BAP^A(x^A, \alpha^A, x^{A\prime})$
(7)	Guard Strengthening (GRDS)	$A \wedge I^A(x^A) \wedge J(x^A, x^C)$ $\wedge W(\alpha^A, \alpha^C, x^A, x^{A\prime}, x^C, x^{C\prime})$ $\wedge G^C(x^C, \alpha^C) \Rightarrow G_A(x^A, \alpha^A)$

Table 2.3: Refinement Proof obligations

- *Proof Obligations (PO) and Property Verification.* Several POs are associated with the Event-B models shown in Table 2.2 and 2.3. These POs are generated automatically, and all of them must be successfully discharged to guarantee the correctness of an Event-B model, including refinements. Two additional POs related to refinement, guard strengthening and simulation, are required in our shallow modeling approach.

- *Core Well-definedness (WD).* The WD POs are associated to all built-in operators of the Event-B modelling language. Once proved, these WD conditions are used as hypotheses to prove other POs related to invariants, theorems, feasibility, etc.

2.2.2 Event-B extensions with Theories

In order to handle more complex modelling concepts not supported by native Event-B, an extension of Event-B based on mathematical definitions has been proposed in [J.-R. Abrial et al., 2009; M. J. Butler & Maamria, 2013]. This extension, like Isabelle/HOL [Nipkow et al., 2002] or PVS [Owre et al., 1992], allows to define new *theories* by introducing new data types, operators, theorems and proof rules. They can be further used in the core development of Event-B models.

- *Theory description.* Table 2.4 shows core modelling elements for developing new theories. The core modelling elements are classified in different clauses known as data types, operators, axiomatic definitions, axioms, theorems and proof rules. A theory can be parameterized by Type in the clause `TYPE PARAMETERS`. The description of the data-type, operator, theorems and proof rules use the type parameters. Data types (`DATATYPES` clause) can be defined with *constructors*, and each constructor can have some *destructors*. Note that a destructor can also have an inductive definition.

Theory
THEORY Th
IMPORT Th1, ...
TYPE PARAMETERS E, F, ...
DATATYPES
Type2(E, ...)
constructors
cstr1(p_1: T_1, ...)
OPERATORS
Op1 <nature> (p_1: T_1, ...)
well−definedness $WD(p_1, ...)$
direct definition D_1
AXIOMATIC DEFINITIONS
TYPES A_1, ...
OPERATORS
AOp2 <nature> (p_1: T_1, ...): T_r
well−definedness $WD(p_1, ...)$
AXIOMS A_1, ...
THEOREMS T_1, ...
END

Table 2.4: Global structure of Event-B Theories

A theory may contain several *operators* of different nature (<nature> tag), expression or predicate. These new defined operators extend the capabilities of the Event-B core language and can be used directly in core modelling components like expression and predicate. Operators may be defined in two ways. First, explicitly in the `direct definition` clause where the operator is equivalent to an expression, and second, axiomatically in the `AXIOMATIC DEFINITIONS` clause where the behaviour of the operator is expressed by a set of axioms. Last, a theory defines a set of theorems proven with the help of defined operators and axioms.

Many theories have been defined for sequences, lists, groups, reals, differential equations, and so on [M. J. Butler & Maamria, 2013; Dupont et al., 2021].

- *Well-definedness (WD) in Theories.* This useful clause associates *well-definedness* (WD) conditions to each operator defined in a theory. This condition restricts the use of an operator to its licit parameters (partial definitions). In particular, when a function is denoted as operator, this condition defines the domain of this function as well-definedness additional constraints. When the defined operator is used, a WD proof obligation is generated and must be discharged to ensure the correctness of the modeled specification as well as defined properties.

All the WD POs and theorems are proved using the Event-B proof system.

- *Event-B proof system and its IDE Rodin.* Rodin[1] is an open-source Eclipse-based Integrated Development Environment for modelling in Event-B. It offers resources for model editing, automatic PO generation, project management, refinement

[1]Rodin Integrated Development Environment http://www.event-b.org/index.html

and proof, model checking, model animation and code generation. The theories extension for Event-B is available as a plug-in. Theories are tightly integrated in the proof process. Depending on their definition (direct or axiomatic), operator definitions are expanded either using their direct definition (if available) or by enriching the set of axioms (hypotheses in proof sequents) using their axiomatic definition. Theorems can be imported as hypotheses and used in proofs just like any other theorem. The proof system is partially automatic, the other parts are interactive. Many tools are available to help with proof like predicate provers or SMT solvers.

2.3 The EB4EB Framework

2.3.1 Motivation

As mentioned in the introduction, Event-B extensions are not possible as the modelling language does not offer the capability to manipulate Event-B concepts as first-order objects. Meta-modelling features are not available in the core Event-B modelling language. Offering meta-modelling capabilities is the main idea of the EB4EB framework.

Embedding modelling language constructs in another modelling language is well accepted by the model-driven engineering. When this embedding is realised in the same modelling language, it is qualified as reflexive. Two embedding techniques have been identified: deep and shallow embeddings. Deep embedding describes explicitly the semantics and syntax of the source language in the logic of the host language, whereas shallow embedding simply expresses by translation the semantics of the source language in the semantics host language [Boulton et al., 1992] (i.e. here the translator carries the semantics). Both approaches have their pros and cons. Deep embedding requires more modelling effort to address structural and semantic elements of the source language. As a result, while this approach may be difficult to grasp and tedious, it offers full access, in the logic of the host modelling language, to the elements of the source modelling language for formal verification. On the other hand, the shallow embedding approach is straightforward and easy to use once the semantics of the source modelling language is directly formalised in the modelling language enconding the transformation. It leads to limited access to the source modelling language constructs for formal verification, in particular when tracing verification results (e.g. counter-examples). Munoz et al. [Muñoz & Rushby, 1999] proposed a structural embedding approach in which only the language structure is deep/shallow embedded in the host logic and the source language expression is replaced by the host logic expression.

In order to design a formal setting for defining Event-B extensions, the proposed EB4EB framework defines a reflexive embedding on Event-B in an Event-B theory. Before entering into the details of the EB4EB framework, we review some approaches of the literature which addressed the problem of embedding formal modelling languages in other formal modelling languages.

2.3.2 Related work

Several modelling languages use a reflexive approach to handle higher-order modelling concepts and their manipulation for improving reasoning mechanisms and other advanced level modelling features. Riccobene et al. [Riccobene & Scandurra, 2004] proposed the ASM-Metamodel (AsmM) for manipulating Abstract State Machine (ASM) [Börger & Stärk, 2003] concepts like abstract machines, signatures, terms, rules, and so on. The developed API offers to express analyses and ASM tool extensions, such as requirements validation [Scandurra et al., 2012], model checking [Arcaini et al., 2010a], animation [Bonfanti et al., 2018; Carioni et al., 2008], flattener for the ASMETA framework [Arcaini et al., 2018], and reviewing ASM model by meta property verification [Arcaini et al., 2010b]. Bicarregui et al. [Bicarregui & Ritchie, 1991] proposed reflexive concepts for VDM [C. B. Jones, 1986] in a mathematical reasoning environment MURAL [C. B. Jones et al., 1991] to provide modelling and reasoning capabilities for higher-order concepts. The Event-B API available in Rodin tools enables the development of core plug-ins such as model checker and animation ProB [Leuschel & Butler, 2003], code generation [Fürst et al., 2014; Méry & Singh, 2011], extending modelling features [T. S. Hoang et al., 2017; Silva & Butler, 2012].

The reflexive approach is not limited to modelling languages; other formal methods approaches related to type theory use it to manipulate their syntax and higher-order modelling concepts. For example, the reflexive approach is proposed for Agda [Stump, 2016], Lean [Ebner et al., 2017], and Coq [Anand et al., 2018]. Moreover, this approach can be used in functional programming languages such as MetaML [Taha & Sheard, 1997], and Template Haskell [Sheard & Jones, 2002]. In [Sozeau et al., 2020], the authors proposed a framework in the MetaCoq project to define the semantic of Coq in order to support the certified meta-programming environment. This framework aided in the development of CertiCoq [Anand et al., 2017], a certified compiler of Coq. The reflection principle is implemented in Isabelle/HOL [Fallenstein & Kumar, 2015] to express HOL models as well as reasoning mechanisms in order to describe complex systems with self-replacement functionality. Similarly, Mitra et al. [Mitra & Archer, 2005] proposed the reflection mechanism in PVS based on theories and templates to generalise proofs and make them highly reusable using *strategies* concepts for proving abstraction relation between automata.

Regarding the B method [J.-R. Abrial, 1996], Munoz et al. [Muñoz & Rushby, 1999] proposed a formalisation in the higher-order theorem prover PVS [Owre et al., 1992]. In the same vein, Event-B is also formalised to ensure the correctness of modelling and reasoning concepts. Bodeveix et al. [Bodeveix & Filali, 2021] proposed context formalisation in order to prove the theorems expressing properties on Event-B models. Schneider et al. [Schneider et al., 2011] proposed the core semantics of Event-B, including refinement, in CSP [Hoare, 1978], which is based on trace semantics. In [Farrell et al., 2017], the authors proposed the Event-B formalisation to express the theory of institution, but it is not tool supported. Event-B modelling constructs are also formalised in Coq to express Event-B traces, and a set of theorems is proved in Coq to ensure the correctness of proof obligations.

Our approach provides a homogeneous framework for using the Event-B machine concept as a first-class object in models, similar to Coq and HOL, and the user does not require to use different semantic frameworks to manipulate it, as described in CSP [Schneider et al., 2011] and Coq [Castéran, 2021]. Thus, our work is free of semantic heterogeneity constraints, which could reduce embedding correctness. Our method can handle two types of semantics: *native* and *axiomatic*. The first *native* semantics deal with the core concept of state-based modelling and refinement, while the second *axiomatic* semantics deal with first-order logic. These semantical representations play a central role in analysing and ensuring any complex systems that have been built correctly in a non-intrusive manner.

In [Castéran, 2021; T. S. Hoang & Abrial, 2011; Leuschel & Butler, 2003; Schneider et al., 2011, 2014], trace-based semantic was used to validate the Event-B modelling and analysis concepts. Most of this work emphasises on Event-B embedding in other formalisms or APIs to ensure the Event-B semantics, whereas our work provides a set of operators, axioms, and theorems developing a theory (a meta-theory) to manipulate and extend the core concepts of Event-B while preserving the semantics in the same formal modelling language. Moreover, our framework allows expressing some important properties such as liveness, deadlock freeness, event scheduling and so on. In addition, this framework also provides a set of operators to represent the Event-B trace semantics in order to ensure the correctness of modelling features, functionalities, properties, and proof obligations. This framework enables non-intrusive analysis for checking the correctness of complex systems. As far as we know, this is the first *reflexive* framework for the Event-B method to analyse a system systematically.

2.3.3 The EB4EB framework

The EB4EB framework is based on first-order logic and set theory, which enable a simple and easy mechanism for exporting Event-B core concepts, including semantics, in other formalisms without redoing the entire work, and its use does not impose many well-typed proof obligations. This framework supports two types of proof processes: the first is operational with axiomatic semantics in the Event-B context, and the second is induction to handle machine mechanisms similar to Event-B native proof process.

The EB4EB framework defines a set of generic and reusable Event-B theories formalising all the concepts available in an Event-B model. It uses an algebraic style with concept types, constructors, operators and a set of axioms and theorems providing their properties. This theory is instantiated to define specific Event-B models. Two instantiation mechanisms have been defined: *deep* and *shallow*. Fig. 2.1 depicts the architecture of this framework. The core theory (Fig 2.1.A) models the core Event-B method. The correctness of the defined proof obligations with respect to the provided-trace based semantics is supported by Fig. 2.1.B and Fig. 2.1.C. Last, the extensions of the framework and their correctness are presented in the theories of Fig. 2.1.D.

In the following, we provide a detailed presentation of this framework. The formalisation of the model the constructs (Section 2.4), Event-B proof obligations

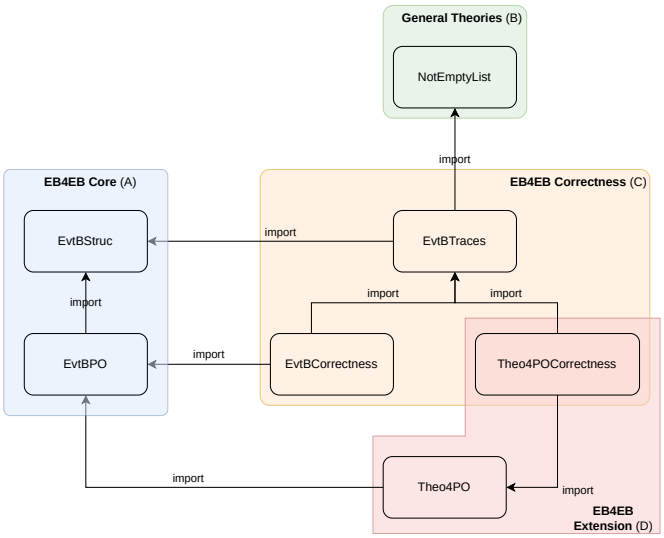


Figure 2.1: Architecture of the theories

(Section 2.5) and the semantics and the theorem guaranteeing the correctness of the approach (Sections 2.6 and 2.7) are presented. The two instantiation mechanisms are presented in Section 2.8.

2.4 EB4EB structure (see Fig. 2.1.(A))

This section introduces the `EvtBStruc` Event-B theory of EB4EB (see Fig. 2.1.(A)) dedicated to the definition of the structure of an Event-B model. It includes data types constructors and well-structured machine.

2.4.1 Data types and constructors

In order to model states and events (transitions), the two main components of a state-transition system, the Event-B meta-theory `EvtBStruc` introduces, in the `TYPE PARAMETERS` clause, two polymorphic type parameters, represented as carrier sets, $STATE$ and $EVENT$ (see Listing 2.1). The type parameter $STATE$ is used to represent a set of variables. An explicit description of each variable is not required at this abstract level. Indeed, the type parameter $STATE$ abstracts the state as a Cartesian product of all variables. At the instantiation step, this abstract type is replaced with concrete variables of the considered Event-B model. The second type parameter $EVENT$ is used to abstract the label of events.

These type parameters are used in the definition of a new datatype $Machine$ in the `DATATYPES` clause. A single constructor `Cons_machine` is defined in the `CONSTRUCTOR` clause associated with destructors to represent and access various

constituents of Event-B components. The following destructors are defined.

- *Event* - a set of machine events;
- *State* - a set of machine states;
- *Init* - an initialisation event;
- *Progress* - a set of progress events;
- *AP* - the after-predicate defining the initialisation state;
- *Grd* - a set of event guards as a pair made of allowed state and an event;
- *BAP* - a set of before after-predicates as a triple made of an event and before and after states;
- *Inv* - machine invariants as a set of licit states;
- *Thm* - machine theorems as a set of licit states;
- *Variant* - machine variants as a pair associating an integer to a state;
- *Ordinary* - a set of ordinary events, i.e. events which do not constrain the variant;
- *Convergent* - a set of events decreasing a variant.

```
THEORY EvtBStruc // Part 1
  TYPE PARAMETERS EVENT, STATE
  DATA TYPES
    Machine(STATE, EVENT)
    CONSTRUCTORS
      Cons_machine(
        Event : ℙ(EVENT),
        State : ℙ(STATE),
        Init : EVENT,
        Progress : ℙ(EVENT),
        AP : ℙ(STATE),
        Grd : ℙ(EVENT × STATE),
        BAP : ℙ(EVENT × (STATE × STATE)),
        Inv : ℙ(STATE),
        Thm : ℙ(STATE),
        Variant : ℙ(STATE × ℤ),
        Ordinary : ℙ(EVENT),
        Convergent : ℙ(EVENT)
      )
```

Listing 2.1: Machine Data-type Definition

2.4.2 Well Structured Machine

The `DATATYPES` clause defines a constructor and destructors to access the Event-B modelling components. The above-defined constructors and destructors contain typing information only. Therefore, they may lead to ill-defined datatype definitions. It is necessary to associate well-definedness (WD) conditions that restrict their use in consistent cases. For example, the BAP destructor is a relation

between events and states, but the initialisation event is not concerned by this before-after relation and shall be excluded from the set of events involved in a BAP.

In order to avoid such ill-defined typing definitions, we introduce a set of new operators in Listing 2.2 each of which is equipped with WD conditions. In Listing 2.2, the first well-defined operator `BAP_WellCons` is declared with one argument machine m, and its direct definition shows that all events in the domain of the BAP relation are progress events, implying that the event set contains no initialisation event. The next well-defined operator `Grd_WellCons` is also defined with single machine m argument. Its direct definition states that all events in the domain of the Grd relation are progress events. To check the well definedness condition of the `Event` operator, the `Event_WellCons` is declared. Its direct definition states that the union of the progress and initialisation events equals the machine events.

The direct definition of the next `Variant_WellCons` operator shows that all the states belonging to the variant states are convergent and identified from the set of invariant states, i.e. each variant state element is associated with an integer. Note that the variant is a total function in the invariant states. The direct definition of the `Tag_Event_WellCons` operator shows that the union of convergent and ordinary events equals mutually exclusive machine events and the initialisation event is an ordinary event.

The last `Machine_WellCons` operator is important. It collects all the well-definedness conditions of all the defined operators. Its direct definition is the conjunction of all other well-defined operators. It represents the global well-defined condition associated with an Event-B machine m.

```
//THEORY EvtBStruc  Part 2
  OPERATORS
    BAP_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        dom(BAP(m)) = Progress(m)
    Grd_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        dom(Grd(m)) = Progress(m)
    Event_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        partition(Event(m), {Init(m)}, Progress(m))
    Variant_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ◁ Variant(m) ∈ Inv(m) → ℤ
    Tag_Event_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        partition(Event(m), Ordinary(m), Convergent(m))∧
              Init(m) ∈ Ordinary(m)
    Machine_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        BAP_WellCons(m)∧
        Grd_WellCons(m)∧
        Event_WellCons(m)∧
        Tag_Event_WellCons(m)∧
        Variant_WellCons(m)
```

Listing 2.2: Machine Well Constructed Operators

2.5 EB4EB Proof obligations (see Fig. 2.1.(A))

Once Event-B models are structurally well built, semantics can be addressed. This section presents a set of proof obligations formalised as operators of the `EvtBPO` theory (see Fig. 2.1.(A)) of the EB4EB framework. These operators express and help to discharge the generated proof obligations given in Section 2.2, such as `INV`, `FIS`, `NAT` and `VAR`. Their definitions are inductive as they apply to the initialisation and then to all other events. The formalisation relies on an encoding of FOL expressions set comprehension. Below, we formalise all POs at the meta-theory level.

2.5.1 Feasibility Proof Obligation (FIS)

Principle

The objective of this proof obligation rule is to ensure that when the guard of an event holds, its BAP allows to reach the next state, i.e. the action defined by the BAP is feasible. It is defined as,

$$\frac{M \vdash \forall i, \alpha, x \cdot G_i(\alpha, x) \land I(x) \Rightarrow (\exists x' \cdot BAP_i(\alpha, x, x'))}{M \vdash FIS}$$

FIS Operators formalised in EB4EB

The feasibility rule is encoded in the Event-B meta-theory presented in Listing 2.3. We defined three operators `Mch_FIS_Init`, `Mch_FIS_One_Ev`, and `Mch_FIS`. The first two operators represent the base case and induction case for the feasibility PO, respectively. The first operator is declared with one argument machine m, and its direct definition for the base case ensures that the intersection of machine invariants ($Inv(m)$) and machine after-predicates ($AP(m)$) should not be empty. The second operator is declared with two arguments machine m and event e. The direct definition for the induction case ensures that the invariants and guards of the progress event e are a subset of the domain of the BAP of e. The last operator checks the machine feasibility by induction. Its direct definition shows that the machine is feasible at initialisation (base case) and for all progress events (inductive case).

```
Mch_FIS_Init predicate (m : Machine(STATE, EVENT))
  direct definition
    Inv(m) ∩ AP(m) ≠ ∅
Mch_FIS_One_Ev predicate (m : Machine(STATE, EVENT) , e : EVENT)
  well−definedness e ∈ Progress(m)
  direct definition
    Inv(m) ∩ Grd(m)[{e}] ⊆ dom(BAP(m)[{e}])
Mch_FIS predicate (m : Machine(STATE, EVENT))
  direct definition
    Mch_FIS_Init(m)∧
    (∀e · e ∈ Progress(m) ⇒ Mch_FIS_One_Ev(m, e))
```

Listing 2.3: Feasibility proof obligation operators

2.5.2 Invariant Proof Obligation (INV)

Principle

Invariant proof obligation rule ensures that each event of a machine preserves the invariant. It uses two abbreviations to increase it readability. It is defined as,

$$initCase_{INV} \equiv \forall \alpha, x' \cdot AP(\alpha, x') \Rightarrow I(x')$$

$$inducCase_{INV}(i) \equiv \forall \alpha, x, x' \cdot G_i(\alpha, x) \wedge BAP_i(\alpha, x, x') \wedge I(x) \Rightarrow I(x')$$

$$\frac{M \vdash initCase_{INV} \qquad M \vdash \forall i \cdot i \in 1..n \Rightarrow inducCase_{INV}(i)}{M \vdash INV}$$

Here, for an Event-B machine containing n progress events, $G_i(x, \alpha)$ and $BAP_i(\alpha, x, x')$ represent the guard and the before-after predicate of the event $e_i(i \in 1..n)$

INV Operators in EB4EB

The invariant proof obligation rule is also formalised in Event-B meta-theory presented in Listing 2.4. Three predicate operators, `Mch_INV_Init`, `Mch_INV_One_Ev` and `Mch_INV`, define the initialisation, the induction case of the invariant PO for a single event e, and the induction case of invariant properties for all progress events, respectively. The direct definition of the first operator states that the machine after predicate ($AP(m)$) is a subset of machine invariant. The next operator, `Mch_INV_One_Ev`, ensures that the BAP of progress event e preserves the invariants if guards and invariants are held before. The last operator is defined to check each event preserves the machine invariants by induction. The direct definition of this operator shows that the machine invariant is preserved at initialisation and for all progress events, i.e. invariant for all machine events.

Mch_INV_Init *predicate* $(m : Machine(STATE, EVENT))$
  direct definition
    $AP(m) \subseteq Inv(m)$
Mch_INV_One_Ev *predicate* $(m : Machine(STATE, EVENT), e : EVENT)$
  well−definedness $e \in Progress(m)$
  direct definition
    $BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$
Mch_INV *predicate* $(m : Machine(STATE, EVENT))$
  direct definition
    $Mch\_INV\_Init(m) \wedge$
    $(\forall e \cdot e \in Progress(m) \Rightarrow Mch\_INV\_One\_Ev(m, e))$

Listing 2.4: Invariant proof obligation operators

2.5.3 Natural Variant Proof Obligation (NAT)

Principle

The objective of this proof obligation rule is to ensure that a proposed numeric variant is a natural number under the guards of each convergent event. The variant proof obligation rule is defined as,

$$\frac{M \vdash \forall i, \alpha, x \cdot e_i \in convergent \wedge G_i(\alpha, x) \wedge I(x) \Rightarrow v(x) \in \mathbb{N}}{M \vdash NAT}$$

Natural Variant in EB4EB

Listing 2.5 shows two operators, `Mch_NAT_One_Ev` and `Mch_NAT`, to define a variant for an event e as a natural number and that all convergent events have a natural number as a variant, respectively. Their direct definitions are provided below.

Mch_NAT_One_Ev *predicate* $(m : Machine(STATE, EVENT), e : EVENT)$
  well-definedness $e \in Convergent(m)$
  direct definition
    $Variant(m)[Inv(m) \cap Grd(m)[\{e\}]] \subseteq \mathbb{N}$
Mch_NAT *predicate* $(m : Machine(STATE, EVENT))$
  direct definition
    $Variant(m)[Inv(m) \cap Grd(m)[Convergent(m)]] \subseteq \mathbb{N}$

Listing 2.5: Variant proof obligation operators

2.5.4 Variant decrease Proof Obligation (VAR)

Principle

This proof obligation rule ensures that each convergent event decreases the proposed numeric variant. This proof obligation rule is defined as,

$$\frac{\begin{array}{c} M \vdash \forall i, \alpha, x, x' \cdot e_i \in Convergent \wedge G_i(\alpha, x) \\ \wedge I(x) \wedge BAP(x, x', \alpha) \Rightarrow v(x') < v(x) \end{array}}{M \vdash VAR}$$

Variant decrease in EB4EB

Two new operators, `Mch_VARIANT_One_Ev` and `Mch_VARIANT`, are declared to represent convergent properties in Listing 2.6. The `Mch_VARIANT_One_Ev` definition guarantees that if invariants and guards hold, then the BAP decreases the variant associated with the convergent event e. The WD clause defines other well-defined operators to ensure the correctness and the required WD conditions for the variants. Similarly, the operator `Mch_VARIANT` generalises the definition of convergence, it checks the required properties for all convergent events of machine m.

Mch_VARIANT_One_Ev *predicate* $(m : Machine(STATE, EVENT), e : EVENT, s : STATE)$
  well-definedness $Variant\_WellCons(m), Mch\_INV\_One\_Ev(m, e), e \in Progress(m), e \in Convergent(m), s \in Inv(m), s \in Grd(m)[\{e\}]$
  direct definition
    $\forall sp \cdot sp \in BAP(m)[\{e\}][\{s\}] \Rightarrow (Inv(m) \lhd Variant(m))(s) > (Inv(m) \lhd Variant(m))(sp)$
Mch_VARIANT *predicate* $(m : Machine(STATE, EVENT))$
  well-definedness $Variant\_WellCons(m), Mch\_INV(m), BAP\_WellCons(m), Tag\_Event\_WellCons(m), Event\_WellCons(m)$
  direct definition
    $\forall e, s \cdot e \in Event(m) \wedge e \in Convergent(m) \wedge s \in State(m) \wedge s \in Inv(m) \wedge s \in Grd(m)[\{e\}] \Rightarrow Mch\_VARIANT\_One\_Ev(m, e, s)$

Listing 2.6: Variant decrease proof obligation operators

2.5.5 Theorem THM

Principle

This rule ensures that a context or machine theorem can be proven. Theorems are important for simplifying some proofs. The theorem proof obligation rule states that theorems are deduced from invariants, it is defined as,

$$\frac{M \vdash \forall x \cdot I(x) \Rightarrow Thm(x)}{M \vdash THM}$$

Theorem THM in EB4EB

The declared operator `Mch_THM` consists of one argument machine m, and its direct definition shows that the invariants are a subset of theorems in Listing 2.7.

```
Mch_THM predicate (m : Machine(STATE, EVENT))
  direct definition
    Inv(m) ⊆ Thm(m)
```

Listing 2.7: Theorem proof obligation operator

2.5.6 Proof Obligation Generation

Listing 2.8 shows the most important predicate operator `check_Machine_Consistency`. When this predicate is used as a theorem at the instance level, it allows generating automatically all possible POs. This predicate expresses the proof obligations as the conjunction of all the proof obligations related to Event-B constituents, expressed using the predicate operators previously defined. By the WD condition associated to this operator, it only applies to well-built machines, as defined by the properties described in Section 2.4.

```
check_Machine_Consistency predicate
  (m : Machine(STATE, EVENT))
    well−definedness Machine_WellCons(m)
    direct definition
      Mch_INV(m)∧
      Mch_FIS(m)∧
      Mch_NAT(m)∧
      Mch_VARIANT(m)∧
      Mch_THM(m)
```

Listing 2.8: Machine Proof Obligation

Note that the POs generation mechanism described in this section can be seen as another approach to generating them. Instead of using the PO generator of the RODIN platform, one can use the WD and Theorem proof obligations obtained by the use of the EB4EB theories.

2.6 Trace's semantics of Event-B

In this section, we present a trace-based semantics for Event-Machines and then relate it to the proof obligations formalised in the previous section.

2.6.1 Event-B traces

For a given machine M, a sequence of states $tr = s_0 \mapsto s_1 \mapsto \ldots \mapsto s_n$ describes a trace of machine M iff,

1. tr contains at least one state. A trace includes at least the initialisation event;
2. s_0 is the initial state satisfying the After Predicate (AP) of the initialisation event;
3. for each successive states s_i, s_{i+1} of the sequence tr, there exists a progress event e such that
   - s_i satisfies the guard of e and
   - $s_i \mapsto s_{i+1}$ satisfies the Before-After Predicate of the event e

This notion of trace is formalised in an Event-B theory.

2.6.2 Trace's Semantics in EB4EB

Non-empty lists (see Fig. 2.1.(B))

To formalise the trace semantics of Event-B, we develop another theory `NotEmptyList` as shown in Listing 2.9. This theory relies on the list data type and declares a List type `T`. The data type `NotEmptyList` has two constructors, one for describing the base case and one for describing the inductive case. The base case constructor has only one element in the list, while the inductive case constructor has one element at the head of the list and a tail to represent a list of other elements. In this theory, we define the `first` operator, which takes a list as an argument and returns the first element of the list. The list theory is used to represent a trace as a list of states, where the list is the inverse of the state sequence. If n is the size of the list, then the state s_i is at the index $n - i$ of the list, and the state s_n is at the head of the list.

```
THEORY NotEmptyList // Part 1
  TYPE PARAMETERS T
  DATA TYPES
    NotEmptyList(T)
    CONSTRUCTORS
      cons(el : T,
           next : NotEmptyList(T))
      base_case(base : T)
  OPERATORS
    first expression (
        l : NotEmptyList(T))
      recursive definition
        case l :
          cons(t, q) => t
          base_case(t) => t
```

Listing 2.9: An inductive list

A theory of Event-B traces (See Fig. 2.1.(C))

The formalisation of Event-B machine traces is proposed in the theory of Listing 2.10. This theory imports the two developed theories, `EvtBStruc` and `NotEmptyList` allowing to access to the Event-B features already formalised and to lists respectively. It defines two operators: `IsATrace` and `IsANextState`. The first operator is a predicate that checks if a trace tr of a machine m is a trace of this machine. It is defined inductively on the trace structure and refers to the

second operator `IsANextState` to check that every state in the trace is a correct next state. The direct definition of this operator states that there exists a progress event e such that s belongs to the guard of event e and $s \mapsto sp$ satisfies the Before-After predicates of event e.

```
THEORY EvtBTraces IMPORT EvtBStruc , NotEmptyList
  TYPE PARAMETERS STATE, EVENT
  OPERATORS
    IsATrace predicate (tr : NotEmptyList(STATE) ,m : Machine(STATE, EVENT))
      recursive definition
        case tr :
          base_case(s) => s ∈ AP(m)
          cons(sp, q) => IsANextState(sp, first(q), m) ∧ IsATrace(q, m)
    IsANextState predicate (sp : STATE ,s : STATE ,m : Machine(STATE, EVENT))
      direct definition
        ∃e · e ∈ Progress(m)∧
            s ∈ Grd(m)[{e}] ∧ s ↦ sp ∈ BAP(m)[{e}])
```

Listing 2.10: Inductive trace of Event-B

2.7 EB4EB Correctness (see Fig. 2.1.(B,C))

The correctness of the expression of each proof obligation defined in the Event-B Theory presented in section 2.5 is established at the trace semantics level of Section 2.6. Correctness is stated according to the definition of the proof obligations available in the Event-B reference book [J.-R. Abrial, 2010]. The EB4EB is set up for this purpose. A theorem ensuring that the defined proof obligation entails the property on the traces is formalised and proved.

2.7.1 Principle (See Fig.2.1.(C))

To establish correctness, we introduce another Event-B theory *EvtBCorrectness* that imports the *EvtBPO* and *EvtBTraces* two theories (See Fig.2.1.(C)). It includes a set of theorems stating that the expressed PO implies that the PO property holds on the trace.

We demonstrate our approach using invariant proof obligation. In this case, we ensure that invariant is a valid machine invariant if any state in a trace satisfies it.

2.7.2 Correctness of the Invariant PO formalised in EB4EB

In order to define the theorem ensuring the correctness of the definition of the Invariant PO defined in Section 2.5.2, the *NotEmptyList* theory has been extended with the `AllAre` operator (see Listing 2.11) checking that the predicate *pred* (expressed using set-theoretical "belongs to" relationship) holds for all the elements of a non-empty list l (used later to model a trace). It is recursively defined on the elements of a non-empty list

```
// THEORY NotEmptyList Part 2
OPERATORS
    AllAre predicate (l : NotEmptyList(T) ,pred : ℙ(T))
      recursive definition
```

```
        case 1:
          base_case(el) => el ∈ pred
          cons(t, q) => t ∈ pred ∧ AllAre(q, pred)
END
```

Listing 2.11: New Operator on the list

Listing 2.12 presents the proved theorem for checking the Invariant's PO correctness. It states that if for all traces tr of any well structured Event-B machine m satisfying the invariant PO Mch_INV of Listing 2.4, then the invariant property $inv(m)$ of machine m holds in any state of the trace tr. The previous $AllAre(tr, inv(m))$ operator is used for this purpose.

```
THEORY EvtBCorrectness
  IMPORT EvtBTraces, EvtBPO
  TYPE PARAMETERS STATE, EVENT
  THEOREMS
    thm1:
      ∀m, tr · m ∈ Machine(STATE, EVENT)∧
        Machine_WellCons(m) ∧ IsATrace(tr, m) ∧ Mch_INV(m)
          ⇒ AllAre(tr, Inv(m))
END
```

Listing 2.12: Theorem of correction of the proof obligation

The theorem of Listing 2.12 has been proved using the proof system of the Rodin platform. The main proof step of this proof is a case-based proof on the structure of the trace by unfolding the definition of the *AllAre* operator. Following this approach, similar theorems have been defined and proved for the remaining proof obligations.

2.8 Modelling Event-B machines in EB4EB

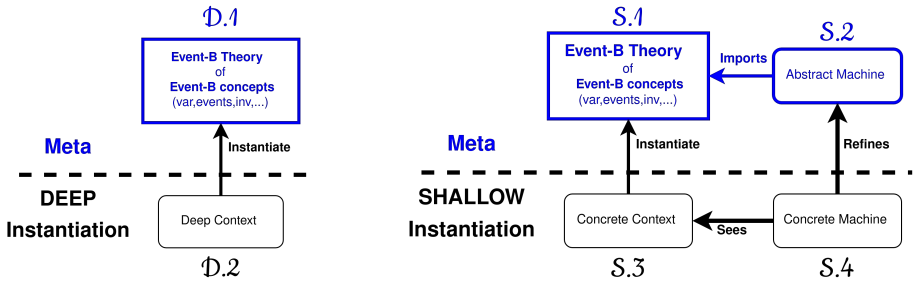
In the previous section, we presented the theory axiomatising Event-models. The well-definedness conditions and the relevant theorems introduced allows to check the correctness of the specific models defined as instances of this generic theory. Below, we describe the defined instantiation mechanisms.

2.8.1 Instantiation Methodology

Two instantiation mechanisms, depicted in Fig. 2.2, are envisioned: *deep* and *shallow* modelling.

Deep modelling consists in the definition of the various elements composing a machine conforming to the `EvtBStruc`. At instantiation, these definitions are provided in an Event-B context, where witnesses for type parameters STATE and EVENT are provided. Deep instantiation mechanism is recommended when manipulating and/or reasoning on Event-B features as first order objects is required. In particular, this mechanisms allows for the extension of Event-B to formalise and prove other properties not available in core Event-B.

Shallow modelling is close to shallow embedding [Boulton et al., 1992]. It relies on an abstract Event-B machine and model instances are defined as a refinement of this machine. This instantiation mechanism is recommended when the model can be expressed and proved. The benefit of this mechanism is the use of the



(a) Deep Modelling (b) Shallow Modelling

Figure 2.2: EB4EB framework

refinement operation associated to the built-in induction principle available in core Event-B.

To compare both mechanisms, one can state that deep instantiation mechanism offers the capability to extend the Event-B method to handle other type of properties not available in core Event-B while shallow instantiation mechanism allows for the use of the refinement operation offered by core Event-B.

In both mechanisms, the defined operators, when invoked in a machine or a context, automatically generate well-definedness and theorem POs that must be proven in order to ensure machine consistency. These two instantiation mechanisms are detailed below.

2.8.2 Deep modelling based instantiation (see Fig. 2.2a)

This instantiation mechanism consists in defining an instance of the data type *Machine* in an Event-B context ($\mathcal{D}.2$) where the generic type parameters of the meta-theory are instantiated by sets describing machines state variables and events. All the Event-B constructs described in the `EvtBStruc` Meta-theory such as invariants, theorems, event guards and before after predicate and so on are defined in the form of axioms. Consistency is ensured by the introduction of the theorem `check_Machine_Consistency` corresponding to the *predicate* consistency operator of the Meta-theory. It generates two kinds of proof obligation: first the well-definedness PO to ensure that the Event-B machine is well built, and second the PO related to the Event-B machine consistency such as invariant preservation, and variant decreasing corresponding to the theorem proof obligation. Both proof obligations must be proved (see Listing 2.13). These obtained POs are proved using the Event-B Rodin theorem prover as well as the other supporting theorem provers.

Listing 2.13 represents the skeleton of a context representing an instantiated machine in which each axiom characterises different components of the Event-B machine for reasoning and analysis using EB4EB framework. Our goal is to generate an instance machine automatically, thus the skeleton of the context model

is fixed in a sequence of axioms. These sequences are: $axm1$ - to define partitions set Ev of machine events; $axm2$ - to define a machine m as an instance of theory's data type by instantiating $EVENT$ as an event set Ev, and $STATE$ as a cartesian product of variables type; $axm3$ - to set event accessor as partition event set Ev; $axm4$ - to set state accessor as a cartesian product variables type; $axm5$ - to set initial event of a machine m; $axm6$ - to define a set of progress events; $axm7$ - to define a set of machine after-predicates; $axm8$ - to define a set of machine guards; $axm9$ - to define a set of machine before-after predicates; $axm10$ - to define a set of machine invariants; $axm11$ - to define a set of machine theorems; $axm12$ - to define a set of a machine variants; $axm13$ - to define a set of ordinary events; $axm14$ - to define a set of convergent events. Finally, the theorem $check\_Machine\_Consistency$ is defined to ensure that the machine m is well-constructed and consistent by satisfying all associated proof obligations. A THM PO is generated for this theorem.

```
CONTEXT Deep
SETS Ev, ...
CONSTANTS mch, ...
AXIOMS
  axm1: partition(Ev, ...)
  axm2: mch ∈ Machine(..., Ev)
  axm3: Event(mch) = Ev
  axm4: State(mch) = ...
  axm5: Init(mch) = ...
  axm6: Progress(mch) = {...}
  axm7: AP(mch) = {...}
  axm8: Grd(mch) = {...}
  axm9: BAP(mch) = {...}
  axm10: Inv(mch) = {...}
  axm11: Thm(mch) = {...}
  axm12: Variant(mch) = {...}
  axm13: Ordinary(mch) = {...}
  axm14: Convergent(mch) = {...}
THEOREMS
  thm1: check_Machine_Consistency(mch)
END
```

Listing 2.13: A skeleton of a machine in the deep modelling

2.8.3 Shallow modelling based instantiation (see Fig. 2.2b)

As mentioned above, this mechanism introduces a context and an abstract machine to be refined by the instance Event-B model. Listings 2.14 and 2.15 show these context and machine of the abstract model for shallow instantiation. The context defines the sets Ev and St as instances of the type parameters for events and states. For this purpose, a constant mch is introduced as a member of $Machine(St, Ev)$.

```
CONTEXT ShallowGenCtx
SETS St, Ev
CONSTANTS mch
AXIOMS
  axm1: mch ∈ Machine(St, Ev)
END
```

Listing 2.14: A static element of abstract machine ($\mathcal{S}.2$)

In the abstract machine model, two variables s and $InitDone$ (for scheduling event triggering) are declared in the $inv1-inv2$ invariant clauses. These variables are set in the INITIALISATION event. $inv3$ ensures that the invariant $Inv(mch)$ of the instance model is satisfied. In this model, we define three events: `Do_Init`, `Do_Ordinary`, and `Do_Convergent` whose actions modify the state using the AP (for initialisation) and BAP operators ($act1$). The first event is used to initialise state variables in action ($act1$). Its guards ensure that $InitDone$ is $FALSE$ ($grd1$), and the feasibility and invariants hold for the $Init$ event ($grd2$). The `Do_Ordinary` event updates the machine state s for an event e annotated as $Ordinary$. Its guards state that $InitDone$ is TRUE; the event e is a progress and ordinary event ($grd2$); the machine state s belongs to Grd of e ($grd3$); and feasibility and invariant properties of mch hold for the event e ($grd4$). Similar to the ordinary event, the last event `Do_Convergent` contains additional guards $grd2$ to tag the event e as convergent and $grd6$ to ensure that the variant properties of mch for the event e hold.

Note that our generic abstract model contains *initialisation*, *ordinary* and *convergent* events, whereas we may only have *initialisation* and *progress* events, in the same spirit of TLA$^+$, where the *progress* event can be refined by *ordinary* and *convergent* events later in further refinement. In this instantiation mechanism, the proof process relies on the induction principle offered by Event-B. Following the shallow modelling principle, the proof of machine consistency is delegated to Event-B itself. The defined properties are verified in the machine refining the generic machine *ShallowMchGen*.

MACHINE ShallowGenMch
SEES ShallowGenCtx
VARIABLES
s,
$InitDone$
INVARIANTS
inv1 : $s \in St$
inv2 : $InitDone \in BOOL$
inv3 : $InitDone = TRUE \Rightarrow s \in Inv(mch)$
EVENTS
INITIALISATION
THEN
act1 : $s, InitDone :| s' \in St \land InitDone' := FALSE$
END

Do_Init
WHERE
grd1 : $InitDone = FALSE$
grd2 : $Mch\_INV\_Init(mch) \land Mch\_FIS\_Init(mch)$
THEN
act1 : $s, InitDone :| s' \in AP(mch) \land InitDone := TRUE$
END

Do_Ordinary
ANY e
WHERE
grd1 : $InitDone = TRUE$
grd2 : $e \in Progress(mch) \land e \in Ordinary(mch)$
grd3 : $s \in Grd(mch)[\{e\}]$
grd4 : $Mch\_INV\_One\_Ev(mch, e) \land Mch\_FIS\_One\_Ev(mch, e)$
THEN
act1 : $s :\in BAP(mch)[\{e\}][\{s\}]$
END

```
  Do_Convergent
  ANY  e
  WHERE
    grd1 : InitDone = TRUE
    grd2 : e ∈ Progress(mch) ∧ e ∈ Convergent(mch)
    grd3 : s ∈ Grd(mch)[{e}]
    grd4 : Mch_INV_One_Ev(mch, e) ∧ Mch_FIS_One_Ev(mch, e)
    grd5 : Variant_WellCons(mch)
    grd6 : Mch_VARIANT_One_Ev(mch, e, s)∧
                Mch_NAT_One_Ev(mch, e)
  THEN
    act1 : s :∈ BAP(mch)[{e}][{s}]
  END

END
```

Listing 2.15: A generic abstract machine ($\mathcal{S}.2$)

Listing 2.16 presents a skeleton of an Event-B machine representing an instance formalising an Event-B model. Similarly to the deep modelling instantiation approach, a static part is described in another context. This skeleton machine refines the abstract machine (see Listing 2.15). It represents the dynamic part of the machine instantiate, where:

- **inv1** provides the gluing invariant of the abstract state s and the concrete one.
- each event is refined by the concrete one, and each concrete event provides witnesses ($WITH$ clause) for the event parameters.
- **grd1** is the guard's state, $InitDone$ is true or false depending on which events are refined, and the refinement of abstract state s in the concrete guard.
- **grd2** and **grd3** introduces instances of the guard, and BAP/AP.
- **act1** defines the concrete assignment as well as updates event parameters with a concrete one.

```
MACHINE ShallowMch REFINES ShallowGenMch SEES ...
VARIABLES
  ...,
  InitDone
INVARIANTS
  inv1 : s = ...  // Gluing invariant for abstract state
                  // variables s and concrete state variables
EVENTS // Static parts describe in the Event-B context
  INITIALISATION
  WITH s' : s' = ...
  THEN
    act1 : ..., InitDone :| ... ∧ InitDone' = FALSE
  END
  Do_Init REFINES Do_Init
  WHERE
    grd1 : InitDone = FALSE
    grd2 : ... = AP(mch)
  WITH s' : s' = ...
  THEN
    act1 : ..., InitDone :| ... ∈ AP(mch) ∧ InitDone' = TRUE
  END
```

```
... REFINES Do_Ordinary // Refines Do_Convergent if the event
                        // has the tag convergent
  WHERE // Guard strengthening encode the PO describe
        // as a guard in the abstract m.
    grd1: InitDone = TRUE ∧ ... ∈ Grd(mch)[{...}]
    grd2: ... = Grd(mch)[{...}]
    grd3: ... = BAP(mch)[{...}]
  WITH e: e = ..., s': s' = ...
  THEN
    act1: ... :| ... ∈ BAP(mch)[{...}][{...}]
  END
  ...
END
```

Listing 2.16: A shallow modelling machine skeleton

2.9 Case Study

To illustrate our approach, we have chosen the case study of a 24h hour clock originally defined by L. Lamport. The main functionalities (FUN) and requirements (REQ) of the clock case study are given as follows:

- **FUN1** A minute can progress
- **FUN2** An hour can progress
- **REQ1** The hours are represented in a 24-hour format.
- **REQ2** The clock must converge at midnight.
- **REQ3** The clock never stops.

```
MACHINE Clock
VARIABLES m, h
INVARIANTS
  inv1: m ∈ ℕ
  inv2: h ∈ ℕ
  inv3: m < 60
  inv4: h < 24
THEOREMS
  thm1: m < 59∨
        (m = 59 ∧ h < 23)∨
        (m = 59 ∧ h = 23)
VARIANTS
    24 * 60 − 1 − (m + h * 60)
EVENTS
  INITIALISATION
  THEN act1: m, h :|
        m' = 0 ∧ h' = 0
  END
  tick_min <convergent>
  WHERE grd1: m < 59
  THEN act1: m :| m' = m + 1
  END
  tick_hour <convergent>
  WHERE grd1: m = 59 ∧ h < 23
  THEN act1: m, h :|
        m' = 0 ∧ h' = h + 1
  END
  tick_midnight <ordinary>
  WHERE grd1: m = 59 ∧ h = 23
  THEN act1: m, h :|
        m' = 0 ∧ h' = 0
  END
END
```

Listing 2.17: A machine of clock

In Listing 2.17, we describe the clock model that is formalised in the native Event-B language. In this model, two variables are defined, minute m and hour h, in $inv1 - inv2$. Two safety properties are introduced in $inv3 - inv4$. The first safety property (REQ1) states that the minute m is always less than 60 and hour h is less than 24.The next safety property (REQ3) is defined as a theorem that is a disjunction of all guards to state that the clock never stops means always the guard of at least one event is true. The last safety property (REQ2) is related to convergence (variant) expressed by the number $24 * 60 - 1 - (m + h * 60)$. In this model, in addition to the initialisation ordinary event $INITIALISATION$, we introduce three events: *tick_min* - to model the minute progress by 1; *tick_hour* - to model the hour progress by 1 (two convergent events); and *tick_midnight* - to reset the clock at midnight (an ordinary event).

The required guards are added in the defined events to update the minute m and hour h.

2.10 EB4EB deep and Shallow modelling of the clock case study

Below we present the two instantiation mechanisms corresponding to the clock model of Listing 2.17.

2.10.1 Deep modelling instantiation for the clock model

We describe the development of the clock case study using the deep modelling instantiation technique of Section 2.8 using the meta-theory introduced in Section 2.4 and 2.5. All constituents of the Clock model are explicitly expressed in terms of the `EvtBStruc` and `EvtBPO` Meta-theory constructs. The Clock Event-B model is represented as an Event-B context using the skeleton shown in the Listing 2.13 of the Section 2.8, and POs are described either as theorems or as well-definedness POs.

The deep modelling resulting context of the Event-B clock model given in Listing 2.17 is presented in Listing 2.18. In this context, a set Ev lists all the clock events in $axm1$. The clock machine *clock* is defined by axiom $axm2$ as a member of $Machine(\mathbb{Z}\times\mathbb{Z}, Ev)$, where the first argument defines machine state as $\mathbb{Z}\times\mathbb{Z}$ and the second one machine events Ev.Note that $\mathbb{Z}\times\mathbb{Z}$ and Ev correspond respectively to the instantiation of the type parameters $STATE$ and $EVENT$ of the $EvtBPO$ theory. Furthermore, three axioms ($axm3 - axm5$) are used to instantiate $Event$ with the enumerated set Ev, $State$ with $\mathbb{Z}\times\mathbb{Z}$, and $Init$ with the event label *init*.

The next axiom ($axm6$) is instantiated with progress events. Axiom $axm7$ instantiates the after-predicate AP derived from the action of the initialisation event ($act1$) in the Clock machine. Similarly, axioms $axm8$ and $axm9$ are used to instantiate the guard Grd and the before-after predicate BAP with a set of guards and actions of all events derived from the Clock machine using comprehensive sets. The next axiom ($axm10$) is defined to instantiate invariant Inv using comprehensive sets derived from $inv1 - inv4$ of Listing 2.17, and $axm11$ is used to instantiate theorem Thm derived from $thm1$ of the clock model (see Listing 2.17). Axiom $axm12$ is used to instantiate the $Variant$ with the defined variant of the Clock model. The next axioms $axm13 - axm14$ instantiate the $Ordinary$ and $Convergent$ with a list of ordinary and convergent events, respectively. In this model, we have only two ordinary events *init* and *tick_midnight* and two convergent events *tick_min* and *tick_hour*.

Machine correctness. It is important to note the introduction, in Listing 2.18, of a *theorem* $thm1$ referring to the *check_Machine_Consistency* operator. The automatically generated well-definedness PO associated to the *check_Machine_Consistency* operator and the one for the theorem shall be proved. They entail machine correctness.

CONTEXT ClockDeep
SETS Ev
CONSTANTS clock , tick_min , tick_hour , tick_midnight , init
AXIOMS
axm1: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm2: $clock \in Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$
axm3: $Event(clock) = Ev$
axm4: $State(clock) = \mathbb{Z} \times \mathbb{Z}$
axm5: $Init(clock) = init$
axm6: $Progress(clock) = \{tick\_midnight, tick\_hour, tick\_min\}$
axm7: $AP(clock) = \{m \mapsto h \mid m = 0 \land h = 0\}$
axm8: $Grd(clock) = \{t \mapsto (m \mapsto h) \mid ($
$(t = tick\_min \land m < 59) \lor$
$(t = tick\_hour \land m = 59 \land h < 23) \lor$
$(t = tick\_midnight \land m = 59 \land h = 23))\}$
axm9: $BAP(clock) = \{t \mapsto ((m \mapsto h) \mapsto (mp \mapsto hp)) \mid ($
$(t = tick\_min \land mp = m + 1 \land hp = h) \lor$
$(t = tick\_hour \land mp = 0 \land hp = h + 1) \lor$
$(t = tick\_midnight \land mp = 0 \land hp = 0))\}$
axm10: $Inv(clock) = \{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}$
axm11: $Thm(clock) = \{m \mapsto h \mid$
$m < 59 \lor (m = 59 \land h < 23) \lor (m = 59 \land h = 23)\}$
axm12: $Variant(clock) = \{m \mapsto h \mapsto v \mid$
$v = 24 * 60 - 1 - (m + h * 60)\}$
axm13: $Ordinary(clock) = \{init, tick\_midnight\}$
axm14: $Convergent(clock) = \{tick\_min, tick\_hour\}$
THEOREMS
thm1: $check\_Machine\_Consistency(clock)$
END

Listing 2.18: A deep instance of the clock machine ($\mathcal{D}.2$)

2.10.2 Shallow modelling instantiation for the clock model

We describe the development of the clock case study using the shallow modelling instantiation technique of Section 2.8 using the meta-theory introduced in Section 2.4 and 2.5.

All the constituents of the Clock model are explicitly expressed using the `EvtBStruc` and `EvtBPO` Meta-Theory constructs. The corresponding Clock Event-B model is represented as a context (Listing 2.19) and a machine (Listing 2.20) corresponding to the skeleton machine of Listing 2.16. The POs guaranteeing the correctness of the instantiation are obtained by the theorems and by guard strengthening POs generated by the refinement of the abstract machine of Listing 2.15.

In the same vein as shallow embedding, we use Event-B to preserve semantics. We describe an abstract Event-B model formalising the required properties for Event-B models correctness: a `context` for the static part and properties and a generic `machine` for the dynamic parts i.e. transitions represented by events.

The concrete model refines the abstract generic model introduced above. The static elements of the clock model are described by the context of Listing 2.19 and dynamic elements are described in machine of Listing 2.20.

Static constituents (Event-B context). In the context of Listing 2.19, we define a constant pr in $axm1$ as a bijection relation between $(\mathbb{Z} \times \mathbb{Z})$ and St to maintain an exact correspondence between abstract and concrete states. We enumerate the set Ev with clock events in $axm2$. Axioms ($axm3$-$axm5$) are used to instantiate

Event with enumerated set Ev, *State* with St, and *Init* with the event *init*. Axiom $axm6$ is defined to instantiate progress events of the clock machine. Axiom $axm7$ defined invariant Inv using comprehensive sets derived from $inv1 - inv4$ of Listing 2.17. Axiom $axm8$ instantiates theorem Thm derived from $thm1$ of the clock model. Variant of the clock machine is introduced in $axm9$. Then two axioms ($axm10 - axm11$) are used to instantiate *Ordinary* and *Convergent* with a set of ordinary and convergent events.

Context correctness. We define four theorems ($thm1$-$thm4$) to check the proof obligation associated with the well-constructed event, well-constructed variant, well-constructed events tags (ordinary or convergent), and machine theorem. Once proved, these theorems guarantee that the context is well-defined and the required properties hold.

CONTEXT ClockShallowCtx
EXTENDS ShallowGenCtx
CONSTANTS tick_min , tick_hour , tick_midnight , init , pr
AXIOMS
axm1: $pr \in (\mathbb{Z} \times \mathbb{Z}) \rightarrowtail St$
axm2: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm3: $Event(mch) = Ev$
axm4: $State(mch) = St$
axm5: $Init(mch) = init$
axm6: $Progress(mch) = \{tick\_midnight, tick\_hour, tick\_min\}$
axm7: $Inv(mch) = pr[\{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}]$
axm8: $Thm(mch) = pr[\{m \mapsto h \mid m < 59 \lor (m = 59 \land h < 23) \lor (m = 59 \land h = 23)\}]$
axm9: $Variant(mch) = \{s, v, m, h \cdot s = pr(m \mapsto h) \land v = (24 * 60 - 1 - (m + h * 60)) \mid s \mapsto v\}$
axm10: $Ordinary(mch) = \{init, tick\_midnight\}$
axm11: $Convergent(mch) = \{tick\_min, tick\_hour\}$
THEOREMS
thm1: $Event\_WellCons(mch)$
thm2: $Variant\_WellCons(mch)$
thm3: $Tag\_Event\_WellCons(mch)$
thm4: $Mch\_THM(mch)$
END

Listing 2.19: Instances for static elements: clock machine ($\mathcal{S}.3$)

Dynamic constituents (event-B machine). The abstract machine is refined in Listing 2.20 to introduce the events of the *Clock* Event-B machine. In this model, we declare two new variables m and h and a gluing invariant $inv1$ to link (glue) concrete and abstract variables. No new event is added but each abstract event has been refined by concrete ones by providing concrete guards and actions. The newly introduced variables are set in the refined INITIALISATION event, and a witness is provided to map the abstract and concrete variables. In the `Do_Init` event, we introduce a new guard ($grd2$) to instantiate AP operator and a witness is provided for the state s'. The action of this event modifies the concrete clock variables m and h. The event `Do_Convergent` is refined by two concrete clock events `Tick_min` and `Tick_hour`, and the event `Do_Ordinary` is refined by the concrete event `Tick_midnight`. In the `Tick_min` event, $grd1$ is a data refinement, it introduces the two concrete variables m and h. The two other guards $grd2$ and $grd3$ respectively refer to the concrete guard and before after predicate of the *tick_min* event defined in the context of Listing 2.19. In this event, two witnesses are provided for the abstract parameter event e and the state s'. The action of

this event uses BAP operator to update concrete variables m and h. Similarly, the two other events `Tick_hour` and `Tick_midnight` are also obtained by refining the abstract events *Do_Convergent* and *Do_Ordinary* by providing witnesses and appropriate instantiations of guards and before-after predicates.

Machine correctness. Gluing invariant ($inv1$), witnesses and guard strengthening are introduced to check the POs associated with machine events (initial and tagged events). New generated POs are proved to guarantee that the machine is correct and the required properties hold. Here, the classical proof process associated to Event-B with its inductive reasoning is used.

MACHINE ClockShallow **REFINES** ShallowGenMch
SEES ClockShallowCtx
VARIABLES
m ,
h ,
$InitDone$
INVARIANTS
$inv1$: $s = pr(m \mapsto h)$ // *Gluing Invariant*
EVENTS
INITIALISATION
WITH
s' : $s' = pr(m' \mapsto h')$
THEN
$act1$: $m, h, InitDone :| m' \in \mathbb{Z} \land h' :\in \mathbb{Z} \land InitDone' = FALSE$
END

Do_Init REFINES Do_Init
WHERE
$grd1$: $InitDone = FALSE$
$grd2$: $pr[\{0 \mapsto 0\}] = AP(mch)$
WITH
s' : $s' = pr(m' \mapsto h')$
THEN
$act1$: $m, h, InitDone :| pr(m' \mapsto h') \in AP(mch) \land InitDone = TRUE$
END

Tick_min REFINES Do_Convergent
WHERE
$grd1$: $InitDone = TRUE \land pr(m \mapsto h) \in Grd(mch)[\{tick\_min\}]$
$grd2$: $pr[\{ms, hs \cdot ms < 59 \land hs \in \mathbb{Z} \mid ms \mapsto hs\}] = Grd(mch)[\{tick\_min\}]$
$grd3$: $\{ss, ssp, ms, hs, msp, hsp \cdot ss = pr(ms \mapsto hs) \land ssp = pr(msp \mapsto hsp) \land msp = ms + 1 \land hs = hsp \mid ss \mapsto ssp\} = BAP(mch)[\{tick\_min\}]$
WITH
e : $e = tick\_min$, s' : $s' = pr(m' \mapsto h')$
THEN
$act1$: $m, h :| pr(m' \mapsto h') \in BAP(mch)[\{tick\_min\}][\{pr(m \mapsto h)\}]$
END

Tick_hour REFINES Do_Convergent
WHERE
$grd1$: $InitDone = TRUE \land pr(m \mapsto h) \in Grd(mch)[\{tick\_hour\}]$
$grd2$: $pr[\{ms, hs \cdot hs < 23 \land ms = 59 \mid ms \mapsto hs\}] = Grd(mch)[\{tick\_hour\}]$
$grd3$: $\{ss, ssp, ms, hs, msp, hsp \cdot ss = pr(ms \mapsto hs) \land ssp = pr(msp \mapsto hsp) \land msp = ms \land hsp = hs + 1 \mid ss \mapsto ssp\} = BAP(mch)[\{tick\_hour\}]$
WITH
e : $e = tick\_hour$, s' : $s' = pr(m' \mapsto h')$

```
  THEN
    act1 :  m, h :|
          pr(m' ↦ h') ∈ BAP(mch)[{tick_hour}][{pr(m ↦ h)}]
  END

  Tick_midnight REFINES Do_Ordinary
  WHERE
    grd1 :  InitDone = TRUE∧
              pr(m ↦ h) ∈ Grd(mch)[{tick_midnight}]
    grd2 :  pr[{ms, hs · ms = 59 ∧ hs = 23 | ms ↦ hs}]
              = Grd(mch)[{tick_midnight}]
    grd3 :  {ss, ssp, ms, hs, msp, hsp·
          ss = pr(ms ↦ hs) ∧ ssp = pr(msp ↦ hsp)∧
          msp = 0 ∧ hsp = 0 | ss ↦ ssp}
              = BAP(mch)[{tick_midnight}]
  WITH
    e :  e = tick_midnight ,  s' :  s' = pr(m' ↦ h')
  THEN
    act1 :  m, h :|
          pr(m' ↦ h') ∈ BAP(mch)[{tick_midnight}][{pr(m ↦ h)}]
  END
END
```

Listing 2.20: A shallow instance of the clock machine ($\mathcal{S}$.4)

2.11 Extending the EB4EB Reasoning Mechanism (see Fig. 2.1.(D))

Extensibility is one of the benefits of the meta-theory *EvtBStruc* and *EvtBPO* of Sections 2.4 and 2.5: every Event-B feature is explicitly formalised and can be manipulated using operators, making it possible to define specific development operations or new reasoning mechanisms, as new operators, in a *non-intrusive* way. By non-intrusive, we mean that these development operations do not affect the classical Event-B development as machines are manipulated as instances of the meta-theory *EvtBStruc* and *EvtbPO*. The principle of designing such Event-B machine analyses is described below.

2.11.1 Analysis principle: New POs

In our framework, model analysis is defined by introducing a new PO which must fulfil two requirements 1) first this PO shall be reusable and 2) and second, it shall be generated automatically. The first requirement is met by formalising the PO at the meta-theory level as a predicate operator and the second one relies on the exploitation of well-definedness (WD) and Theorems (THM) POs automatically generated.

Event-B machine analysis pattern

The definition of a new PO is depicted by the theory pattern of Listing 2.21. `Theo4PO` theory (see Fig. 2.1.(D)) imports the meta-theory *EvtBPO* and introduces a third, optional type parameter T_{Args} possibly needed by the analysis. The PO associated to the analysis is defined by a predicate operator `[PO]_Definition` formalising the PO as a predicate. Then, checking the defined PO, is realised by

the `check_Machine[PO]` predicate operator which is well-defined when machine m is well structured and consistent.

```
THEORY Theo4PO IMPORT EvtBPO
TYPE PARAMETERS STATE, EVENT, T_Args
OPERATORS
  [PO]_Definition <predicate> (m : Machine(STATE, EVENT), args : T_Args)
    well-definedness condition ...
    direct definition ...
  check_Machine_[PO] <predicate> (m : Machine(STATE, EVENT), args : T_Args)
    well-definedness condition
        Machine_WellCons(m) ∧ check_Machine_Consistency(m)
    direct definition [PO]_Definition(m, args)
END
```

Listing 2.21: Analyses Theory

Once a PO is defined, its correctness is established following the same approach as the one we set up in section 2.7 to prove the correctness of native proof obligation. Another theory `Theo4POCorrectness` (see Fig 2.1.(C, D)) is introduced. It relies on trace semantics proposed in Section 2.6. Listing 2.22 shows the skeleton of the `Theo4POCorrectness` theory with a correctness theorem to be proved. It states that `[PO]_Definition` implies the expression $PO\_Spec\_On\_Traces(\ldots)$, on the traces, of the specification of the concerned PO.

```
THEORY Theo4POCorrectness IMPORT EvtBTraces , Theo4PO
TYPE PARAMETERS STATE, EVENT, T_Args
THEOREMS
    thmCorrectnessPO :
        ∀m, tr · m ∈ Machine(STATE, EVENT)∧
        Machine_WellCons(m) ∧ IsATrace(tr, m)∧
        ... ∧ [PO]_Definition(m, args)
            ⇒ PO_Spec_On_Traces(...)
```

Listing 2.22: Analyses Correctness

Checking PO context pattern

Listing 2.23 shows the Event-B context pattern defined to check the newly defined PO. A consistent instance machine context *Deep*, associated to an Event-B machine resulting from the instantiation of the meta-theory *EvtBStruc* and *EvtBPO*, is extended by the context *MachinePO* instantiating the extended theory *Theo4PO*. Theorem *thmPO* checks that the PO holds for the machine *mch*. The WD and THM associated POs are automatically generated.

```
CONTEXT MachinePO
EXTENDS Deep
THEOREMS
  thmPO :
      check_Machine_[PO](mch, args)
END
```

Listing 2.23: Analyses Machine

The main key points of using this framework is that 1) well-definedness conditions ensure elements are used correctly, 2) meta-properties on these analyses are done once and for all, and 3) these analyses can be performed without altering the machine's behaviour, in a non-intrusive way.

This approach is demonstrated below on the definition of the deadlock freeness proof obligation.

2.11.2 Introduction of deadlock-freeness as a new proof obligation

Requirements

Deadlock-freeness states that a machine m cannot be in a state where no progress is possible, i.e. at least one event in a machine m is always enabled. Informally, it can be formulated as: when the invariant holds then the disjunction of all the events guards holds.

PO Definition

The PO shall state that, for a machine m, there exists at least one event e such that the current state satisfies its guard i.e. $s \in Grd(m)(e)$. When expressed using the operators of the meta-theory, we write $Inv(m) \subseteq Grd(m)[Progress(m)]$. This operator does not require any additional argument for $args$.

```
THEORY Theo4Deadlock IMPORT EvtBPO
TYPE PARAMETERS STATE, EVENT
OPERATORS
  DeadlockFreeness_Definition <predicate> (m : Machine(STATE, EVENT))
    direct definition Inv(m) ⊆ Grd(m)[Progress(m)]
END
```

Listing 2.24: DeadlockFree Theory

According to the defined pattern, we introduce, in Listing 2.24, a new theory `Theo4Deadlock`, with two new operators and required well-definedness condition.

Deadlock freeness PO for Clock model

The extended context with the $thmDLK$ theorem generating WD and THM POs of the *clock* machine is shown in Listing 2.25.

```
CONTEXT ClockDeadlockFree
EXTENDS ClockDeep
THEOREMS
  thmDLK: check_Machine_DeadLock(clock)
END
```

Listing 2.25: Clock DeadlockFreeness

Correctness

We specify the deadlock freeness properties using the Event-B trace semantics described in Section2.7 in order to check the correctness of the defined PO, and thus ensure that a developed model is deadlock-free. The deadlock freeness PO is defined as `DeadlockFreeness_Definition` in Listing 2.24.

```
THEOREMS
    ThmCorrectnessDeadlockFree :
        ∀m, tr · m ∈ Machine(STATE, EVENT)∧
        Machine_WellCons(m) ∧ IsATrace(tr, m)∧
        Mch_INV(m) ∧ DeadlockFreeness_Definition(m)
            ⇒ AllAre(tr, Grd(m)[Progress(m)])
```

Listing 2.26: Theorem of Deadlock freeness' correctness

Listing 2.26 presents the theorem `Theo4DeadlockCorrectness` to ensure the correctness of deadlock freeness PO. It states that for all well-constructed machines ($Machine\_WellCons(m)$) and for all traces of any machine such that traces of machines ($IsATrace(tr, m)$) are valid ones, machine invariants fold ($Mch\_INV(m)$), and the proof obligation of deadlock freeness ($DeadlockFreeness\_Definition(m)$) holds, then any state satisfies at least one guard of a progress event at any state of the trace ($AllAre$).

2.12 Proof Process

The Rodin platform is well-equipped with different types of provers and SMT solvers to support proof automation for the core Event-B. However, the current proof process for meta models developed in the EB4EB framework lacks automation. Thus, user interaction is needed to discharge the generated proof obligations. One of our primary goals is to experiment with shallow modelling mechanisms to use the native induction proof process in Event-B to reduce overall proof efforts and improve proof automation. On the other hand, we propose some proof rules in the developed theories to define some rewriting rules to simplify the direct definitions and well definedness of the EB4EB framework's defined operators. One of these rewrite rules is based on the deep modelling template given in the Listing 2.13, where the destructor of the machine instantiated in the proof obligation is replaced by the instance's set comprehension. Then, the definitions of these proof rules are integrated with existing or new proof tactics of Rodin. These rules are automatically applied by the Rodin prover when tactics are invoked.

Model	PO	Max Depth	Nodes	Interactive Nodes	Number of Tactic application
Clock deep	thm1/WD	47	137	1	2
	thm1/THM	108	352	0	1
DeadlockFree	thmDLK/THM	169	221	1	2

Table 2.5: Proof statistics

Table 2.5 shows the number of automatic nodes for each theorem of the deep modelling. Without any tactic, these nodes are discharged manually for each operator by instantiating correctly. The introduction of new proof rules in form of tactics enables to discharge most of the nodes automatically. For example, most of the proof obligations of the clock model are discharged automatically and only one node requires manual interaction. Similarly, the deadlock freeness has 169 nodes without tactic and only one interactive node with tactic.

2.13 Conclusion

We presented the EB4EB framework, which allows for the explicit manipulation of Event-B features using meta-modelling concepts. The developed framework is a collection of theories that includes data types, operators, WD conditions,

theorems, and proof rules. These theories are specially designed for encoding the core modelling constructs and proof obligation rules of Event-B. In addition, trace semantics is provided to ensure the correctness of the introduced Event-B language constructs. We developed two instantiation mechanisms, deep and shallow, to use the defined theories and associated operators, definitions, WD and proof rules. These mechanisms allow manipulation of static and dynamic concepts of Event-B features as well as defining new proof obligations to support advanced level reasoning once and for all. Note that these theories must be instantiated in new development, and the generated POs must be discharged to ensure correct instantiation. The expressiveness, effectiveness, portability, and scalability of our developed EB4EB framework and its trace semantics were evaluated using Lamport's clock case study. Finally, we showed, on the case of the deadlock freeness, that correct extensions can be introduced.

In the future, we intend to use EB4EB framework to extend the reasoning mechanism by supporting externally defined POs to analyse domain-specific properties, such as continuous behaviour, human-machine interactions etc. In addition, we plan to certify Rodin plugins like composition/decomposition, code generation and model transformations and so on, using EB4EB framework. Another important goal is to use Dedukti [Boespflug et al., 2012] to import and export the Event-B theory and models into proof assistants such as Coq, PVS and Isabelle/HOL.

Acknowledgements This study was undertaken as part of the EBRP (*Enhancing Event-B and RODIN: EventB-RODIN-Plus*) project funded by ANR, the French National Research Agency.

Proof automation for Event-B theories

P. Rivière, N. K. Singh, Y. Aït-Ameur, G. Dupont [2]

INPT-ENSEEIHT/IRIT, University of Toulouse, France
{peter.riviere, nsingh, yamine, guillaume.dupont}@enseeiht.fr

In order to enrich its expressiveness, Butler et al. [M. J. Butler & Maamria, 2013] proposed a mathematical extension to Event-B. This extension enables the description of algebraic definitions for data-types and operators in a reusable component, *theories*. Theories may also present new theorems and proof rules to handle the new user-defined constructs, which may be used seamlessly when proving models.

Currently, the elements introduced by theories are not always properly handled by automatic provers, especially SMT solvers and Atelier-B provers. If users want to use these tools, they need to manually unfold and rewrite each operator to classical Event-B expressions, which can be cumbersome.

In this presentation, we propose to encode new proof principles as well as to introduce new strategies to automatically unfold theory operator, hence improving proof automation. This solution is adopted in the development of the reflexive EB4EB framework [Riviere et al., 2022a, 2022b].

The main objective of the EB4EB reflexive framework [Riviere et al., 2022a, 2022b] is to provide explicit manipulation of Event-B components as first-class objects, making it possible to reason on these objects and define new Event-B analyses. For this purpose, the concept of Event-B machine is formalised as a data-type in a theory (a meta-theory), together with a set of operators that guarantee the correctness, relative to Event-B semantics, of instances of this data-type. The meta-theory formalises the semantics of Event-B, as described in the Event-B Book [J.-R. Abrial, 2010], i.e. a set of states and guarded events defined as a relation between states. In addition, the EB4EB framework is extended to support new analyses, possibly non-intrusive, mechanisms associated to different properties not expressed in core Event-B [Riviere et al., 2023c]. In this work, we present three properties, *deadlock freeness*, *invariant weakness analysis* and *reachability*, to demonstrate extension of reasoning mechanism using the reflexive Event-B. Furthermore, this reflexive framework EB4EB has been extended to formalise and operationalise the automatic genera-

```
THEORY EvtBTheo
TYPE PARAMETERS STATE, EVENT
DATATYPES
  Machine(STATE, EVENT)
  CONSTRUCTORS
    Cons_machine(
      Event : ℙ(EVENT),
      State : ℙ(STATE),
      Init : EVENT,
      Progress : ℙ(EVENT)
      AP : ℙ(STATE),
      Grd : ℙ(EVENT × STATE),
      BAP : ℙ(EVENT × (STATE × STATE)),
      Inv : ℙ(STATE)
      Thm : ℙ(STATE),
      Variant : ℙ(STATE × ℤ),
      Ordinary : ℙ(EVENT),
      Convergent : ℙ(EVENT))
```

Listing 2.27: Machine Data-Type

[2]The authors thank the ANR-19-CE25-0010 *EBRP:EventB-Rodin-Plus* project.

tion of proof obligations associated to *temporal properties* expressed in LTL [Riviere et al., 2023a].

These theories are extended with automatic rewriting rules that substitute operators by their given definition in order to automate proof processes. These rules are written to extract relevant information from machine objects, add them to the hypotheses, and produce multiple simpler goals. For example, Listing `rew2` shows rewriting rule for simplifying proof process related to deadlock freeness. Similarly, several rules are encoded in the theories. These rules are defined to be applied automatically and chained together, greatly improving proof automation. Indeed, these rewrite rules are included in Rodin's user-defined proof tactics, once and for all, increasing automation when proving the theorems formalising the newly defined POs. Note that these rules follows a pattern that can be applied systematically.

PROOF RULES
extension_def:
Metavariables
$m : Machine(STATE, EVENT)$
Rewrite Rules
...
$rew2:\ DeadlockFreeness\_Definition(m)$
rhs1: $\top \Rightarrow \forall g, i, p \cdot Progress(m) = p \wedge Grd(m) = g \wedge Inv(m) = i \Rightarrow i \subseteq g[p]$

Listing 2.28: Proof Rule to unfold operator definition

Proof automation using rewriting rules is demonstrated on Clock examples in particular analysis of different POs. Table 2.6 presents the proof statistics for each analysis. The important number of nodes (representing atomic steps) in the proof trees is due to the extensive use of theory operators which the prover cannot handle directly, and thus their definitions must be unfolded. The rewrite rules introduced in a proof tactic performs automatically these unfolds and reductions, making almost every steps fully automatic despite the introduction of the meta level (an entry of 0 in the interactive nodes column of Table 2.6). The rightmost column provides the number of tactic applications (iterations) during the proof. Indeed, a single tactic application may not be sufficient to fully discharge the proof goals.

Model	PO	Max Depth	Nodes	Interactive Nodes	Number of Tactic application
DeadlockFree clock	thmDeadlock (THM)	169	221	1	2
Reachability clock	thmReach (WD)	112	577	0	1
	thmReach (THM)	191	731	4	5
Inspect Inv clock	thmInspectInvEVTM5 (THM)	111	167	0	1
	thmInspectInvEVTH5 (THM)	112	169	0	1
	thmInspectInvEVTMH1 (THM)	113	171	0	1
Strong Inv clock	thmInspectInvEVTM5 (THM)	105	158	0	1
	thmInspectInvEVTH5 (THM)	118	171	0	1
	thmInspectInvEVTHM1 (THM)	128	181	0	1

Table 2.6: Proof statistic for the Clock model and its analyses

Assessment

The EB4EB framework lays the foundations to build a powerful mechanism for developing and analysing Event-B models, facilitating modularisation and a higher level of abstraction. It also permits extensions to the Event-B method in terms of the introduction of new kinds of proof obligations and domain-specific properties. It is inbuilt with trace semantics for ensuring correctness. The Lamport's clock case study is used to demonstrate the EB4EB framework.

One of the main advantages of using this framework is the ability to customise the Event-B method to fit specific modelling and verification needs, from novel semantics to domain-specific properties. As machines are first-class elements, the framework also promotes a high level of re-usability for models. Last, having access to explicit trace-based semantics paves the way for encoding other types of semantics, featuring different sorts of transitions (e.g., continuous) while preserving the correctness of the method.

This framework has some limitations that will be addressed over time. These limitations include a lack of support for proof automation, an incomplete algebraic formalisation of Event-B components within the framework, and consistency check for the defined axioms. A set of new proof rules is developed, however they are built on fixed patterns that can be enhanced in the future to improve proof automation. All the defined operators are introduced in accordance with our requirements. In the future, we may need to add more algebraic definitions for other remaining Event-B components to make this framework more advanced and complete. In order to check the consistency in the defined axioms, we may use other advanced theorem prover like Coq[Bertot & Castéran, 2010], Isabelle [Nipkow et al., 2002], or PVS [Owre et al., 1992].

Chapter 3

Advanced Reasoning on Event-B Models

Overview

This chapter describes the core methodology for enhancing the reasoning capabilities of Event-B, in order to check other advanced properties, not available in native Event-B. This core methodology is based on extending the EB4EB framework; it is used to define three analyses: *deadlock-freeness*, *invariant weakness analysis*, and *reachability*. The EB4EB framework is extended via a new theory, which includes a set of additional operators in the form of algebraic definitions, as well as the requisite well-definedness conditions. This methodology is also developed in the Event-B modelling language with the *Theory plugin* and Rodin IDE, and a set of extra proof tactics is defined to aid with proof automation. Finally, Lamport's clock case study is employed to demonstrate the new reasoning capabilities.

Associated paper of this chapter:

- Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2023c]. Standalone Event-B models analysis relying on the EB4EB meta-theory. *International Conference on Rigorous State Based Methods, ABZ 2023*

Standalone Event-B models analysis relying on the EB4EB meta-theory

P. Rivière, N. K. Singh, Y. Aït-Ameur, G. Dupont [1]

INPT-ENSEEIHT/IRIT, University of Toulouse, France
{peter.riviere, nsingh, yamine, guillaume.dupont}@enseeiht.fr

Event-B is a state-based correct-by-construction system design formal method relying on proof and refinement where system models are expressed using set theory and First Order Logic (FOL). Through the generation and discharging of proof obligations (POs), Event-B natively supports the establishment of properties such as safety invariant, convergence and refinement. Other properties, relevant to system verification, may be studied as well, but need to be explicitly formalised by the designer, or expressed in another formal method. This process compromises reusability and is error-prone, especially on larger systems. Recently, the reflexive EB4EB framework has been proposed for formalising Event-B concepts as first-class objects. It allows manipulating these concepts using FOL and set theory in Event-B. In this paper, we propose a rigorous methodology for extending the EB4EB framework, to support new system analysis mechanisms associated to properties that are not natively present in core Event-B. Thanks to the reflexive nature of this framework, new generic and reusable system properties and their associated POs are expressed once and for all, and for any refinement level. For specific systems, designers instantiate these properties and the associated POs are automatically generated and submitted to Event-B's provers. This methodology is used to define three analyses: deadlock-freeness, invariant weakness analysis and reachability, all of which are demonstrated on a case study.

Reflection, Refinement and Proof, Meta-theory, Reachability, Deadlock-Freeness, Invariant weakness, EB4EB framework, Event-B.

3.1 Introduction

Context. The refinement and proof state-based Event-B formal method [J.-R. Abrial, 2010] supports complex system development using a correct-by-construction approach. It is based on set theory and First Order Logic (FOL) for describing state transition systems. It relies on an inductive proof process to discharge a set of proof obligations (POs) expressing various properties. Basically, core Event-B offers built-in modelling constructs to express invariants, event convergence, simulation, guard strengthening and event feasibility. POs associated to these constructs are automatically generated and are discharged using automatic and interactive provers.

In order to enrich the method's expressiveness, Event-B has been extended with the ability to define new algebraic data-types resulting in a richer type sys-

[1]The authors thank the ANR-19-CE25-0010 *EBRP:EventB-Rodin-Plus* project.

tem [J.-R. Abrial et al., 2009; M. J. Butler & Maamria, 2013], through the introduction of *Theories*. This extension allows the formalisation of complex systems at a higher level of abstraction.

Motivation. Event-B theories make it possible to formalise new data types, but *they do not allow the definition of new POs* that correspond to properties other than the usual ones (i.e., invariants preservation, event convergence, etc.).

Indeed, when properties such as deadlock-freeness, event scheduling, liveness, and so on need to be proved, they are explicitly formalised by the designer, or expressed in another formal method. This process compromises reusability and is error-prone, especially on large systems. The designer shall formalise each desired property for each system under design using the native Event-B POs. This process may be cumbersome, must be repeated for each model to be analysed (not reusable) and results in formal developments scattered across multiple heterogeneous frameworks and semantics.

To incorporate such properties in Event-B once and for all and allow the automatic generation of property-specific POs, it is necessary to embed, in the Event-B engine, the POs associated to these new properties. Such embedding requires the manipulation, in Event-B, of Event-B concepts as first-order objects (i.e., through a reflexive meta-model). We have recently proposed a reflexive EB4EB framework[Riviere et al., 2022a, 2022b] that formalises Event-B concepts as first-class objects in Event-B. It allows manipulating these concepts in Event-B using first-order logic and set theory. It is built on an algebraic meta-theory formalised as an Event-B theory, where each Event-B feature can be handled at the meta-model level, as first-class citizen. This framework also formalises Event-B's trace-based semantics and offers constructs for machines, states, and events together with a set of operators for manipulating them. Consequently, the EB4EB framework makes it possible to formally express, at any abstraction level (i.e. in the refinement chain), new reusable and automatically generated POs and high-level constructs, easing the development of complex systems with specific properties or semantics. Furthermore, it opens the door to formally embed Event-B's semantics in other formal methods and exploit their respective strengths.

Objective of this paper. This paper extends and enriches our previously developed EB4EB framework [Riviere et al., 2022a, 2022b] to support new analysis mechanisms (possibly non-intrusive), formalised as logic properties not available in native Event-B nor in its base PO generator. It extends the EB4EB Event-B meta-theory with new operators formalising such new properties. The POs associated to each operator are automatically generated. Adding the desired property, corresponding to a specific analysis, to an Event-B model is performed by invoking an operator. Designers do not need to formalise this property explicitly in the model.

Structure of the paper. The paper is organised as follows. Section 3.2 describes the Event-B method and the Theory mathematical extension. Section 3.3 introduces the EB4EB framework and its Event-B meta-theory, as well as the case study used throughout this paper. Three externally defined Event-B analyses and POs are introduced in Section 3.4 and applied to the case study. The positioning of

Context	Machine	Theory
CONTEXT Ctx **SETS** s **CONSTANTS** c **AXIOMS** A **THEOREMS** T_{ctx} **END**	**MACHINE** M **SEES** Ctx **VARIABLES** x **INVARIANTS** $I(x)$ **THEOREMS** $T_{mch}(x)$ **VARIANT** $V(x)$ **EVENTS** **EVENT** evt **ANY** α **WHERE** $G_i(x,\alpha)$ **THEN** $x :\mid BAP(\alpha, x, x')$ **END** ... **END**	**THEORY** Th **IMPORT** Th1, ... **TYPE PARAMETERS** E, F, ... **DATATYPES** **Type1**(E, ...) **constructors** **cstr1**(p_1: T_1, ...) **OPERATORS** **Op1** <nature> (p_1: T_1, ...) **well—definedness** $WD(p_1, ...)$ **direct definition** D_1 **AXIOMATIC DEFINITIONS** **TYPES** A_1, ... **OPERATORS** **AOp2** <nature> (p_1: T_1, ...): T_r **well—definedness** $WD(p_1, ...)$ **AXIOMS** A_1, ... **THEOREMS** T_1, ... **PROOF RULES** R_1, ... **END**
(a)	(b)	(c)

Table 3.1: Global structure of Event-B Contexts, Machines and Theories

(1)	Ctx Theorems (ThmCtx)	$A(s,c) \Rightarrow T_{ctx}$ `(For contexts)`
(2)	Mch Theorems (ThmMch)	$A(s,c) \wedge I(x) \Rightarrow T_{mch}(x)$ `(For machines)`
(3)	Initialisation (Init)	$A(s,c) \wedge BAP(x') \Rightarrow I(x')$
(4)	Invariant preservation (Inv)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge BAP(x,\alpha,x') \Rightarrow I(x')$
(5)	Event feasibility (Fis)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \Rightarrow \exists x' \cdot BAP(x,\alpha,x')$
(6)	Variant progress (Var)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge BAP(x,\alpha,x') \Rightarrow V(x') < V(x)$

Table 3.2: Relevant POs for Event-B contexts and machines

this work with respect to the state of the art and its advantages are discussed in Section 3.5. Finally, Section 3.6 concludes the paper and discusses future work.

3.2 Event-B

Event-B [J.-R. Abrial, 2010] is based on set theory and FOL. It relies on an expressive state-based modelling language where a set of events models state changes.

3.2.1 Contexts and machines (Tables 3.1.a and 3.1.b)

A `Context` (Table 3.1.a) describes the static part of a model. It introduces *carrier sets* s and *constants* c, and their properties using *axioms* A and *theorems* T_{ctx}. A `Machine` (Table 3.1.b) describes the model behaviour as a transition system. A set of *events* evt, possibly guarded by G and/or parameterized by α, is used to modify a set of state variables x using Before-After Predicates (BAP) to record state changes. A machine may define *invariants* $I(x)$, *theorems* $T_{mch}(x)$ and *variants* $V(x)$ to capture particular properties (e.g., safety and convergence). Model consistency is ensured via a set of generated POs, given in Table 3.2.

Refinements. Refinement decomposes a *machine* into a less abstract one with more design decisions (refined states and events) moving from an abstract level to

a less abstract one (simulation relationship). Gluing invariants relating abstract and concrete variables ensure property preservation.

Core Well-definedness (WD). In addition to machine-related POs, each operator is associated to a *WD*, that must be established for expressions to be meaningful. Once proved, these WD conditions are used as hypotheses to prove further POs.

3.2.2 Event-B extensions with Theories

To handle more complex and abstract concepts beyond set theory and FOL, an Event-B extension for externally defined mathematical objects has been proposed [J.-R. Abrial et al., 2009; M. J. Butler & Maamria, 2013]. It introduces user data types with new types, operators, theorems and associated rewrite and inference rules, all bundled in so-called *theories*. Close to proof assistants like Coq [Bertot & Castéran, 2010], Isabelle/HOL [Nipkow et al., 2002] or PVS [Owre et al., 1992], this capability is convenient to model, as data types, *concepts unavailable in core Event-B*.

Theory description (See Table 3.1.c). Theories define new data types, operators, and theorems. Data types (`DATATYPES` clause) define *constructors* to build inhabitants of the defined type. It may define various *operators* further used in Event-B expressions as FOL *predicates* or *expressions* producing actual values (<nature> tag). Operators may be used in theories, contexts and machines.

Operators may be defined explicitly in the `DIRECT DEFINITION` clause (constructive definition), or axiomatically in the `AXIOMATIC DEFINITIONS` clause (a set of axioms). Last, a theory defines a set of axioms, completing the definitions, as well as theorems and proof rules. Theorems and proof rules are proved from the definitions and axioms used by the proof system. Many theories have been defined for sequences, lists, groups, reals, differential equations, etc.

Well-definedness (WD) in Theories. An important feature provided by Event-B theories is the possibility to define *Well-Definedness* (WD) conditions (close to Type-Correctness Condition (TCC) conditions in PVS [Owre et al., 1992]). TCC must be discharged before the corresponding theory types correctly. Similarly, in Event-B theories, each defined operator (thus partially defined) is associated with a user-defined condition ensuring its well-formedness. Note that, when an operator is applied, it automatically invokes its WD condition and generates a PO requiring to establish that this condition holds, i.e., the operator is used correctly and that its parameters belong to its definition domain.

Event-B proof system and its IDE Rodin. Rodin is an open source IDE for modelling in Event-B. It offers resources for model editing, automatic PO generation, project management, refinement and proof, model checking, model animation and code generation. The Event-B theories extension is available as a plug-in. Theories are tightly integrated in the proof process. Depending on their definition (direct or axiomatic), operator definitions are expanded either using their direct definition (if available) or by enriching the set of axioms (hypotheses in proof sequents) using their axiomatic definition. Theorems may be imported as hypotheses and used in proofs like other theorems. Many provers for first-order

logic as well as SMT solvers are plugged to Rodin for helping the proof process.

3.3 The EB4EB framework

The main objective of the EB4EB reflexive framework [Riviere et al., 2022a, 2022b] is to provide explicit manipulation of Event-B components as first-class objects, making it possible to reason on these objects and define new Event-B analyses. For this purpose, the concept of Event-B machine is formalised as a data-type in a theory (a meta-theory), together with a set of operators that guarantee the correctness, relative to Event-B semantics, of instances of this data-type. The meta-theory formalises the semantics of Event-B, as described in the Event-B Book [J.-R. Abrial, 2010], i.e. a set of states and guarded events defined as a relation between states. In addition, the meta-theory is equipped with relevant proved (once and for all) theorems useful for discharging the generated POs. These additional theorems are available to help users reduce proof efforts and aid in system development and analysis.

Event-B machines (models) are defined using the meta-theory mentioned above, by instantiating the machine data-type and providing appropriate values for each of its fields: states, events, guards, before-after predicates, invariants, variant and so on. At instantiation, operators of the meta-theory are used in theorems; the related POs ensure the defined machine's consistency, including invariant preservation, event feasibility, variant progress, etc.

As previously stated, the goal of this paper is to demonstrate that the meta-theory can be extended with new operators for manipulating machine elements of the meta-theory, in order to define so-called *analyses*, expressed with new POs. Based on the work presented in [Aït Ameur et al., 2022], such analyses allow the system designer to check new properties, obtain feedback about their behaviour, enrich model design phases and check new properties that are not available in core Event-B.

This section summarises the main features of the Event-B meta-theory (Listings 3.1, 3.2 and 3.3), and presents the case study used to illustrate our approach throughout this paper.

3.3.1 The Event-B Meta-theory

Machine structure. Listing 3.1 shows the `Machine` data-type, defined using type parameters for abstracting event labels (`EVENTS`) and states (`STATES`). It is built using the `Cons_machine` single constructor with a parameter for each machine component, and defines a state-transition system with state *State* (constrained by invariants *Inv* and theorems *Thm*) and a set of, possibly parameterised, events (*Event*), with an initialisation event *Init* and progress events *Progress*, split into ordinary *Ordinary* and convergent *Convergent* events. State changes are recorded using an *after-predicate* (*AP*) for initialisation and a set of *before-after predicates* (*BAP*) associated to progress events, possibly guarded (*Grd*). Finally, integer variants for event convergence are introduced as well (*Variant*).

```
THEORY EvtBTheo
TYPE PARAMETERS STATE, EVENT
DATATYPES
  Machine(STATE, EVENT)
 CONSTRUCTORS
   Cons_machine(
    Event : ℙ(EVENT),
    State : ℙ(STATE),
    Init : EVENT,
    Progress : ℙ(EVENT)
    AP : ℙ(STATE),
    Grd : ℙ(EVENT × STATE),
    BAP : ℙ(EVENT × (STATE × STATE)),
    Inv : ℙ(STATE)
    Thm : ℙ(STATE),
    Variant : ℙ(STATE × ℤ),
    Ordinary : ℙ(EVENT),
    Convergent : ℙ(EVENT))
```

Listing 3.1: Machine Data-type

Well-Constructed machines. To ensure machines are structurally well-defined, the meta-theory introduces several predicate operators (Listing 3.2): `BAP_WellCons` to check that each progress event is associated to a BAP, `Grd_WellCons` to check that progress events are possibly guarded, and `Event_WellCons` to check that machine events are composed of an initialisation (`Init`) and progress (`Progress`) events. The `Machine_WellCons` predicate operator, defined as a conjunction of the previous operators (and others), ensures that a machine is well-structured (static semantics).

```
  BAP_WellCons <predicate> (m : Machine(STATE, EVENT))
    direct definition dom(BAP(m)) = Progress(m)
  Grd_WellCons <predicate> (m : Machine(STATE, EVENT))
    direct definition dom(Grd(m)) = Progress(m)
  Event_WellCons <predicate> (m : Machine(STATE, EVENT))
    direct definition partition(Event(m), {Init(m)}, Progress(m))
    ...
  Machine_WellCons <predicate> (m : Machine(STATE, EVENT))
    direct definition
      BAP_WellCons(m) ∧ Grd_WellCons(m) ∧ Event_WellCons(m) ∧ ...
```

Listing 3.2: Operators to check well-defined data-type (static semantics)

Machine POs (behavioural semantics). The `Machine` data-type offers operators to access and handle its components. In addition to structural consistency, machine correctness is also encoded, through its behavioural semantics and correctness criteria. Formally, this is done by providing an operator for each PO of Event-B (see Table 3.2), as shown in Listing 3.3. Such operators are usually defined inductively on the structure of a machine (for initialisation and progress events).

```
  Mch_THM <predicate> ...
  Mch_INV_Init <predicate> (m : Machine(STATE, EVENT))
    direct definition   AP(m) ⊆ Inv(m)
  Mch_INV_One_Ev <predicate> (m : Machine(STATE, EVENT), e : EVENT)
    well-definedness e ∈ Progress(m)
    direct definition BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]] ⊆ Inv(m)
  Mch_INV <predicate> (m : Machine(STATE, EVENT))
    direct definition
      Mch_INV_Init(m) ∧ (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
  Mch_FIS_Init <predicate> (m : Machine(STATE, EVENT))
    direct definition Inv(m) ∩ AP(m) ≠ ∅
  Mch_FIS_One_Ev <predicate> (m : Machine(STATE, EVENT), e : Event)
```

```
  well−definedness e ∈ Progress(m)
  direct definition Inv(m) ∩ Grd(m)[{e}] ⊆ dom(BAP(m)[{e}])
Mch_FIS <predicate> (m : Machine(STATE, EVENT))
  direct definition
    Mch_FIS_Init(m) ∧ (∀e · e ∈ Progress(m) ⇒ Mch_FIS_One_Ev(m, e))
Mch_VARIANT_One_Ev <predicate> ...
Mch_VARIANT <predicate> ...
Mch_NAT_One_Ev <predicate> ...
Mch_NAT <predicate> ...
```

Listing 3.3: Well defined Data-type operators (behavioural semantics)

The details of the invariant preservation (INV - 3 and 4 in Table 3.2) and feasibility (FIS - 5 in Table 3.2) POs are shown in Listing 3.3. Three operators are associated to the definition of these POs: `Mch_INV_Init`, stating that an invariant holds at initialisation (i.e., states after the AP are included in the invariant states, $AP(m) \subseteq Inv(m)$); `Mch_INV_One_Ev`, stating that any event e characterised by its guard and BAP preserves the invariant (e.g. the image of invariant states through BAP is included in invariant states, $BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$); and `Mch_INV`, the conjunction of these two operators, where `Mch_INV_One_Ev` must hold for all progress events. Similarly, three operators `Mch_FIS_Init`, `Mch_FIS_One_Ev` and `Mch_FIS_Init` define the event feasibility PO (existence of a next state after AP or BAP of progress events). The other POs in Table 3.2 are defined in the same manner.

The POs of an Event-B machine are gathered in the conjunctive predicate `check_Machine_Consistency`, with `Machine_WellCons` as well-definedness (see Listing 3.4). It formalises machine's behavioural semantics and general correctness.

```
check_Machine_Consistency <predicate> (m : Machine(STATE, EVENT))
  well−definedness Machine_WellCons(m)
  direct definition Mch_THM(m) ∧
                    Mch_INV(m) ∧ Mch_FIS(m) ∧
                    Mch_VARIANT(m) ∧ Mch_NAT(m)
```

Listing 3.4: Operator encoding Event-B machine consistency

When this operator is used in a `theorem` clause, two POs, corresponding to its definition and WD condition, are automatically generated. Proving the theorem ensures the consistency of the machine, defined as an instance of the meta-theory.

Instantiation of the meta-theory. Specific Event-B machines are defined by instantiating the meta-theory. The instantiation process presented in this paper is so-called *deep*, as it relies *solely* on set theory and FOL with a set of axioms and theorems. It consists in defining an Event-B context with witnesses (sets) for type parameters `STATE` and `EVENT` defined as sets using `Cons_machine`. Operators may be used in theorems, triggering the generation of POs ensuring machine consistency. Another instantiation process qualified as *shallow* has also been defined [Riviere et al., 2022a, 2022b]. It relies on the definition of an Event-B machine and its refinement. It is not reviewed here as it is not used in this paper.

3.3.2 The Clock Example

This section presents a case study adapted from Lamport's clock case study [Lamport, 2002a]. It is used to demonstrate the application of the proposed frame-

```
MACHINE Clock
VARIABLES m, h
INVARIANTS
  inv1: m ∈ ℕ ∧ h ∈ ℕ
  inv2: m < 60 ∧ h < 24
EVENTS
  INITIALISATION
  THEN act1: m, h :| m' = 0 ∧ h' = 0
  END
  tick_min
  WHERE grd1: m < 59
  THEN act1: m :| m' = m + 1
  END
  tick_hour
  WHERE grd1: m = 59 ∧ h < 23
  THEN act1:
       m, h :| m' = 0 ∧ h' = h + 1
  END
  tick_midnight
  WHERE grd1: m = 59 ∧ h = 23
  THEN act1: m, h :| m' = 0 ∧ h' = 0
  END
END
```

Listing 3.5: Clock as Event-B machine

```
CONTEXT ClockMachineInstance
SETS Ev, ℤ × ℤ
CONSTANTS clock, tick_min, tick_hour,
    tick_midnight, init
AXIOMS
  axm1: clock ∈ Machine(ℤ × ℤ, Ev)
  axm2: partition(Ev, {init}, {tick_midnight},
        {tick_hour}, {tick_min})
  axm3: State(clock) = ℤ × ℤ
  axm4: Event(clock) = Ev
  axm5: Init(clock) = init
  axm6: Inv(clock) = {m ↦ h | m ∈ ℕ ∧ h ∈ ℕ
                  ∧m < 60 ∧ h < 24}
  axm7: AP(clock) = {m ↦ h | m = 0 ∧ h = 0}
  axm8: Grd(clock) = {e ↦ (m ↦ h) |
      (e = tick_min ∧ m < 59)∨
      (e = tick_hour ∧ m = 59 ∧ h < 23)∨
      (e = tick_midnight ∧ m = 59 ∧ h = 23)}
  axm9: BAP(clock) =
      {e ↦ ((m ↦ h) ↦ (m' ↦ h')) |
      (e = tick_min ∧ m' = m + 1 ∧ h' = h)∨
      (e = tick_hour ∧ m' = 0 ∧ h' = h + 1)∨
      (e = tick_midnight ∧ m' = 0 ∧ h' = 0)}
  ...
THEOREMS
  thm1: check_Machine_Consistency(clock)
END
```

Listing 3.6: Clock as meta-theory instance

work, including meta-theory instantiation and definition of new POs. Note that this simple case study is chosen to demonstrate the usability of the new extended mechanism.

The functional requirements of the clock state that minutes and hours progress by 1 and hours are represented in a 24-hour format. The clock must converge to midnight, and never stop. Listing 3.5 gives a model of the clock as an Event-B machine. In this model, variables m and h represent minutes and hours, respectively. A safety property ($inv2$) ensures that minutes m (resp. hours h) are always less than 60 (resp. 24). The clock's behaviour is expressed through three events: `tick_min` (progressing minutes by 1), `tick_hours` (progressing hours by 1) and `tick_midnight` (resetting the clock to midnight).

While the previous example does not show parameterised events, however, our approach handles such events. The same approach has been successfully applied to complex case studies in [Mendil et al., 2022] for critical interactive systems.

3.3.3 The clock machine as an instance of *EvtBTheo* theory

Listing 3.6 shows the Event-B context `ClockMachineInstance` instantiating the meta-theory `EvtBTheo`. First, $axm1$ defines the *clock* machine with the sets Ev (set of events enumerated in $axm2$) and $\mathbb{Z} \times \mathbb{Z}$ (for m and h). $axm3 - axm9$ define associated machine components. Note that invariant is defined ($axm6$) on the state as a set of pairs $m \mapsto h$, AP is defined on the initialisation event $axm7$ and guards and BAPs are associated with an event and a state and defined ($axm8$ and $axm9$) on a set of triples $e \mapsto m \mapsto h$. In the case of BAPs, it is necessary to

record before $(m \mapsto h)$ and after $(m' \mapsto h')$ states $(axm9)$.

Last, theorem $thm1$ uses `check_Machine_Consistency` (see Listing 3.4). It is associated with a well-definedness (WD) PO, $Machine\_WellCons(clock)$, and a theorem (THM) PO for machine correctness.

3.4 POs for new properties: Extending the Meta-Theory

The meta-theory *EvtBTheo* presented in Section 3.3.1 is highly extensible: every Event-B feature is explicitly formalised, and can be manipulated using operators, making it possible to define specific development operations or new reasoning mechanisms as new operators. Doing so is *non-intrusive* (self-contained), in the sense that no modification is needed to the classical development of Event-B models, as machines are handled as instances of the meta-theory.

The main design principle for such Event-B machine analyses, including theories with required operators, definitions, and WD conditions, is given below.

3.4.1 Analysis principle: New POs

In the proposed extension to the EB4EB framework, a model analysis is defined as a PO and must meet two requirements: 1) it must be reusable, and 2) it must be generated automatically. The first requirement is met by formalising the PO at the meta-theory level, while the second one is met by leveraging automatically generated well-definedness (WD) and theorem (THM) POs.

Event-B machine analysis pattern. Listing 3.7 depicts a generic pattern for defining new POs for Event-B machine analysis. *Theo4PO* theory imports the meta-theory *EvtBTheo* and introduces a third, optional type parameter T_{Args} possibly needed by the analysis, depending on the nature of new POs (e.g. guards, BAP, etc.). The PO associated to the analysis is formalised as a predicate operator `[PO]_Definition`. Then, checking the PO is done using the `check_Machine[PO]` predicate, which is well-defined when machine m is consistent.

```
THEORY Theo4PO IMPORT EvtBTheo
TYPE PARAMETERS STATE, EVENT, T_Args
OPERATORS
  [PO]_Definition <predicate> (m : Machine(STATE, EVENT), args : T_Args)
    well-definedness condition ...
    direct definition ...
  check_Machine_[PO] <predicate> (m : Machine(STATE, EVENT), args : T_Args)
    well-definedness condition Machine_WellCons(m)
    direct definition [PO]_Definition(m, args)
END
```

Listing 3.7: Analyses Theory Pattern

```
CONTEXT MachinePO
EXTENDS MachineInstance
THEOREMS
  thmPO: check_Machine_[PO](m, args)
END
```

Listing 3.8: Analyses Machine

Checking PO context pattern. Listing 3.8 shows an Event-B context pattern for checking the newly defined PO. A consistent instance machine context *MachineInstance*, that defines the Event-B machine m by instantiation of the meta-theory *EvtBTheo*, is extended by context *MachinePO* instantiating the extended theory *Theo4PO*. Theorem *thmPO* performs the check of the defined PO for machine m. The associated WD and THM POs are automatically generated.

Following this idea, this section introduces new reasoning mechanisms, not natively present in Event-B, based on the EB4EB framework and the *EvtBTheo* meta-theory, in the form of analyses that handle Event-B components. Three analyses are detailed: *deadlock-freeness*, *invariant weakness analysis* (tracking model holes) and *reachability*. The key points of using this framework are that: 1) WD conditions ensure elements are used correctly, 2) meta-properties on these analyses are established once and for all, and 3) these analyses can be performed without altering the machine's behaviour, in a non-intrusive way.

Note that only the definition of the *[PO]_Definition* operator is given, as *check_Machine_[PO]* is derived by replacing `[PO]` with the proposed PO name.

3.4.2 Deadlock-freeness

Requirements. Deadlock-freeness states that a machine m can always progress; i.e., there is always at least one enabled event in machine m, or more formally when the invariant holds then the disjunction of the guards holds.

PO Definition. The PO states that, for a machine m, there exists a progress event e such that any correct state $s \in Inv(m)$ verifies the guard of e ($s \in Grd(m)[\{e\}]$). When expressed using the meta-theory operators, it is formalised as $Inv(m) \subseteq Grd(m)[Progress(m)]$. This operator does not require any additional argument for $args$.

```
THEORY Theo4Deadlock IMPORT EvtBTheo
TYPE PARAMETERS STATE, EVENT
OPERATORS
  DeadlockFreeness_Definition <predicate> (m : Machine(STATE, EVENT))
    direct definition Inv(m) ⊆ Grd(m)[Progress(m)]
  ...
END
```

Listing 3.9: DeadlockFree Theory

Following the defined pattern, Listing 3.9 introduces a new theory `Theo4Deadlock` with two new operators together with the required WD condition.

```
CONTEXT ClockDeadlockFree
EXTENDS ClockMachineInstance
THEOREMS
  thmDeadlock: check_Machine_DeadLock(clock)
END
```

Listing 3.10: Clock DeadlockFreeness

Deadlock-freeness PO for Clock model. Listing 3.10 shows the context with *thmDeadlock* theorem generating WD and THM POs of the *clock* machine.

3.4.3 Invariant Weakness as a Non-intrusive Analysis

Requirements. A deployed system may present a number of vulnerabilities, that can be exploited by opponents (or make it weak to the environment) to modify its behaviour. These vulnerabilities usually come from *under-specification*, i.e., "holes" in the system's requirements or in its formal specification. To address this issue, a non-intrusive analysis of the model's specification is implemented, that does not alter its behaviour. It consists in investigating the robustness of the model's invariants with regard to *bad-events*, that model potential attacks (under-specification) against the system (model holes). *If the system's invariant is preserved by the bad-event*, it implies that *the invariant is* ***not strong enough*** to prevent the attack. For instance, the bad event of Listing 3.11 can be added to the *clock* machine without falsifying its original invariant. Similarly, other bad-events may be introduced: the event `tick_H5` guarded by $h < 19$ with action $m, h :| m' = 0 \land h' = h + 5$ and the event `tick_HM1` guarded by $h < 23 \land m < 59$ with action $m, h :| m' = m + 1 \land h' = h + 1$. Note that a class of bad events could be added using two parameters $hn \neq 1$ and $mn \neq 1$ and a corresponding action of the form $m, h :| m' = m + mn \land h' = h + hn$.

```
tick_M5
WHERE grd1 : m < 55
THEN act1 : m :| m' = m + 5
END
```

Listing 3.11: An Bad-event: progress by 5 min.

Bad-events PO definition. This PO is formalised with the `AllowedMachineHoleSub_Definition` operator (Listing 3.12), with the bad-events as parameters.

```
THEORY EvtBTheorySubs IMPORT THEORY EvtBTheory
TYPE PARAMETERS STATE, EVENT
OPERATORS
  AllowedMachineHoleSub_Definition <predicate> (m : Machine(STATE, EVENT),
      nGrd : ℙ(STATE), nBAP : ℙ(STATE × STATE))
    direct definition nBAP[Inv(m)∩nGrd]⊆Inv(m)
  ...
END
```

Listing 3.12: Weak specification analysis theory

Each bad-event is characterised by its guard $nGrd$ and its BAP $nBAP$. This operator defined as $nBAP[Inv(m) \cap nGrd] \subseteq Inv(m)$ states that the bad-event preserves the invariant. So, if the given PO is proved, the bad-event represents a successful attack, and the defined invariant is not strong enough.

Bad events PO for clock model. The analysis to Check the clock specification forbids minutes from progressing by 5 rather than 1, is handled by theorem *thmInspectInvEVTM5* of Listing 3.13, using the `AllowedMachineHoleSub_Definition` operator, where the bad-event is enabled when minutes are below 55 and thus progresses by 5. This corresponds to adding event `tick_M5` of Listing 3.11. Similar theorems are written for the `tick_H5` and `tick_HM1` bad-events.

```
CONTEXT ClockInspectInv EXTENDS ClockMachineInstance
THEOREMS
  thmInspectInvEVTM5: check_Machine_AllowedMachineHoleSub(clock,
        {m ↦ h | h ∈ ℤ ∧ m < 55},
        {(m ↦ h) ↦ (m' ↦ h') | m' = m + 5 ∧ h' = h ∧ h ∈ ℤ})
  thmInspectInvEVTH5: check_Machine_AllowedMachineHoleSub···
  thmInspectInvEVTHM1: check_Machine_AllowedMachineHoleSub···
END
```

Listing 3.13: Performing analysis on clock model

Note that the *thmInspectInvEVTM5*, *thmInspectInvEVTH5* and *thmInspectInvEVTHM1* theorems are proven for the *clock* model of the *ClockMachineInstance* corresponding to the Event-B machine of Listing 3.5. As a conclusion, the original model is insufficiently strong and does not provide sufficient constraints on the safe evolution of variables.

A strengthened machine. The designer strengthens the original machine, through instantiation, resulting in the new model shown in Listing 3.14. New state variables mb and hb are introduced to explicitly record the value of minutes and hours before a tick event occurs. In addition, the events are required to explicitly link these variables as $m = mb + 1$ and $h = hb + 1$.

CONTEXT ClockInvStrong
SETS Ev, $\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
CONSTANTS clock, tick_min, tick_hour, tick_midnight, init
AXIOMS
axm1: $clock \in Machine(\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}, Ev) \ldots$
axm2: ...
axm3: $State(clock) = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
axm4−5: ...
axm6: $Inv(clock) = \{m \mapsto h \mapsto mb \mapsto hb \mid m \in \mathbb{N} \wedge h \in \mathbb{N} \wedge m < 60 \wedge h < 24 \wedge$
$(\boldsymbol{m = mb + 1 \wedge hb = h}) \vee (\boldsymbol{m = 0 \wedge (h = hb + 1 \vee h = 0)})\}$
axm7: $AP(clock) = \{m \mapsto h \mapsto mb \mapsto hb \mid m = 0 \wedge h = 0 \wedge \boldsymbol{mb \in \mathbb{Z} \wedge hb \in \mathbb{Z}}\}$
axm8: $BAP(clock) = \{t \mapsto ((m \mapsto h \mapsto mb \mapsto hb) \mapsto (m' \mapsto h' \mapsto mb' \mapsto hb')) \mid$
$(t = tick\_min \wedge m' = m + 1 \wedge h' = h \wedge \boldsymbol{hb' = h \wedge mb' = m \wedge mb \in \mathbb{Z} \wedge hb \in \mathbb{Z}}) \vee$
$(t = tick\_hour \wedge m' = 0 \wedge h' = h + 1 \wedge \boldsymbol{hb' = h \wedge mb' = m \wedge mb \in \mathbb{Z} \wedge hb \in \mathbb{Z}}) \vee$
$(t = tick\_midnight \wedge m' = 0 \wedge h' = 0 \wedge \boldsymbol{hb' = h \wedge mb' = m \wedge mb \in \mathbb{Z} \wedge hb \in \mathbb{Z}})\}$
...
THEOREMS
thm1: $check\_Machine\_Consistency(clock)$
thmInspectInvEVTM5: $\neg check\_Machine\_AllowedMachineHoleSub(clock,$
$\{m \mapsto h \mapsto mb \mapsto hb \mid mb \in \mathbb{Z} \wedge hb \in \mathbb{Z} \wedge h \in \mathbb{Z} \wedge \boldsymbol{m < 55}\},$
$\{(m \mapsto h \mapsto mb \mapsto hb) \mapsto (m' \mapsto h' \mapsto mb' \mapsto hb') \mid$
$\boldsymbol{m' = m + 5} \wedge h' = h \wedge hb' = h \wedge mb' = m \wedge mb \in \mathbb{Z} \wedge hb \in \mathbb{Z}\})$
thmInspectInvEVTH5: $\neg check\_Machine\_AllowedMachineHoleSub \ldots$
thmInspectInvEVTMH1: $\neg check\_Machine\_AllowedMachineHoleSub \ldots$
END

Listing 3.14: Clock resulting after the strengthening of the invariant

To guarantee that the identified bad-events are no longer triggerable, the predicates are negated in `thmInspectInvEVTM5`, `thmInspectInvEVTH5` and `thmInspectInvEVTHM1`. These theorems are proven to hold, demonstrating that the provided specification prohibits the presented inconsistent behaviour.

3.4.4 Reachability

Requirements. The reachability property is not natively available in Event-B. Such a property can be expressed using the EB4EB framework. Reachability property asserts that particular states can be attained under given constraints. The definition used below asserts that there exists a trace where a given state is reachable. This definition differs from the eventually operator of LTL. Note that a formalisation of the eventually operator of LTL is available in [Mendil et al., 2022; Riviere et al., 2023a].

```
THEORY Theo4Reachability IMPORT THEORY EvtBTheory
TYPE PARAMETERS STATE, EVENT
OPERATORS
// At least one "trgSet" event is triggerable after "src" event
  At_Least_One_Triggerable_Evt <predicate> (m : Machine(STATE, EVENT),
      src : EVENT, trgSet : ℙ(EVENT)) ...
// All "SubSetEvt" events decrease the "variant"
  VariantDecrease <predicate> (m : Machine(STATE, EVENT), variant : ℙ(STATE × ℤ),
      SubSetEvt : ℙ(EVENT))...
// For all "SubSetEvt" events, the "variant" is a Natural number
  NaturalVariant <predicate> (m : Machine(STATE, EVENT), variant : ℙ(STATE × ℤ),
      SubSetEvt : ℙ(EVENT))...
// When "variant" is not null, there exists a "SubSetEvt" triggerable
    event
  One_Next_Evt_Is_Triggerable <predicate> (m : Machine(STATE, EVENT),
      variant : ℙ(STATE × ℤ), SubSetEvt : ℙ(EVENT))...
// "trg" event is reachable from "src" event through at least one "
    SubSetEvt" event
  Evt_Is_Reachable_From_Definition <predicate> (m : Machine(STATE, EVENT),
      src : EVENT, trg : EVENT, SubSetEvt : ℙ(EVENT), variant : ℙ(STATE × ℤ))
    well-definedness Machine_WellCons(m), trg ∈ Progress(m), src ∈ Event(m),
        Inv(m) ◁ variant ∈ Inv(m) → ℤ, Mch_INV(m), SubSetEvt ⊆ Progress(m)
    direct definition
      NaturalVariant(m, variant, SubSetEvt)∧ // Preserve the "variant" natural
      VariantDecrease(m, variant, SubSetEvt)∧ // "SubSetEvt" decrease the "
          variant"
      Next_Conv_Evt_Is_Triggerable(m, variant, SubSetEvt)∧ // the "variant" are
          always possible to decrease
      At_Least_One_Triggerable_Evt(m, src, SubSetEvt)∧ // "src" can trigger a "
          SubSetEvt"
      variant⁻¹[ℤ \ ℕ] ∩ Inv(m) ⊆ Grd(m)[{trg}] // "variant"=0 can trigger "trg"
  ...
END
```

Listing 3.15: Thoery of reachable property in Event-B

A *trace* σ of a machine m is a sequence of states $s_0, s_1, \ldots$ where s_0 is in the AP of the initialisation event and, for two consecutive state s_i, s_{i+1} in the trace, s_i must satisfy the guards of at least one event and (s_i, s_{i+1}) must satisfy the before-after predicate of this event. For $k \geq 0$, $\sigma(k)$ denotes the k-th state s_k of the trace. Then, s_j is reachable from s_i (denoted $s_i \mathcal{R} s_j$) if and only if $\exists \sigma, k, n \cdot n \geq 0 \wedge \sigma(n) = s_i \wedge k > 0 \wedge \sigma(n+k) = s_j$.

Reachability PO definition. The reachability property $s_i \mathcal{R} s_j$ is encoded using the Event-B meta-theory (Listing 3.15). The `Theo4Reachability` theory begins by defining the `At_Least_One_Triggerable_Evt` predicate, which states that, for any state reached after the *source* event, the guard of at least one *target* event is enabled. Then, the predicates `VariantDecrease` and `NaturalVariant` are defined. The former is satisfied only if, for machine m, each event of the *SubSetEvt* set decreases the given *variant*; the latter ensures that the guards of the *SubSetEvt* events imply that the variant is a natural number. The `One_Next_Evt_Is_Triggerable` predicate evaluates to true in machine m if the given *variant* is positive and at least one event in *SubSetEvt* is activated.

These four operators formalise the induction-based definition of reachability. They are used to define the main predicate, `Evt_Is_Reachable_From_Definiton`, stating that, in machine m, target event *trg* can be triggered after a (finite) sequence of *SubSetEvt* event triggers for the given *variant*, beginning with *src* event. Formally, triggering *src* activates at least one event in *SubSetEvt* and each

event of $SubSetEvt$ decreases the variant and enables at least one other event of $SubSetEvt$, and then trg is enabled when the variant reaches 0.

CONTEXT ClockReachability **EXTENDS** ClockMachineInstance
THEOREMS
thmReach: $check\_Machine\_Evt\_Is\_Reachable\_From(clock, init, tick\_midnight, \{tick\_min, tick\_hour\}, \{m \mapsto h \mapsto v \mid v = 24 * 60 - 2 - (m + h * 24)\})$
END

Listing 3.16: Clock machine with a reachable property checked

Clock machine reachability PO for clock model. In the clock model of Listing 3.6, it is worth checking that midnight is reachable from the initial event. This analysis is performed with theorem `thmReach` (see Listing 3.16), that checks whether the event *tick_midnight* is reachable from the event *init*, via events *tick_min* and *tick_hours*. The proposed variant is then $v = 24 * 60 - 2 - (m + h * 24)$. Proving the generated POs for this theorem establishes reachability.

3.4.5 Proof assessment

The defined operators of the proposed framework have been designed in the spirit of Event-B, i.e., 1) complex analyses are decomposed into simple ones (case of reachability in Section 3.4.4) and 2) expressed in a single semantic setting: the one of Event-B (reflexive modelling) with set theory. This formalisation is influenced by two characteristics of the proof process, that 1) the Rodin prover is efficient when handling set expressions, and 2) theories may define customised *proved rewrite rules*, that may be summoned manually or automatically in the proof. Automatic rewriting rules that substitute operators by definitions are automatically generated. These rules are written to extract relevant information from machine objects, add them to the hypotheses, and produce multiple simpler goals. They are defined to be applied automatically and chained together, greatly improving proof automation. Indeed, these rewrite rules are included in Rodin's user-defined proof tactics, once and for all, increasing automation when proving the theorems formalising the newly defined POs.

Model	PO	Max Depth	Nodes	Interactive Nodes	Number of Tactic application
DeadlockFree clock	thmDeadlock (THM)	169	221	1	2
Reachability clock	thmReach (WD)	112	577	0	1
	thmReach (THM)	191	731	4	5
Inspect Inv clock	thmInspectInvEVTM5 (THM)	111	167	0	1
	thmInspectInvEVTH5 (THM)	112	169	0	1
	thmInspectInvEVTMH1 (THM)	113	171	0	1
Strong Inv clock	thmInspectInvEVTM5 (THM)	105	158	0	1
	thmInspectInvEVTH5 (THM)	118	171	0	1
	thmInspectInvEVTHM1 (THM)	128	181	0	1

Table 3.3: Proof statistic for the Clock model and its analyses

Table 3.3 presents the proof statistics for each analysis. The important number of nodes (representing atomic steps) in the proof trees is due to the extensive use of theory operators which the prover cannot handle directly, and thus their definitions must be unfolded. The introduction of the rewrite rules in a proof tactic perform automatically these unfold and reductions, making almost all steps fully automatic despite the introduction of the meta level (An entry of 0 in the interactive nodes column of Table 3.3). The rightmost column provides the number of tactic applications (iterations) during the proof. Indeed, a single tactic application may not be sufficient to fully discharge the proof goals.

3.5 Positioning this approach

3.5.1 Related work

Formalising model analyses has been addressed by several authors: Riccobene et al. [Riccobene & Scandurra, 2004] presented the ASM-Metamodel (AsmM) for Abstract State Machine (ASM) models considering core modelling constructs and semantics, expressed as an API manipulating ASM-related concepts like abstract machines, signatures, terms, rules, and so on. It is used to embed ASM in another formal method. This work resulted in a number of analyses, tools, and extensions for a variety of purposes [Gargantini et al., 2008]. A similar approach exists for VDM with MURAL, an interactive mathematical reasoning environment extended to support VDM [Bicarregui & Ritchie, 1991] specifications based on meta-modelling concepts, and designed to offer a theorem prover for VDM models. Similarly, the Rodin tool offers an API for handling Event-B models, intended to be used to develop plug-ins. This API is used by ProB [Leuschel & Butler, 2003] as well as by plug-ins handling model development [T. S. Hoang et al., 2017] and code generation [Fürst et al., 2014; Méry & Singh, 2011].

Ebner et al. [Ebner et al., 2017] described the meta-programming framework used in Lean, which is an interactive theorem prover based on dependent type theory. This framework provides a means for reflecting object-oriented expressions into a meta-language by extending Lean's object language, based on Lean's modelling constructs. In [Paul van der Walt, 2012], the authors present reflection in Agda in the style of Lisp, MetaML, and Template Haskell, as well as several typed programming applications. The MetaCoq [Sozeau et al., 2020] project proposed a certified meta-programming environment in Coq based on meta-modelling Coq concepts, including typing and operational semantics. This certified meta-modelling environment was also used in the development of the CertiCoq [Anand et al., 2017] certified compiler project. Similarly, this reflection principle [Fallenstein & Kumar, 2015] is implemented in Isabelle/HOL to build a HOL model within HOL to analyse and reason about various modelling concepts such as infinite hierarchy of large cardinals, polymorphism, verifying systems with self-replacement functionality, etc. In PVS, Miltra et al. [Mitra & Archer, 2005] proposed *strategies* for proving abstraction relations between automata, based on theories and templates. This mechanism generalises proofs, making them highly

reusable. With regard to Event-B, the formalisation of *contexts* (and only contexts) in the Event-B language has been proposed [Bodeveix & Filali, 2021]. In related approaches, the B method has been embedded in PVS [Muñoz & Rushby, 1999], to benefit from the modelling power of B, while accessing the proving power of the PVS theorem prover. However, this embedding is not formalised, and leads to the use of two separate methods.

Abstract interpretation showed its power to check system properties (absence of runtime errors, dead code, ...). Frameworks like [Brat et al., 2014; Bühler, 2017; Cousot et al., 2005] apply to programs through the definition of parameterised abstract domains corresponding to model analyses. The correctness of these analyses is expressed outside the framework.

The proposed approach is based on reflecting Event-B in itself i.e. its elements can be used as first-class objects in models. This is similar in Coq and HOL based approaches using dependent types, except that 1) it relies on set theory and FOL, easing transfer to other formalisms and 2) it is defined in the same setting as the state-transitions model of the system to be designed.

3.5.2 Advantages of the approach

This paper highlights several advantages of the EB4EB framework.
- Formal modelling and verification integrated in EB4EB. This framework enables the simultaneous use of two approaches for both modelling (operational with machines or axiomatic with contexts) and proving (meta-theory-based and model/induction-based) allowing users to use one or the other non-intrusively on pre-existing models. The proposed theories of the EB4EB framework can be easily extended following the methodology introduced in this paper, to handle new reusable models analyses by introducing, in Event-B, new *automatically generated POs* that preserve the semantics of Event-B.
- Easing proof process. The EB4EB reflexive framework enables the explicit manipulation of Event-B components by introducing meta-elements such as required datatypes, operators and theorems, extremely useful for expressing complex problems as well as proposing new reasoning mechanisms. However, due to the lack of advanced level proof engines such as SMTs, this resulted in enormous manual proof efforts. The introduced proved proof rules reduce interactive proof efforts while increasing proof automation.
- On-the-fly analysis. The EB4EB framework, which includes reasoning extensions, enables on-the-fly model analysis as well as advanced reasoning level for each Event-B model in the refinement chain. Note that the majority of Event-B models consist of several refinement layers, where each model of a given abstraction level can be analysed; i.e, the model is *lifted* as an instance of the EB4EB meta-level and is submitted for performing model analyses, at an advanced reasoning level, ensured by new POs generation.
- Correctness of the defined analyses. The EB4EB framework associates a *trace* to any Event-B machine (trace-based semantics). Such semantics is used to prove the correctness of the defined analyses. Indeed, a theorem stating that the property specifying a given model analysis holds on the traces of a machine

is defined for this purpose. Such a correctness theorem has been proved for each of the analyses introduced in Section 3.4.

3.6 Conclusion

This paper presented a technique allowing a designer to define new POs for Event-B corresponding to model analyses that are not available in core Event-B. It is based on the extension of the reflexive EB4EB framework and its meta-theory *EvtBTheo*. The defined extended reasoning mechanisms and POs are not available in core Event-B. They have been defined as Event-B meta-modelling concepts allowing to express deadlock-freeness, bad-events and invariant strengthening, and reachability. It is demonstrated that non-intrusive analysis for Event-B models formalised in Event-B can be performed, at any abstraction level in the refinement chain, and without resorting to another formal method, which would require additional proofs to ensure the correct embedding of Event-B in that method. Moreover, the proof process has been enriched with relevant and proved rewrite rules, included in tactics, leading to a high level of proof automation. All the developments shown in this paper are completely formalised and all the proofs are realised[2].

Two future directions extending this work have been identified. The first one consists in defining domain-specific engineering theories in order to define specific domain-oriented properties as POs to be satisfied by system models. Such an approach opens towards standard conformance and certification. The second future direction exploits the fact that EB4EB defines an Event-B machine as an instance of a meta-theory as a set of axioms and theorems instances in FOL and set theory. This format can be exported into the higher order framework Dedukti [Boespflug et al., 2012; Dowek, 2015], and thus makes way for the design of correct import in, and export from Event-B of formal models through Dedukti.

[2]https://www.irit.fr/~Peter.Riviere/models/

Assessment

This chapter contributes by extending the EB4EB framework to provide reasoning mechanisms and advanced characteristics not present in native Event-B. A new theory is developed, consisting of a collection of new operators and axiomatic definitions, as well as new proof rules. Three key analysis techniques, *deadlock-freeness*, *invariant weakness analysis*, and *reachability*, are presented, and Lamport's clock example is used to demonstrate each of them.

The key advantage of this extension in the EB4EB framework is the ability to scale Event-B reasoning capabilities simply by specifying new reasoning mechanisms. In addition, this extension allows non-intrusive analysis of Event-B models, as well as the possibility to formalise modelling techniques usual in Event-B, but not natively supported by the method explicitly (i.e., deadlock-freeness). There are two main limitations to this extension. First, there is a lack of proof automation that can be improved by defining new generic proof rules, and a lack of refinement preservation for performing non-intrusive analysis.

Chapter 4

Extending Event-B with Temporal Logic

Overview

There is a lack of support for expressing and verifying both temporal and liveness properties in core Event-B. However, such properties may be validated using other external tools such as model checkers. The main contribution of this chapter is to extend the EB4EB framework to support the modelling and verification of temporal properties. This extension enables the production of the required proof obligations, and aid in their proving process. This work was inspired by the work of Hoang and Abrial [T. S. Hoang & Abrial, 2011]. Similarly to our previous developments, a set of new operators is specified in algebraic form to encode basic temporal properties and composition of temporal properties, and the necessary well-defined conditions are provided to ensure the correct usage of the said operators. Furthermore, to ensure correctness, all defined operators are encoded in trace-based semantics, to assess that the proof obligations indeed correspond to the expected behaviour. The whole development is also carried out in the Event-B modelling language with the *Theory plugin* and Rodin IDE, and a set of new proof tactics are added to support the proving processes. This approach is demonstrated on the same case study presented in the initial paper of Hoang and Abrial.

Associated paper of this chapter:

- Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. [2023a]. Formalising liveness properties in Event-B. *NASA Formal Methods 2023*

Formalising Liveness Properties in Event-B with the Reflexive EB4EB Framework

P. Rivière, N. K. Singh, Y. Aït-Ameur, G. Dupont [1]

INPT-ENSEEIHT/IRIT, University of Toulouse, France
{peter.riviere, nsingh, yamine, guillaume.dupont}@enseeiht.fr

The correct-by-construction state-based Event-B formal method lacks the ability to express liveness properties using temporal logic. To address this challenge, two approaches can be envisioned. First, embed Event-B models in another formal method supporting liveness properties verification. This method is cumbersome and error-prone, and the verification result is not guaranteed on the source model. Second, extend Event-B to support the expression of and reasoning on liveness properties, and more generally temporal properties. Following the second approach, in [T. S. Hoang & Abrial, 2011], J.-R. Abrial and T. S. Hoang proposed an axiomatisation of linear temporal logic (LTL) for Event-B with a set of proof obligations (POs) allowing to verify these properties. These POs are mathematically formalised, but are neither implemented nor generated automatically. In this paper, using the reflexive EB4EB framework [Riviere et al., 2022a, 2022b] allowing for manipulation of the core concepts of Event-B, we propose to formalise and operationalise the automatic generation of proof obligations associated to liveness properties expressed in LTL. Furthermore, relying on trace-based semantics, we demonstrate the soundness of this formalisation, and provide a set of intermediate and generic theorems to increase the rate of proof automation for these properties. Finally, a case study is proposed to demonstrate the use of the defined operators for expressing and proving liveness properties.

Proof and state-based methods, Event-B and Theories, Meta-theory, Reflexive EB4EB framework, Temporal logic, Liveness properties, Traces and soundness

4.1 Introduction

Event-B is a formal method based on explicit state expression, refinement and formal proof. It enables the design of complex systems using a correct by construction approach. This method has been used successfully for the design of many complex systems in various engineering areas such as aeronautics [Su & Abrial, 2017], railway systems [M. J. Butler et al., 2017, 2020], health and medicine [Singh, 2013], etc. In particular, it has shown its effectiveness in establishing properties related to system functionalities, safety, security, reachability, compliance with some temporal requirements, and so on.

Event-B models are machines that express state-transition systems using set theory and first-order logic (FOL). A mechanism of proof by induction enables the demonstration of inductive properties based on the preservation of properties

[1] The authors thank the ANR-19-CE25-0010 *EBRP:EventB-Rodin-Plus* project.

at initialization and by each transition (event). Refinement, on the other hand, is defined by a weak simulation relation in which proof obligations guarantee the preservation of behaviours between levels of abstraction. The Rodin platform supports the development of Event-B models. It offers an environment for model editing, automatic and interactive proofs, animation, model checking, etc.

However, Event-B, like every formal methods, lacks some capabilities. It supports the verification of a fragment of temporal logic properties: $\Box$ using invariants and theorem clauses and $\Diamond$ using variants and convergence proof obligations. However, there is a lack of composition of temporal logic operators, as well as the ability to express and reason about liveness properties. To remedy this absence, two solutions are possible in general. The first solution consists in embedding an Event-B model in another formal method offering the possibility of expressing and reasoning about liveness properties such as TLA$^+$ [Lamport, 2002a], NuSMV [Cimatti et al., 2002], PRISM [Kwiatkowska et al., 2011], PAT [Sun et al., 2009], Spin [Holzmann, 2003], Uppaal [Behrmann et al., 2004], ProB [Leuschel & Butler, 2008] etc. However, tracing the verification results on the source Event-B models is difficult and care must be taken to guarantee the correctness of this embedding. This approach is very popular and is followed by many authors who use other formal methods allowing to express and verify this type of property without worrying about the correctness of the transformation. However, there exist several approaches to ensuring the transformation's correctness [Bodeveix et al., 2015; Halchin et al., 2020; Leroy et al., 2016; Pnueli et al., 1998]. The second solution consists in extending the Event-B method to allow the expression of and reasoning on liveness properties. This second approach requires the expression of the semantics and the proof system of the temporal logic in Event-B, as well as establishing the soundness of this extension.

Based on the second approach, JR. Abrial and TS. Hoang [T. S. Hoang & Abrial, 2011] proposed an axiomatisation of linear temporal logic (LTL) for Event-B in their article entitled *"Reasoning about liveness properties in Event-B"*. This work has defined a set of proof obligations allowing to establish temporal properties such as reachability, progress, persistence or until. However, these proof obligations are mathematically formalised in that paper but are neither implemented nor generated automatically. They must be explicitly described in Event-B by the developer for each model, thus leading to formalization errors. Moreover, their proofs are cumbersome and require too much manual effort to proving them.

Relying on the reflexive EB4EB framework [Riviere et al., 2022a, 2022b, 2023c] defined in Event-B, we propose to formalise and operationalise the automatic generation of proof obligations associated with liveness properties expressed in LTL temporal logic. We define an extension of EB4EB including a set of operators expressing these properties on traces. In addition, we demonstrate the soundness of these properties on model traces. Finally, a set of intermediate and generic theorems are also proposed to increase the rate of proof automation.

Note that our proposed approach is non-intrusive (self-contained) and does not require the use of any other formal techniques or tools; it is fully formalised in Event-B and mechanised on the Rodin platform.

Context	Machine	Theory
CONTEXT Ctx **SETS** s **CONSTANTS** c **AXIOMS** A **THEOREMS** T_{ctx} **END**	**MACHINE** M **SEES** Ctx **VARIABLES** x **INVARIANTS** $I(x)$ **THEOREMS** $T_{mch}(x)$ **VARIANT** $V(x)$ **EVENTS** **EVENT** evt **ANY** α **WHERE** $G_i(x, \alpha)$ **THEN** $x :\mid BAP(\alpha, x, x')$ **END** ... **END**	**THEORY** Th **IMPORT** Th1, ... **TYPE PARAMETERS** E, F, ... **DATATYPES** **Type1**(E, ...) **constructors** **cstr1**(p_1: T_1, ...) **OPERATORS** **Op1** <nature> (p_1: T_1, ...) **well−definedness** $WD(p_1, ...)$ **direct definition** D_1 **AXIOMATIC DEFINITIONS** **TYPES** A_1, ... **OPERATORS** **AOp2** <nature> (p_1: T_1, ...): T_r **well−definedness** $WD(p_1, ...)$ **AXIOMS** A_1, ... **THEOREMS** T_1, ... **PROOF RULES** R_1, ... **END**
(a)	(b)	(c)

Table 4.1: Global structure of Event-B Contexts, Machines and Theories

This paper is organised as follows. Section 4.2 describes the Event-B modelling language and its Theory plugin extension. Section 4.3 recalls linear temporal logic, and the EB4EB framework is described in Section 4.4. Section 4.5 presents the trace-based semantics of Event-B, and its soundness properties. Section 4.6 describes a case study that will be used as a running example to show how to use defined LTL operators. Section 4.7 presents the temporal logic proof rules encoded as EB4EB proof obligations. Their correctness is discussed in Section 4.8. Section 4.9 summarises related work, and Section 4.10 concludes the paper.

4.2 Event-B

Event-B [J.-R. Abrial, 2010] is a state-based, *correct-by-construction* formal method, where systems are modelled with a set of events representing state changes, using first-order logic (FOL) and set theory.

Contexts and machines (Tables 4.1.a and 4.1.b). `Context`s (Table 4.1.a) encompass the model's *static* part: *carrier sets* s and *constants* c, as well as their properties, through *axioms* A and *theorems* T_{ctx}. `Machine`s (Table 4.1.b) describe the model's behaviour, using a set of *events* evt, each of which may be guarded G and/or parameterized by α. An event models the evolution of a set of variables x using a Before-After Predicate (BAP) that links the before (x) and after (x') value of the variables. Safety properties are encoded using *invariants* $I(x)$ and *theorems* $T_{mch}(x)$, and *variants* $V(x)$ may be defined to demonstrate the machine's convergence. Model consistency is established by discharging a number of automatically generated POs (Table 4.2).

Refinements. One strength of Event-B is its *refinement* operation, which is used to transform an abstract model into a more concrete one, adding information (refined states) and behavioural (refined events) details gradually, while retaining a similar observational behaviour (simulation relationship). Refinement correctness is established with the help of a gluing invariant, and ensures properties are

(1)	Ctx Theorems (ThmCtx)	$A(s,c) \Rightarrow T_{ctx}$ `(For contexts)`
(2)	Mch Theorems (ThmMch)	$A(s,c) \wedge I(x) \Rightarrow T_{mch}(x)$ `(For machines)`
(3)	Initialisation (Init)	$A(s,c) \wedge BAP(x') \Rightarrow I(x')$
(4)	Invariant preservation (Inv)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge BAP(x,\alpha,x') \Rightarrow I(x')$
(5)	Event feasibility (Fis)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \Rightarrow \exists x' \cdot BAP(x,\alpha,x')$
(6)	Variant progress (Var)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge BAP(x,\alpha,x') \Rightarrow V(x') < V(x)$

Table 4.2: Relevant Proof Obligations for Event-B contexts and machines

preserved from the abstract to the concrete model.

Extension with theories. Being based on set theory and FOL, the Event-B formalism is mathematically low-level and thus very expressive. However, it lacks features to build up more complex structures. The theory extension has been proposed to address this issue [M. J. Butler & Maamria, 2013]. A theory is a type of component that makes it possible to define new type-generic datatypes together with constructive and axiomatic operators, specific theorems and axioms and even proof rules (see Table 4.1.c). The resulting theories consistency can be established by providing witnesses for axioms and definitions, ensuring conservative extensions of Event-B. Once defined, elements of a theory become seamlessly available in an Event-B model and its proofs.

This extension is central for embedding, as data types, *concepts that are unavailable in core Event-B*, similar to Coq [Bertot & Castéran, 2010], Isabelle/HOL [Nipkow et al., 2002] or PVS [Owre et al., 1992]. Many theories have been defined, for supporting real numbers, lists, differential equations and so on.

Well-definedness (WD). Beyond machine-related POs, one key aspect of model consistency is the *well-definedness* (WD) of the expressions involved in it. This notion supplements the one of syntactical correctness with the idea of a formula being "meaningful", i.e. it can always be safely evaluated (e.g., dividing by a term that is provably non 0). Each formula of a model is associated to a WD PO, usually consisting in checking that operators are correctly used and combined. Once proven, WDs are added to set of hypotheses of other POs.

Note that theories allow designers to provide custom WD conditions for partially defined operators in order to precisely characterise their proper use.

The Rodin Platform. Rodin is an open source integrated development platform for designing, editing and proving Event-B models. It also supports model checking and animation with ProB, as well as code generation. Being based on Eclipse, it also allows the definition of *plug-ins*, including theory extensions. Many provers for first-order logic as well as SMT solvers are plugged to Rodin for helping the proof process.

4.3 Linear Temporal Logic

This section recalls the principles of linear temporal logic (LTL) following the definition of Manna and Pnueli [Manna & Pnueli, 1984]. Linear temporal logic is defined syntactically as an extension of propositional logic. A valid LTL formula

consists of literals (usually, predicates on the state of the system), the usual logical connectors ($\wedge$, $\vee$, $\neg$ and $\Rightarrow$) as well as *modal operators* $\Box$, $\Diamond$ and $\mathcal{U}$. The semantics of LTL is expressed in terms of *traces* of a system. Given a trace $tr = s_0 \mapsto s_1 \mapsto \ldots$, then tr_i ($i \in \mathbb{N}$) denotes the *suffix* trace of tr, starting from s_i, $tr_i = s_i \mapsto s_{i+1} \mapsto \ldots$

A state that satisfies a predicate P is called a P-state. LTL semantics are given with the following rules:

1. For any state predicate P, $tr \vDash P$ iff s_0 is a P-state.
2. $tr \vDash \phi_1 \wedge \phi_2$ iff $tr \vDash \phi_1$ and $tr \vDash \phi_2$
3. $tr \vDash \phi_1 \vee \phi_2$ iff $tr \vDash \phi_1$ or $tr \vDash \phi_2$
4. $tr \vDash \neg\phi$ iff not $tr \vDash \phi$
5. $tr \vDash \phi_1 \Rightarrow \phi_2$ iff not $tr \vDash \phi_1$ or $tr \vDash \phi_2$
6. $tr \vDash \Box\phi$ iff for all k, $tr_k \vDash \phi$
7. $tr \vDash \Diamond\phi$ iff there exists a i such that $tr_i \vDash \phi$
8. $tr \vDash \phi_1 \mathcal{U} \phi_2$ iff there exists a i such that $tr_i \vDash \phi_2$, and for all $j < i$, $tr_j \vDash \phi_1$

A machine M satisfies a property ϕ, denoted $M \vDash \phi$ if and only if for all traces tr of M, that trace satisfies ϕ ($tr \vDash \phi$).

4.4 The EB4EB Framework

The EB4EB framework [Riviere et al., 2022a, 2022b] proposes to extend the reasoning capabilities of Event-B by enabling the access of Event-B components as first-class citizens within Event-B models (reflection), thereby making it possible to express new reasoning mechanism at the meta-level.

```
THEORY EvtBTheo
TYPE PARAMETERS St, Ev
DATATYPES Machine(St, Ev)
 CONSTRUCTORS
    Cons_machine(
       Event : ℙ(Ev),
       State : ℙ(St),
       Init : Ev, Progress : ℙ(Ev)
       Variant : ℙ(St × ℤ),
       AP : ℙ(St),
       BAP : ℙ(Ev × (St × St)),
       Grd : ℙ(Ev × St),
       Inv : ℙ(St),
       ...)
```

Listing 4.1: Machine Data type

```
Event_WellCons <predicate>
      (m : Machine(St, Ev))
   direct definition
      partition(Event(m), {Init(m)}, Progress(m))
...
Machine_WellCons <predicate>
      (m : Machine(St, Ev))
   direct definition Event_WellCons(m) ∧ ...
```

Listing 4.2: Operators to check well-defined data type (static semantics)

Machine structure. Event-B is formalised in an Event-B theory. A machine is represented using the data-type `Machine` (see Listing 4.1) parameterised by generic types with event labels (`Ev`) and states (`St`). Constructor `Cons_machine` gathers the components of a machine, such as `Event`, `State`, `Grd`, `Inv`, `BAP`, etc.

Well-Construction. A machine built using `Cons_machine` may not be consistent, despite being syntactically correct. Thus, additional operators are defined to encode the *well-construction* of a machine, i.e. the consistency of its components with regard to each others (Listing 4.2). For instance, `Event_WellCons` ensures that events are partitioned between initialisation and progress events.

Machine Proof Obligations. For any machine expressed in the framework, its associated proof obligations are provided under the form of operators (see Listing 4.3). Such operators are predicates that rely on the set-theoretical definition of the machine and guarded transition system semantics.

In particular, for a given machine m the predicate `Mch_INV`(m) holds if and only if the invariants of m hold with regard to m's behaviour, corresponding to PO INV (see Table 4.2). Following similar principles, every machine-related POs of the Event-B method is formalised in the theory.

```
Mch_INV_Init <predicate> (m : Machine(St, Ev))
  direct definition  AP(m) ⊆ Inv(m)
Mch_INV_One_Ev <predicate> (m : Machine(St, Ev), e : Ev)
  well−definedness  e ∈ Progress(m)
  direct definition  BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]] ⊆ Inv(m)
Mch_INV <predicate> (m : Machine(St, Ev))
  direct definition
      Mch_INV_Init(m) ∧ (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
...
```

Listing 4.3: Well-defined data type operators (behavioural semantics)

Finally, the PO operators are all gathered in a conjunctive expression within the `check_Machine_Consistency` operator (Listing 4.4), which thus encode the correctness condition for the machine. It uses `Machine_WellCons` as WD condition. At instantiation, it is used as a theorem to ensure machine correctness.

```
check_Machine_Consistency <predicate> (m : Machine(St, Ev))
  well−definedness  Machine_WellCons(m)
  direct definition  Mch_INV(m) ∧ ...
```

Listing 4.4: Operator for Event-B machine consistency

Remark. The EB4EB framework makes accessible all the features of Event-B machines, and thus enables the formalisation and verification of the fragment of temporal logic properties already supported by classical Event-B machines: $\Box$ using invariants and theorem clauses and $\Diamond$ using variants and convergence proof obligations. However, it does not support the composition of these operators nor any of the other temporal logic properties.

Instantiation of the meta-theory is used to define specific Event-B machines (instantiation) using the `Cons_machine` constructor. An Event-B context where values for the type parameters `St` and `Ev` are provided.

4.5 Trace-Based Semantics of Event-B

Establishing the *correctness* of the POs provided in the EB4EB framework requires modelling of Event-B trace-based semantics. We express traces in an Event-B theory and relate them to an EB4EB machine. It becomes possible to prove that a PO defined in EB4EB encodes correctly the property it formalises.

4.5.1 Semantics: traces of Event-B machines in EB4EB

A machine m consists of state variables and events describing their evolution. A trace tr of m is a sequence of states $tr = s_0 \mapsto s_1 \mapsto \ldots \mapsto s_n \mapsto \ldots$ such that:

1. the initial state s_0 satisfies the after predicate (AP) of the initialisation event
2. each pair of consecutive states s_i, s_{i+1} corresponds to the activation of an event e of m, i.e.: 1) s_i verifies the guard, and 2) $s_i \mapsto s_{i+1}$ verifies the BAP
3. if tr is *finite*, its final state deadlocks (i.e., system cannot progress any more)

In EB4EB, traces are encoded in a theory (Listing 4.5) extending `EvtBTheo`. They are linked to machines. A trace is a partial function $tr \in \mathbb{N} \nrightarrow St$ such that, for any n in the domain, $tr(n) = s_n$ is the n-th state of the trace.

```
THEORY EvtBTraces IMPORT EvtBTheo
TYPE PARAMETERS St , Ev
OPERATORS
  IsANextState predicate (m : Machine(St, Ev) , s : St , sp : St)
    direct definition ∃e · e ∈ Progress(m) ∧ s ∈ Grd(m)[{e}] ∧ s ↦ sp ∈ BAP(m)[{e}]
  IsATrace predicate (m : Machine(St, Ev) , tr : ℙ(ℕ × St))
    direct definition
      (tr ∈ ℕ → St ∨ (∃n · n ∈ ℕ ∧ tr ∈ 0..n → St ∧ tr(n) ∉ Grd(m)[Progress(m)]))∧
      tr(0) ∈ AP(m)∧
      (∀i, j · i ∈ dom(tr) ∧ j ∈ dom(tr) ∧ j = i + 1 ⇒ IsANextState(m, tr(i), tr(j)))
  ...
END
```

Listing 4.5: Theory of Event-B Traces

The operator `IsATrace` captures the relation between machines and traces. A transition associated to an event in a trace is defined by the `IsANextState` operator. Considering a machine m and two states s and sp, the operator checks that there exists an event e such that: 1) s verifies the guard of e ($s \in Grd(m)[\{e\}]$), and 2) the pair $s \mapsto sp$ verifies the BAP of e ($s \mapsto sp \in BAP(m)[\{e\}]$).

4.5.2 Correctness Principle

Soundness properties can be expressed with the formalisation of the semantics using traces, in particular the correctness of the newly defined POs [Riviere et al., 2022b]. A generic principle can be stated as follows.

In Listing 4.6, each PO `[PO]` is associated with a `thm_of_Correctness_of_[PO]` soundness theorem in the `Theo4[PO]Correctness` theory. It states that the `[PO]` predicate definition (see Section 4.7) implies the PO predicate definition expressed on traces using the `PO_Spec_On_Traces` expression. Such theorems have been proved for each PO introduced in the EB4EB framework.

```
THEORY Theo4[PO]Correctness IMPORT EvtBTraces , Theo4[PO]
TYPE PARAMETERS St, Ev
THEOREMS
  thm_of_Correctness_of_[PO] : ∀m, tr · m ∈ Machine(St, Ev) ∧ Machine_WellCons(m)∧
    IsATrace(tr, m) ∧ ... ∧ [PO](m, args) ⇒ PO_Spec_On_Traces(...)
```

Listing 4.6: Liveness Analyses Correctness

Example: Soundness of the Invariant PO (INV). The theorem of Listing 4.7 states that for any well-constructed machine m, if the invariant PO holds ($Mch\_INV(m)$) then for any trace tr associated to this machine ($IsATrace(tr, m)$), each state of that trace is in the invariant of the machine ($tr(i) \in Inv(m)$).

It has been proved, by induction on the indexes of the traces, using the Rodin platform provers. This principle is applied for all the newly introduced POs, in particular for the temporal logic properties POs introduced in this paper.

```
THEORY EvtBCorrectness IMPORT EvtBTraces, EvtBPO
TYPE PARAMETERS St, Ev
THEOREMS
  thm_of_Correctness_of_Invariant_PO: ∀m, tr · m ∈ Machine(St, Ev)∧
    Machine_WellCons(m) ∧ IsATrace(tr, m) ∧ Mch_INV(m)
      ⇒ (∀i · i ∈ dom(tr) ⇒ tr(i) ∈ Inv(m))
END
```

Listing 4.7: Theorem of correction of the proof obligation

This approach follows the work presented in [Aït Ameur et al., 2022]. It has been used in particular for hybrid systems as well [Dupont et al., 2021].

4.6 A Case Study: A read write machine

In the original paper [T. S. Hoang & Abrial, 2011], the authors used the read-write case study to illustrate their approach. For comparison purposes, we use the same case study.

```
MACHINE RdWrMch
VARIABLES r, w
INVARIANTS
  inv1−2: r ∈ ℕ, w ∈ ℕ
  inv3−4: 0 ≤ w − r, w − r ≤ 3
EVENTS
  INITIALISATION
  THEN
    act1: r, w := 0, 0
  END

  read
  WHERE grd1: r < w
  THEN act1: r := r + 1
  END

  write
  WHERE grd1: w < r + 3
  THEN act1: w := w + 1
  END

END
```

a

```
CONTEXT RdWr
SETS Ev
CONSTANTS rdwr, init, read, write
AXIOMS
  axm1: partition(Ev, {init}, {read}, {write})
  axm2: rdwr ∈ Machine(ℤ × ℤ, Ev)
  axm3: Event(rdwr) = Ev
  axm5: State(rdwr) = ℤ × ℤ
  axm6: Init(rdwr) = init
  axm7: Inv(rdwr) = {r ↦ w | r ∈ ℕ ∧ w ∈ ℕ∧
                    0 ≤ w − r ∧ w − r ≤ 3}
  axm8: AP(rdwr) = {0 ↦ 0}
  axm9: BAP(rdwr) = {e ↦ (
                (r ↦ w) ↦ (rp ↦ wp)) |
          (e = read ∧ rp = r + 1 ∧ wp = w)
        ∨(e = write ∧ rp = r ∧ wp = w + 1)}
  axm10: Grd(rdwr) = {e ↦ (r ↦ w) |
                (e = read ∧ r < w)∨
                (e = write ∧ w < r + 3)}
  axm11: Progress(rdwr) = {read, write}
  ...
  thm1: check_Machine_Consistency(rdwr)
END
```

b

Listing 4.8: Read write machine in Event-B (a) and instantiation with EB4EB (b)

The system requirements are: **Req1** – The reader process reads data from the buffer; **Req2** – The writer process writes data to the buffer; **Req3** – The reader

and the writer share the same buffer; **Req4** – The shared buffer has a fixed size of 3; **Req5** – The system does not stop when data is written and not read; and **Req6** – The reader eventually reads L, $L \in \mathbb{N}$, pieces of data.

Listing 4.8.a proposes the `RdWrMch` Event-B machine fulfilling the above requirements. The reader (resp. writer) is modelled by variable r (resp. w) corresponding to its position in the buffer and by event *read* (resp. *write*) that represents the associated input/output operation and increments the pointer (**Req1** and **Req2**). The shared buffer is captured by interval $r+1..w$ (**Req3**). The correct formalisation of the events, i.e. data that has not been written yet is not read and the amount of data in the buffer does not exceed 3 (**Req4**), is guaranteed by invariants *inv3-4*. Listing 4.8.b shows the context obtained when instantiating the `EvtBTheo` theory (Listing 4.1) of the EB4EB framework. The `thm1` theorem guarantees the consistency of the `RdWrMch` Event-B machine.

Missing requirements. Req5 and **Req6** are not *safety properties* in the usual sense and are not present in the current model. Event-B does not natively provide explicit constructs for handling them. Additional modelling effort is necessary, like introducing variants and new theorems and altering events.

4.7 Temporal logic proof rules as EB4EB POs

```
THEORY Theo4Liveness
IMPORT EvtBTheo
TYPE PARAMETERS Ev , St
...
```

Listing 4.9: Liveness operators Theory

To support temporal logic properties and handle the missing requirements, we propose an Event-B extension relying on the EB4EB framework. This section presents the formalisation of the liveness properties, introduced in [T. S. Hoang & Abrial, 2011], that are missing in core Event-B. For this purpose, we extend the EB4EB framework to introduce the corresponding PO definitions. All the definitions are formalised in the `Theo4Liveness` theory (see Listings 4.9) extending the `EvtBTheo` theory of EB4EB using a set of operators, defined for each proof rule defined in [T. S. Hoang & Abrial, 2011]. Each of these definitions is introduced below. Note that each of the following tables contain two parts, where (a) is from [T. S. Hoang & Abrial, 2011] and (b) our corresponding formalization.
Notations. For a predicate P on states of St, we define the subset $\hat{P}$ of states satisfying the property P as $\hat{P} = \{x \in St \mid P(x)\}$.

4.7.1 Liveness properties

This section presents core definitions for expressing formal definition of liveness properties. We first describe the basic building operators.

Machine M Leads From P_1 to P_2, $P_1 \curvearrowright P_2$ (`TLLeads_From_P1_To_P2` operator). For a machine M, given two state formulas P_1 and P_2, we state that M leads from P_1 to P_2 if for every trace of M with two successor states such that $s_i \in \hat{P}_1$ then $s_{i+1} \in \hat{P}_2$. The given property of Table 4.3(a) is formally defined by the operator `TLLeads_From_P1_To_P2` with a machine m and two set of states $\hat{P}_1$ and $\hat{P}_2$ as parameters. Its direct definition is a predicate $BAP(m)[\{e\}][\hat{P}_1 \cap Grd(m)[\{e\}] \cap$

$Inv(m)] \subseteq \hat{P}_2$ stating that for all progress events of machine m that preserve invariant, states of $\hat{P}_1$ lead to $\hat{P}_2$.

The Sequent Rule for $\curvearrowright$	Associated Operator in EB4EB
$P_1 \curvearrowright P_2 \equiv \forall v, v', x \cdot$ $P_1(v) \wedge G(x,v) \wedge A(x,v,v') \Rightarrow P_2(v')$	**TLLeads_From_P1_To_P2** <*predicate*> $(m : Machine(St, Ev),\ \hat{P}_1 : \mathbb{P}(St), \hat{P}_2 : \mathbb{P}(St))$ **direct definition** $\forall e \cdot e \in Progress(m) \Rightarrow$ $BAP(m)[\{e\}][\hat{P}_1 \cap Grd(m)[\{e\}] \cap Inv(m)] \subseteq \hat{P}_2$
(a)	(b)

Table 4.3: Leads from P1 to P2 encoded in EB4EB

Machine M is Convergent in P, $\downarrow P$ (`TLConvergent_In_P` operator). For a given property P, a machine M is convergent in P if it does not allow for an infinite sequence of P-states (i.e. states satisfying the property P). It is formalised in Table 4.4(a) by the predicate operator `TLConvergent_In_P` on machine m, set of states $\hat{P}$ and variant v. The operator's WD condition ensures that the variant is associated to each state. The operator states that, for all progress events e, when its before-after-states s and s' satisfy P, variant v decreases ($v(s') < v(s)$).

The Sequent Rule of $\downarrow$	Associated Operator in EB4EB
$\downarrow P \equiv \forall x, v, v' \cdot$ $(P(v) \wedge G(x,v) \Rightarrow V(v) \in \mathbb{N}) \wedge$ $(P(v) \wedge G(x,v) \wedge A(x,v,v')$ $\Rightarrow V(v') < V(v))$	**TLConvergent_In_P** <*predicate*> $(m : Machine(St, Ev),\ \hat{P} : \mathbb{P}(St), v : \mathbb{P}(St \times \mathbb{Z}))$ **well–definedness** $v \in St \rightarrow \mathbb{Z}$ **direct definition** $\forall e \cdot e \in Progress(m) \Rightarrow ($ $v[\hat{P} \cap Grd(m)[\{e\}] \cap Inv(m)] \subseteq \mathbb{N} \wedge$ $(\forall s, s' \cdot s \in Inv(m) \wedge s \in \hat{P} \wedge$ $s \in Grd(m)[\{e\}] \wedge s' \in BAP(m)[\{e\}][\{s\}]$ $\Rightarrow v(s') < v(s)))$
(a)	(b)

Table 4.4: Convergence in P encoded in EB4EB

Machine M is Divergent in P, $\nearrow P$ (`TLDivergent_In_P` operator). Divergence property guarantees that any infinite trace of a machine M ends with an infinite sequence of P-states. The operator `TLDivergent_In_P` of Table 4.5(a) is identical to the previous convergent operator, except that the variant does not decrease *strictly* ($v(s') \leq v(s)$) allowing divergent sequences of P-states.

The Sequent Rule of $\nearrow$	Associated Operator in EB4EB
$\nearrow P \equiv \forall x, v, v' \cdot$ $(\neg P(v) \wedge G(x,v) \Rightarrow V(v) \in \mathbb{N}) \wedge$ $(\neg P(v) \wedge G(x,v) \wedge A(x,v,v') \Rightarrow$ $V(v') < V(v)) \wedge$ $(P(v) \wedge G(x,v) \wedge A(x,v,v') \wedge V(v') \in \mathbb{N}$ $\Rightarrow V(v') \leq V(v))$	**TLDivergent_In_P** <*predicate*> $(m : Machine(St, Ev),\ \hat{P} : \mathbb{P}(St), v : \mathbb{P}(St \times \mathbb{Z}))$ **well–definedness** $v \in St \rightarrow \mathbb{Z}$ **direct definition** $TLConvergent\_In\_P(m, St \setminus \hat{P}, v) \wedge$ $\forall e \cdot e \in Progress(m) \Rightarrow ($ $(\forall s, s' \cdot s \in Inv(m) \wedge s \in \hat{P} \wedge s \in Grd(m)[\{e\}]$ $\wedge\ s' \in BAP(m)[\{e\}][\{s\}] \wedge v(s') \in \mathbb{N}$ $\Rightarrow v(s') \leq v(s)))$
(a)	(b)

Table 4.5: Divergence in P encoded in EB4EB

Machine M is Deadlock-free in P, ⟳ P (`TLDeadlock_Free_In_P` operator). The deadlock-freeness states that a trace of a machine M never reaches a P-state where no event is enabled. It requires that, in a P-state, at least one event of M is enabled. This property is defined in Table 4.6(a) and is formalised by the operator `TLDeadlock_Free_In_P` in Table 4.6(b).

The expression $\hat{P} \cap Inv(m) \subseteq Grd(m)[\ Progress(m)]$ ensures that at least one progress event of the $Progress(m)$ set is enabled in a P-state satisfying the invariant.

The Sequent Rule of ⟳	**Associated Operator in EB4EB**
$\circlearrowleft\ P \equiv \forall v \cdot P(v) \Rightarrow \bigvee_i (\exists x \cdot G_i(x, v))$	**TLDeadlock_Free_In_P** <*predicate*> $(m : Machine(St, Ev), \hat{P} : \mathbb{P}(St))$ **direct definition** $\hat{P} \cap Inv(m) \subseteq Grd(m)[Progress(m)]$
(a)	(b)

Table 4.6: Deadlock-freeness in P encoded in EB4EB

4.7.2 *Deadlock freeness* ⟳ P applied to the Read-Write machine

We illustrate how the operators defined above work in the extended EB4EB framework on the read write case study, with the case of the deadlock-freeness property ensuring requirement **Req5**.

A context `RdWrDeadlockFree`, extending the context `RdWr` of Listing 4.8 is defined with a theorem, `thmDeadlockFreeInP`. This theorem uses the predicate operator `Deadlock_Free_In_P`, previously formalised. Here, the $\hat{P}$ parameter is composed of the pair of state variables $r \mapsto w$ and the property P defined by $w \in \mathbb{Z} \wedge r \in \mathbb{Z} \wedge r < w$. Indeed, the machine does not deadlock if it reads less data than it writes. Remember that when a theorem is stated, a PO is automatically generated requiring to prove it.

```
CONTEXT RdWrDeadlockFree
EXTENDS RdWr
THEOREMS
  thmDeadlockFreeInP :
    TLDeadlock_Free_In_P(rdwr ,
      {r ↦ w | w ∈ ℤ ∧ r ∈ ℤ ∧ r < w})
END
```

Listing 4.10: Generation of Proof Obligation of Deadlock_Free_In_ P

4.7.3 Temporal operator proof rules

Section 4.7.1 presents a formalisation of the basic temporal operators allowing to define liveness properties. This section is devoted to the formalisation of more complex temporal properties, relying on the operators previously defined, like `TLGlobally`, `TLExistence` `TLUntil`, `TLProgress`, and `TLPersistence`. Each of them is defined in the same manner as the previous ones.

Invariance, □I (`TLGlobally` operator). In Event-B, safety properties are commonly described as invariants. Although this property is already available in core Event-B, it can be formalised in EB4EB as well.

Table 4.7(a) expresses this property using two sequents. The first one is the inductive invariant proof rule and the second one defines, as theorems, all of the entailed stronger invariants. The `TLGlobally` operator of Table 4.7(b) defines this property as $Inv(m) \subseteq \hat{I}$; it reuses the native invariant PO of EB4EB.

The Sequent Rule of $\Box$	**Associated Operator in EB4EB**
$\dfrac{\vdash\ init \Rightarrow I \qquad M\ \vdash\ I \curvearrowright I}{M\ \vdash\ \Box I}$ $\dfrac{\vdash\ J \Rightarrow I \qquad M\ \vdash\ \Box J}{M\ \vdash\ \Box I}$	**TLGlobally** <*predicate*> $(m : Machine(St, Ev),\ \hat{I} : \mathbb{P}(St))$ **direct definition** $Inv(m) \subseteq \hat{I}$
(a)	(b)

Table 4.7: Invariance encoded in EB4EB

Existence, $\Box\Diamond P$ *(`TLExistence` operator).* The existence temporal property states that a property P *always eventually* holds for machine M. To express existence $\Box\Diamond P$ in a machine M, we rely on convergence and deadlock-freeness. Indeed, the machine shall be convergent on $\neg P$-states, i.e., sometimes $\neg P$ does not hold and $\neg P$-states are not deadlocks. The defined `TLExistence` predicate operator is defined as the conjunction of the two corresponding previously defined operators on a set $\hat{P}$ and a variant v.

The Sequent Rule of $\Box\Diamond$	**Associated Operator in EB4EB**
$\dfrac{M\ \vdash \downarrow \neg P \qquad M\ \vdash \circlearrowleft \neg P}{M\ \vdash\ \Box\Diamond P}$	**TLExistence** <*predicate*> $(m : Machine(St, Ev),\ \hat{P} : \mathbb{P}(St), v : \mathbb{P}(St \times \mathbb{Z})$ **well−definedness** $v \in St \rightarrow \mathbb{Z}$ **direct definition** $TLConvergent\_In\_P(m, St \setminus \hat{P}, v) \wedge$ $TLDeadlock\_Free\_In\_P(m, St \setminus \hat{P})$
(a)	(b)

Table 4.8: Existence encoded in EB4EB

Until, $\Box(P_1 \Rightarrow (P_1 \mathcal{U} P_2))$ *(`TLUntil` operator).* The *Until* property states that a P_1-state is *always followed eventually* by a P_2-state. Its definition relies on the *leads-to* and *existence* properties we have introduced. The *Until* property requires two antecedents, a leads to from $P_1 \wedge \neg P_2$ to $P_1 \vee P_2$ in the next state and the second is the existence of $\neg P_1 \vee P_2$ (see Table 4.9(a)). This proof rule is directly formalises using the `TLUntil` operator (see Table 4.9(b)). It requires two properties P_1 ($\hat{P}_1$ set) and P_2 ($\hat{P}_2$ set) and a variant v. It is defined as the conjunction of the `TLLeads_From_P1_To_P2` and `TLExistence` predicate operators.

The Sequent Rule of $\Box(P_1 \Rightarrow (P_1 \mathcal{U} P_2)$	**Associated Operator in EB4EB**
$A \equiv (P_1 \wedge \neg P_2) \curvearrowright (P_1 \vee P_2)$ $B \equiv \Box\Diamond(\neg P_1 \vee P_2)$ $\dfrac{M\ \vdash\ A \qquad M\ \vdash\ B}{M\ \vdash\ \Box(P_1 \Rightarrow (P_1 \mathcal{U} P_2))}$	**TLUntil**<*predicate*> $(m : Machine(St, Ev),$ $\hat{P}_1 : \mathbb{P}(St), \hat{P}_2 : \mathbb{P}(St),\ v : \mathbb{P}(St \times \mathbb{Z})$ **well−definedness** $v \in St \rightarrow \mathbb{Z}$ **direct definition** $Leads\_From\_P1\_To\_P2($ $m, \hat{P}_1 \cap (St \setminus \hat{P}_2), \hat{P}_1 \cup \hat{P}_2)$ $\wedge\ TLExistence(m, (St \setminus \hat{P}_1) \cup \hat{P}_2, v)$
(a)	(b)

Table 4.9: Until encoded in EB4EB

Progress, $\Box(P_1 \Rightarrow (\Diamond P_2))$ (*`TLProgress` operator).* Close to the *Until* property, a more general property, namely *Progress* can be defined. It states that always P_1-states reaches $P2$-states. This property does not require P_1 to always hold before reaching P_2-states. To describe this property, an intermediate property P_3 holding before P_2 holds is introduced. It acts as a local invariant between P_1-states and P_2-states.

The Sequent Rule of $\Box(P_1 \Rightarrow \Diamond P_2)$	**Associated Operator in EB4EB**
$A \equiv \Box(P_1 \wedge \neg P_2 \Rightarrow P_3)$ $B \equiv \Box(P_3 \Rightarrow (P_3 \mathcal{U} P_2))$ $\dfrac{M \vdash A \qquad M \vdash B}{M \vdash \Box(P_1 \Rightarrow (\Diamond P_2))}$	**TLProgress**<*predicate*> $(m : Machine(St, Ev),$ $\hat{P_1} : \mathbb{P}(St), \hat{P_2} : \mathbb{P}(St), \hat{P_3} : \mathbb{P}(St), v : \mathbb{P}(St \times \mathbb{Z})$ **well−definedness** $v \in St \rightarrow \mathbb{Z}$ **direct definition** $TLGlobally(m, \hat{P_3} \cup \hat{P_2} \cup (St \setminus \hat{P_1})) \wedge$ $TLUntil(m, variant, \hat{P_3}, \hat{P_2})$
(a)	(b)

Table 4.10: Progress encoded in EB4EB

The *Progress* proof rule of Table 4.10(a) has two antecedents. One states that always $P_1 \wedge \neg P_2 \Rightarrow P_3$ and the second uses the previously defined *Until* property as $\Box(P_3 \Rightarrow (P_3 \mathcal{U} P_2))$. The `TLProgress` predicate operator is the conjunction of the application of the two predicate operators, `Leads_From_P1_To_P2` and `TLUntil` on the $\hat{P_1}$, $\hat{P_2}$ and $\hat{P_3}$ sets and the variant v, encoding the antecedents.

Persistence, $\Diamond\Box P$ (*`TLPersistence` operator). Persistence* is the last property we formalise. It states that a predicate P must eventually hold forever ($\Diamond\Box P$). The two antecedents of the associated proof rule, presented in Table 4.11(a), state that P-states are divergent $\neg P$-states are deadlock-free. The `TLPersistence` predicate operator is defined as a conjunctive expression of `TLDivergent_In_P` and `TLDeadlock_Free_In_P` operators with the $\hat{P}$ for the property P and the variant v as input parameters.

The Sequent Rule of $\Diamond\Box$	**Associated Operator in EB4EB**
$\dfrac{M \vdash \nearrow P \qquad M \vdash \circlearrowleft \neg P}{M \vdash \Diamond\Box P}$	**TLPersistence** <*predicate*> $(m : Machine(St, Ev), \hat{P} : \mathbb{P}(St), v : \mathbb{P}(St \times \mathbb{Z})$ **well−definedness** $v \in St \rightarrow \mathbb{Z}$ **direct definition** $TLDivergent\_In\_P(m, \hat{P}, variant) \wedge$ $TLDeadlock\_Free\_In\_P(m, St \setminus \hat{P})$
(a)	(b)

Table 4.11: Persistence encoded in EB4EB

4.7.4 *Existence* $\Box\Diamond P$ applied to the read write machine

```
CONTEXT RdWrExistence
EXTENDS RdWrDeadlockFree
CONSTANTS L
AXIOMS
  axm1: L ∈ ℕ
  thmExistence: TLExistence(rdwr, {r ↦ w | w ∈ ℤ ∧ r ≥ L},
      {(r ↦ w) ↦ v | v = ((L − r) + (L + 3 − w))})
END
```

Listing 4.11: Generation of Proof Obligation of Existence

The temporal operators defined in [T. S. Hoang & Abrial, 2011] have been successfully formalised in the EB4EB as predicate operators used as theorems to be proved for any Event-B machine.

Here, we show how **Req6** (the reader eventually reads L, $L \in \mathbb{N}$, pieces of data) expressed for the read write case study is fulfilled thanks to the `TLExistence` operator. Like for deadlock freeness in section 4.7.2, we introduce a new Event-B context `RdWrExistence` (see Listing 4.11), extending the `RdWr` context of Listing 4.8, with a theorem stating the existence property. The existence operator is used with a set of states $\{r \mapsto w \mid w \in \mathbb{Z} \wedge r \geq L\}$ and a variant $v = ((L-r)+(L+3-w))\}$).

4.8 Correctness of the temporal logic properties proof rules

The last step establishes the correctness of our formalisation with respect to the semantics of trace, i.e. the defined proof rules actually hold on the traces of the Event-B machines. The verification principle of Section 4.5.2 is set up for this purpose. A theory `Theo4LivenessCorrectness` (Listing 4.12) provides a list of correctness theorems for each of the defined operators. It imports the previously developed theories related to liveness properties `Theo4Liveness` and Event-B traces `EvtBTraces`.

```
THEORY Theo4LivenessCorrectness
  IMPORT Theo4Liveness, EvtBTraces
  TYPE PARAMETERS St, Ev
  ...
```

Listing 4.12: Theory of correctness

Below, we present the correctness theorem for the `TLExistence` property. All the other theorems are formalised[2] and proved using the Rodin Platform.

Existence in P correctness theorem $\Box\Diamond P$ (`TLExistence`). The correctness of the existence property follows the principle of Section 4.5.2. It is supported by the proved `thm_of_correctness_of_Existence` theorem stating that a property P always eventually holds in traces of a machine m. It states that for any well constructed ($Machine\_WellCons(m)$) and consistent ($check\_Machine\_Consistency(m)$) machine, and for any trace tr of this machine satisfying the existence property $TLExistence(m, \hat{P}, variant)$, then for all i there exists j with $j \geq i$ where $tr(j)$ satisfies the property P.

```
THEOREMS
  thm_of_Correctness_of_Existence: ∀m, tr, v, P̂ · v ∈ STATE → ℤ∧
    m ∈ Machine(STATE, EVENT) ∧ Machine_WellCons(m)∧
    check_Machine_Consistency(m) ∧ IsATrace(m, tr) ∧ TLExistence(m, P̂, v)
        ⇒ (∀i · i ∈ dom(tr) ⇒ (∃j · j ≥ i ∧ j ∈ dom(tr) ∧ tr(j) ∈ P̂))
  ...
```

Listing 4.13: Theorem of correctness of the operators Existence

[2] https://www.irit.fr/~Peter.Riviere/models/

4.9 Related Work

Reflexive modelling is present under various forms in formal methods. For instance, the *ASM-Metamodel* API (AsmM) for Abstract State Machines (ASM) has been developed to be able to handle ASM-related concepts. This leads to several extensions, analyses and tools for ASMs [Riccobene & Scandurra, 2004]. This is also the case when using Mural to modify a VDM specification [Bicarregui & Ritchie, 1991]. Furthermore, the reflexive modelling is also addressed with proof assistants like Coq with MetaCoq [Sozeau et al., 2020], Agda [Paul van der Walt, 2012], PVS [Mitra & Archer, 2004], HOL [Fallenstein & Kumar, 2015] and Lean [Ebner et al., 2017] and Event-B with EB4EB [Riviere et al., 2022a, 2022b].

Correctness of the Event-B method and its modelling components has been tackled in various previous work. A meta-level study of Event-B context structure is proposed in particular to validate the expected properties of theorem instantiation [Bodeveix & Filali, 2021]. Event-B has also been formalised as an *institution* in category theory [Farrell et al., 2016, 2022], with the aim to facilitate and enable composition of heterogeneous semantics and of different model specifications. Similarly, Event-B has been embedded in Coq [Castéran, 2021] in order to establish the correctness of refinement, i.e. that the refinement POs entail the validity of refinement in the trace-based semantics. Last, a form of shallow embedding of Event-B in itself has been proposed and serves as the basis of a methodology for proving the correctness of decomposition and re-composition of Event-B machines [Hallerstede & Hoang, 2014].

Event-B's methodology is mainly aimed at defining and proving safety properties (that must always hold), or possible convergence. Expressing liveness properties (that must hold at some point [Lamport, 1977]) is not as trivial, and many authors address this issue. For Event-B, the ProB model-checker [Leuschel & Butler, 2008] handles Event-B models and enables the expression and verification of liveness properties. Some liveness operators have been formalised to be used in Event-B, together with their related hypotheses [T. S. Hoang & Abrial, 2011], making it possible to express *some* liveness properties. However, it is to be noted that liveness properties are not generally preserved by refinement. To address this latter issue, additional conditions on the refinement must be posed, leading to the definition of particular refinement strategies [T. S. Hoang et al., 2016], which are proven to preserve liveness properties through to the concrete model. In addition, the problem of *fairness* has also been studied. For instance, the work of [Méry & Poppleton, 2017] proposes to check fairness of Event-B machines in TLA (on a per-machine basis). Refinement strategies have been defined as well to ensure that fairness and liveness properties are preserved [Zhu et al., 2023].

Our proposed approach is based on the reflexive modelling of Event-B on itself, which is fully integrated into Rodin development environment using the Theory Plugin [M. J. Butler & Maamria, 2013]. Our framework is fully formalised in Event-B and relies solely on FOL and set theory, similar to other approach like MetaCoq [Sozeau et al., 2020] with dependent type. Such characteristic makes it possible to *export* models expressed using the framework to any other formalism based on FOL and set theory while preserving the state-transition semantics of the

model. Therefore, the issue of the translation of the universe and the semantics' preservation are not related to our work due to the reflexive modelling.

4.10 Conclusion

This paper has presented a formalisation of liveness properties for Event-B models by encoding LTL temporal logic expressions on the Rodin platform using the reflexive EB4EB framework. LTL logic expressions of properties are formalised within the defined framework. Automatic generation of proof obligations related to the expressed properties and the soundness of the defined proof rules using a trace based semantics have been addressed as well. The proposed approach relies on the definition of algebraic theories offering the capability to define new operators. The read write machine case study was borrowed from [T. S. Hoang & Abrial, 2011] to illustrate our approach. Other case studies have been developed as well (Peterson algorithm [Riviere et al., 2023c] and behavioural analyses in human computer interaction [Mendil et al., 2022]).

The proposed framework supports non-intrusive analysis for Event-B models, allowing liveness properties to be expressed and verified on any size Event-B formal model and at any refinement level without resorting to any other formal methods. Since our framework allows checking temporal properties at any refinement level, it avoids dealing with the preservation of temporal properties by refinement. Furthermore, the proof process has been enhanced with relevant and proven rewrite rules, which have been incorporated into Rodin tactics, resulting in a high level of proof automation. All the developments illustrated in this paper have been fully formalised and proved using the Rodin platform. They can be accessed on https://www.irit.fr/~Peter.Riviere/models/

This work leads to several perspectives. First, we plan to study the capability to allow compositional definitions of LTL properties relying on the defined basic operators. In addition, the proposed approach makes it possible to define other Event-B model analyses or domain specific theories shared by many Event-B models. Last, we believe that our approach can be scaled up to other state based methods provided that a reflexive meta-model is available.

Assessment

This chapter presents a major result: the EB4EB framework is capable of expressing temporal properties (such as *liveness* properties) on models, and then proving said properties formally. Such properties are expressed using fragments of linear temporal logic, encoded algebraically, directly in the framework, and proven to be sound relative to trace-based semantics. The algebraic definitions are based on the work of Hoang and Abrial.

The primary benefit of this extension to the EB4EB framework is the inclusion of temporal reasoning capabilities within the proof method for infinite state spaces. Currently, such analyses can only be performed on models with a reduced number of states, using model checking tools like ProB. Furthermore, the theory of temporal operators can be enhanced by defining more complex operators to cover other temporal properties, like CTL, as well as to address the notions of completeness for expressing any temporal properties. The proposed mechanism for temporal properties' extension of the EB4EB framework can be adapted for designing various reasoning extensions of different kinds of temporal logic, such as MTL (Metric Temporal Logic). Furthermore, the proposed framework can be extended to include other advanced reasoning notions such as probability, parallel computing, and quantum commuting.

Similar to our previous development, there is a lack of proof automation, but by providing the additional proof tactics the proof burden is reduced significantly. However, there is still room to improve the automation steps.

Chapter 5

Extending Event-B with Explicit Model Annotations

Overview

This chapter presents a methodology for handling domain knowledge requirements and modelling system behaviour in order to perform domain specific behaviour analysis by extending the EB4EB framework and its accompanying theories. This work was carried out in collaboration with PhD student Ismail Mendil, who developed the domain theories based on ontologies as well as a system model based on said theories. Further, we applied the EB4EB framework for analysing domain specific behaviour properties represented using the algebraically defined temporal operators. To demonstrate our approach, we used an Automated Teller Machine (ATM) case study. The *Theory plugin* and Rodin IDE were used to develop domain theories and system models, as well as the domain specific behaviour analysis performed using the EB4EB framework and its associated proof tactics.

Associated paper of this chapter:

- Mendil, I., Riviere, P., Aït Ameur, Y., Singh, N. K., Méry, D., & Palanque, P. A. [2022]. Non-intrusive annotation-based domain-specific analysis to certify event-b models behaviours. *29th Asia-Pacific Software Engineering Conference, APSEC*, 129–138

Non-Intrusive Annotation-Based Domain-Specific Analysis to Certify Event-B Models Behaviours

I. Mendil[1], P. Rivière[1], Y. Ait-Ameur[1], N. K. Singh[1], D. Méry[2], P. Palanque[3]

[1] INPT-ENSEEIHT / IRIT Universié de Toulouse, France
[2] Telecom Nancy, LORIA Université de Lorraine, France
[3] IRIT Université de Toulouse, France

5.1 Introduction

5.1.1 Context

System behaviour analysis necessitates handling domain constraints, knowledge and standards together with the use of different logics and in particular temporal logic which allows for expressing requirements related to the system behaviour. Event-B [J.-R. Abrial, 2010], like other formal methods, offers built-in mechanisms like invariant preservation for verifying properties of complex systems models expressed using abstract machines. However, in order to formalise complex properties, in particular behavioural properties, the designer must accommodate the Event-B modelling language constructs especially since these constructs are based on first-order logic and set theory.

While safety properties are explicitly modelled in Event-B thanks to invariants and theorems, the temporal properties related to liveness require a complex operational formalisation. Moreover, handling constraints raised by standards or domain knowledge properties require ad hoc modelling by the designer.

In article [T. S. Hoang & Abrial, 2011], the authors extend the class of liveness properties for Event-B by defining a list of proof obligations used to express proof rules. A behavioural semantics to Event-B and a collection of conditions allowing for verifying LTL properties across a refinement chain are proposed in [S. Hoang et al., 2016; Schneider et al., 2014]. Although the approach supports extended LTL operators, the paper does not address the issue of explicitly handling domain knowledge. Several modelling frameworks, including DOL, CASL [Mossakowski, 2016] and RAISE [Bjørner, 2006, 2017, 2019], advocate for explicit domain knowledge in formal modelling where several fields, such as railways systems, shipping, and logistics, are described. Additionally, the authors of [Aït Ameur & Méry, 2016] highlight the benefits of expressing explicitly domain properties, and a collection of applications that use explicit domain knowledge in modelling is presented in [Aït Ameur et al., 2021]. A certification process that ensures a system model meets the requirements of a standard formalised as an ontology discussed in [Mendil, Ameur, et al., 2021]. This approach is constructive and relies on the annotation of state variables with ontology concepts and a set of operators used to transfer domain knowledge formalised as properties to models.

To the best of our knowledge, no previous work has addressed the issue of deal-

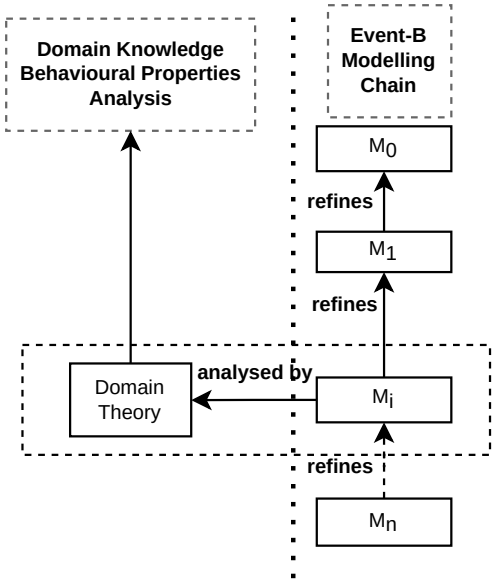


Figure 5.1: Methodology overview

ing with domain-knowledge behavioral properties in formal design model analysis. The methodology described in this article claims novelty in this regard, and the proposed approach goes beyond the intrusive approach in dealing with behavioral analyses, which integrate elements of the analysis within formal design models such as auxiliary variables or events. By contrast the proposed approach employs a non-intrusive technique that is accomplished through external annotation of events using the EB4EB framework, which allows us to lift the Event-B model by providing handles for manipulating and referencing the model's elements. Moreover the methodology shed light on the importance and relevance of the contextual information related to the system under study.

This paper describes behavioural analyses mined from domain knowledge. Moreover, the analyses discussed here do not require an *a priori* alteration of the models (non-intrusive approach), but rather the certification of the behavioural model is established through annotations.

5.1.2 Objective of this paper

The prime objective of this paper is to provide an integrated framework (see Fig. 5.1) for verifying domain-specific behavioural properties of formal design models. The framework meets an important requirement: non-intrusiveness, achieved by annotation i.e. the model's elements are associated with domain knowledge concepts. This high-level goal is divided into subgoals as follows:

1. Supply a language for expressing domain knowledge concepts and rules.

Ontologies are good candidates for this purpose.

2. Lift formal models to a meta-level to manipulate explicitly their constituents.

3. Provide a mechanism for defining analyses combining both domain knowledge and formal modelling concepts

4. Bind system models concepts and domain knowledge concepts and set up the *domain-specific behavioural analysis*.

Note that the subgoals (1) and (2) are studied and described in [Mendil, Aït Ameur, et al., 2021] and [Riviere et al., 2022a], respectively. The novelty of this article is the proposition of an integrated framework for handling behavioral requirements issued from standards through formal models analyses. This framework takes into account explicit domain knowledge of formal design models. Moreover, the proposed methodology is based on non-intrusive reasoning mechanism i.e. it does not require updating the system model.

5.1.3 Organisation of this paper

This paper is structured as follows. Section 5.2 overviews the Event-B method which is the backbone of the proposed framework. Section 5.3 recalls the two formal languages used to address sub-goals (1) and (2) respectively, where the ontology modelling and the meta-Event-B languages are introduced. Section 5.4 presents the integrated framework, meeting sub-goals (3) and (4), to address formal model analyses and annotation mechanisms. Section 5 describes the case study illustrating the proposed approach and Section 6 defines a specific domain knowledge-based analysis according to the methodology devised in this article. Sections 7 provides an assessment and Section 8 concludes with future perspectives.

5.2 Event-B method

Event-B [J.-R. Abrial, 2010] is a *correct-by-construction* method based on set theory and first-order logic. It supports state-based modelling where a set of events encodes state changes. Proof Obligations (PO) (see Table 5.2) are automatically generated.

5.2.1 Contexts and machines (see Table 5.1.(b) and 5.1.(c))

A `Context` describes the static properties of a model: Axioms and theorems describing required concepts using *carrier sets* s, *constants* c, *axioms* A and *theorems* T_{ctx}. A `Machine` describes the model behaviour as a transition system. A set of guarded events is used to modify the state using Before-After Predicates.

Theory	Context	Machine
THEORY Th **IMPORT** Th1, ... **TYPE PARAMETERS** E, F, ... **DATATYPES** **Type1**(E, ...) **constructors** **cstr1**(p_1: T_1, ...) **OPERATORS** **Op1** <nature> (p_1: T_1, ...) **well—definedness** $WD(p_1, ...)$ **direct definition** D_1 **AXIOMATIC DEFINITIONS** **TYPES** A_1, ... **OPERATORS** **AOp2** <nature> (p_1: T_1, ...): T_r **well—definedness** $WD(p_1, ...)$ **AXIOMS** A_1, ... **THEOREMS** T_1, ... **PROOF RULES** R_1, ... **END**	**CONTEXT** Ctx **SETS** s **CONSTANTS** c **AXIOMS** A **THEOREMS** T_{ctx} **END**	**MACHINE** M **SEES** Ctx **VARIABLES** x **INVARIANTS** $I(x)$ **THEOREMS** $T_{mch}(x)$ **VARIANT** $V(x)$ **EVENTS** **EVENT** evt **ANY** α **WHERE** $G_i(x, \alpha)$ **THEN** $x :\mid BAP(\alpha, x, x')$ **END** ... **END**
(a)	(b)	(c)

Table 5.1: Global structure of Event-B Theories, Contexts and Machines

(1)	Ctx Theorems (ThmCtx)	$A(s,c) \Rightarrow T_{ctx}$ `(For contexts)`
(2)	Mch Theorems (ThmMch)	$A(s,c) \wedge I(x)$ $\Rightarrow T_{mch}(x)$ `(For machines)`
(3)	Initialisation (Init)	$A(s,c) \wedge G(\alpha) \wedge BAP(\alpha, x')$ $\Rightarrow I(x')$
(4)	Invariant preservation (Inv)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge$ $BAP(x,\alpha,x') \Rightarrow I(x')$
(5)	Variant progress (Var)	$A(s,c) \wedge I(x) \wedge G(x,\alpha) \wedge$ $BAP(x,\alpha,x') \Rightarrow \ V(x') < V(x)$

Table 5.2: Relevant Proof Obligations for Event-B contexts and machines

Refinements. Refinement decomposes a *machine* into a less abstract one with more design decisions moving from an abstract level to a less abstract one. Gluing invariants relating abstract and concrete variables ensure property preservation.

Core Well-definedness (WD). WD POs are associated with all Event-B operators. Once proved, these WD conditions are used as hypotheses to prove other POs.

5.2.2 Event-B extensions with Theories (see Table 5.1.(a))

To handle more complex and abstract concepts beyond set theory and first-order logic, an Event-B extension for externally defined mathematical objects has been proposed in [J.-R. Abrial et al., 2009; M. J. Butler & Maamria, 2013]. It introduces user data types, operators, theorems and associated rewrite and inference rules, all bundled in so-called *theories* similar to other proof assistants like Coq [Bertot & Castéran, 2010], Isabelle/HOL [Nipkow et al., 2002] or PVS [Owre et al., 1992].

Theories Definition. Theories contain *datatypes* and *operators* that can be used in Event-B expressions as *predicates* or *expressions* producing values. Operators may be defined explicitly in the `OPERATORS` clause, or defined axiomatically in the `AXIOMATIC DEFINITIONS` clause. Last, a theory may provide theorems and proof

rules. Many theories have been defined for lists, reals, differential equations, etc.

Well-definedness (WD) in Theories. An important feature provided by Event-B theories is the possibility to define *Well-Definedness* (WD) conditions (close to TCC conditions in PVS [Owre et al., 1992]). Each defined operator (thus partially defined) is associated with a user-defined condition ensuring its correct definition. When it is applied, this WD condition generates a PO that needs to be discharged.

Event-B proof system and Rodin. Rodin[1] is an open source IDE for modelling in Event-B. It provides model editing, automatic PO generation, project management, refinement and proof, model checking, model animation and code generation. The Event-B theories extension is available as a plug-in. They tightly integrated in the proof process. Depending on their definition, operator definitions are expanded either using their direct definition or by enriching the hypotheses pool using their axiomatic definition. Theorems may be imported as hypotheses. Many provers for first-order logic as well as SMT solvers are plugged into Rodin.

5.3 Background and Related Work

This section reviews the previous work which is the foundation of the methodology presented in this article. The subsections 5.3.1 and 5.3.2 recall important elements of two languages formalised as Event-B theories; (1) ontology modelling language allowing the description of the domain knowledge, and (2) the meta-Event-B allowing a direct manipulation of the Event-B concepts.

Moreover, this section also review the role and the techniques used across different disciplines notably in formal methods for specifying and referencing domain knowledge.

5.3.1 Ontology Modelling Language as Event-B Theory

The ontology modelling language [Mendil, Aït Ameur, et al., 2021] is used for describing domain-specific behavioural properties. It features a trade-off between the expressive power of first-order logic and the practicality of high-level primitives. It comes as an Event-B theory providing one data type and a collection of operators and theorems.

Listing 5.1 shows important elements used for modelling an ontology. `OntologiesTheory` is an Event-B theory which is parameterised by `C`, `P` and `I` denoting Classes, Properties and Instances, respectively. This theory defines a constructor `consOntology` with 7 attributes: `classes`, `properties`, `instances`, `classProperties`, `classInstances`, `classAssociations`, `instanceAssociations`.

In addition, expression and predicate operators allowing to manipulate the ontology are defined. Operators returning an expression allows computing values based on ontology attributes. Predicate operators are used defining well-

[1]Rodin Integrated Development Environment http://www.event-b.org/index.html

definedness conditions and to check logical properties. For example, `getClassInstances` is an operator that allows retrieve the relation class to instances in a *safe way* since the operator is correctly used only if its well-defined condition is discharged. The well-definedness condition of this operator is formalised in the `isWDClassInstances` stating that the classes and instances of the relation correspond respectively to classes and instances of the ontology. Another important operator is `isWDOntology` which checks that an ontology is well defined in the sense that all the attributes obey the individual well-definedness condition. Other operators are provided among them `isA` and `ontologyContainsClasses` where the former checks where a class subsumes another class and the latter verifies whether a collection of classes belongs to a given ontology.

THEORY $OntologiesTheory$
TYPE PARAMETERS C, P, I
DATA TYPES
$\quad Ontology(C, P, I)$
CONSTRUCTORS
$\quad consOntology(\boldsymbol{classes} : \mathbb{P}(C), \boldsymbol{properties} : \mathbb{P}(P), \boldsymbol{instances} : \mathbb{P}(I),$
$\quad\quad \boldsymbol{classProperties} : \mathbb{P}(C \times P), \boldsymbol{classInstances} : \mathbb{P}(C \times I),$
$\quad\quad \boldsymbol{classAssociations} : \mathbb{P}(C \times P \times C),$
$\quad\quad \boldsymbol{instanceAssociations} : \mathbb{P}(I \times P \times I))$
OPERATORS
isWDClassInstances $<predicate>$ ...
getClassInstances $<expression>$ ...
isWDOntology $<predicate>$ $(o : Ontology(C, P, I))$
$\quad$ **direct definition**
$\quad\quad isWDClassProperites(o) \land isWDClassInstances(o) \land$
$\quad\quad isWDClassAssociations(o) \land isWDInstancesAssociations(o)$
ontologyContainsClasses $<predicate>$ ...
isA $<predicate>$ $(o : Ontology(C, P, I), c1 : C, c2 : C)$
$\quad$ **well−definedness** $isWDOntology(o), ontologyContainsClasses(o, c1, c2)$
$\quad$ **direct definition**
$\quad\quad getInstancesOfaClass(o, c1) \subseteq getInstancesOfaClass(o, c2)$
...
THEOREMS
isATrans:
$\quad \forall o, c1, c2, c3 \cdot o \in Ontology(C, P, I) \land isWDOntology(o) \land$
$\quad\quad c1 \in C \land c2 \in C \land c3 \in C \land$
$\quad\quad ontologyContainsClasses(o, c1, c2, c3)$
$\quad\quad\quad \Rightarrow (isA(o, c1, c2) \land isA(o, c2, c3) \Rightarrow isA(o, c1, c3))$

Listing 5.1: Ontology modelling language Event-B theory

Last, theorems may be derived for the ontology modelling language from the data type operator definitions. For example, the theorem `isATran` states that `isA` is transitive. It is noteworthy that the theorems are valid provided that the ontology is well defined, the same applies to all the operators which require a well-defined ontology as an argument.

5.3.2 The Event-B Meta-theory

For enhancing the reasoning support of Event-B, a reflexive framework has been defined in the article [Riviere et al., 2022a]. To access Event-B components as first-class elements and keep the semantics, and propose a reasoning mechanism expressed with the meta-level.

THEORY $EvtBTheo$
TYPE PARAMETERS St, Ev

```
DATATYPES Machine(St, Ev)
 CONSTRUCTORS
   Cons_machine(Event : ℙ(Ev), State : ℙ(St), Init : Ev,
     Progress : ℙ(Ev), Variant : ℙ(St × ℤ), AP : ℙ(St),
     BAP : ℙ(Ev × (St × St)), Grd : ℙ(Ev × St), Inv : ℙ(St), ...)
```

Listing 5.2: Machine Data type

Listing 5.2 shows the data type representing the machine's elements, which are parameterised by two types: `Ev` and `St`. A constructor is defined `Cons_machine` where each argument corresponds to a machine component. The machine denotes a state transition system on the set of states (`State`) constrained by the invariant (`Inv`).

Machine structure Events are triggered by the initialisation event (`Init`) then by progress events (`Progress`). State changes are provoked by the After predicate (`AP`) for `Init`, and the Before After Predicate (`BAP`) for progress events if their corresponding guards (`Grd`) are true. A numeric variant (`Variant`) is defined for liveness properties.

```
Event_WellCons <predicate> (m : Machine(St, Ev))
  direct definition partition(Event(m), {Init(m)}, Progress(m))
  ...
Machine_WellCons <predicate> (m : Machine(St, Ev))
  direct definition Event_WellCons(m) ∧ ...
```

Listing 5.3: Operators to check well-defined data type (static semantics)

Well-Constructed machines The data type requires to formalise the constraints on the constructor's arguments. For example `Event_WellCons` (see Listing 5.3) encodes the property stating that events are partitioned as initialisation event and progress events and `Machine_WellCons` defines well constructed machines (not detailed here because of space limitations reasons).

Machine POs (Semantics of Event-B machines) The proof obligations are formalised upon the semantics of guarded transitions systems. Each proof obligation is formalised using set theory. Predicates over state variables are modelled as sets of states satisfying the predicate and logical connectives are formalised by operations on sets.

```
Mch_THM <predicate> ...
Mch_INV_Init <predicate> (m : Machine(St, Ev))
  direct definition AP(m) ⊆ Inv(m)
Mch_INV_One_Ev <predicate> (m : Machine(St, Ev), e : Ev)
  well−definedness e ∈ Progress(m)
  direct definition BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]] ⊆ Inv(m)
Mch_INV <predicate> (m : Machine(St, Ev))
  direct definition
    Mch_INV_Init(m)∧
    (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
Mch_FIS_Init <predicate> ...
Mch_FIS_One_Ev <predicate> ...
Mch_FIS <predicate> ...
Mch_VARIANT_One_Ev <predicate> ...
Mch_VARIANT <predicate> ...
Mch_NAT_One_Ev <predicate> ...
Mch_NAT <predicate> ...
```

Listing 5.4: Well-defined data type operators (behavioural semantics)

For example, Listing 5.4 describes the induction principle for verifying the invariant PO where `Mch_INV_Init` predicate states that the initialisation event must establish the invariant ($AP(m) \subseteq Inv(m)$) and `Mch_INV_One_Ev` states that a given progress event e must preserve the invariant ($BAP(m)[\{e\}]\ [Inv(m) \cap Grd(m)[\{e\}]] \subseteq Inv(m)$). Last, the `Inv` PO (see. Table 5.2) is formalised by the `Mch_INV` operator as the conjunction of the two previous operators. Likewise, all POs are formalised using the same transformation principle.

check_Machine_Consistency $<predicate>$ $(m : Machine(St, Ev))$
  well−definedness Machine_WellCons(m)
  direct definition $Mch\_THM(m) \wedge Mch\_INV(m) \wedge Mch\_FIS(m) \wedge Mch\_VARIANT(m) \wedge Mch\_NAT(m)$

Listing 5.5: Operator for Event-B machine consistency

Last, the operator `check_Machine_Consistency` of Listing 5.5 is the conjunction of all the predicates formalising the various POs. It formalises an Event-B machine correctness condition. When this predicate is used as a `theorem` in an Event-B system development then the core POs (see Table 5.2) as well as the well-definedness POs are automatically generated by the Rodin platform. Discharging all the generated POs along with this theorem ensures the consistency of the machine.

Instantiation of the meta-theory. The defined meta-theory is instantiated to define specific Event-B machines. Instantiation consists in defining an Event-B context with instances for the type parameters `St` and `Ev` and providing instances for the attributes of `Cons_machine`.

5.3.3 Domain Knowledge in Formal Modelling

This section discusses various works related to research challenges: (1) explicit domain knowledge in formal specification and (2) the well-adopted formalism for representing domain knowledge i.e. ontologies.

Domain Knowledge in Formal Specification

Handling domain knowledge and contextual information in long-term research problem is formal methods. Several approaches and frameworks were developed like in Coq [Bertot & Castéran, 2010], Isabelle/HOL [Nipkow et al., 2002], PVS [Owre et al., 1992], Event-B with theories [J.-R. Abrial et al., 2009; M. J. Butler & Maamria, 2013] (e.g. control theory for control-command systems [Dupont et al., 2020]) and critical systems [Aït Ameur & Méry, 2016], DOL - Distributed Ontology Model and Specification Language based on algebraic specification with CASL [Mossakowski, 2016] integrated to the OntoHub ontology repository and RSL - RAISE Specification Language [Bjørner, 2006, 2017, 2019] for railway systems and shipping, and logistic systems.

The challenge of linking domain knowledge and design models as well as the potential benefits of doing so is discussed in [Henderson-Sellers, 2012]. It includes a mathematical analysis of models and meta models, ontologies, modelling and meta-modelling languages. Design models annotation by domain-specific knowledge has been studied for state-based methods [Aït Ameur & Méry, 2016] as well. More recently, the textbook [Aït Ameur et al., 2021] reviewed many cases of exploiting explicit models of domain knowledge by system models spanning medical systems [Singh et al., 2018, 2021], e-voting systems [Gibson & Raffy, 2021], distributed systems etc.

Last, focusing on Event-B, a proposal for a simplified ontology description language was put forward and illustrated on case studies in [Hacid & Aït Ameur, 2016; Hacid & Ameur, 2017]. This approach was based on context extension where the design models need to discharge proof obligation in form of theorems to completely validate the compliance of the formal design models to the formalised domain knowledge.

In this paper, we are interested in engineering domain ontologies in the view of [Aït Ameur et al., 2017; Jean, Pierra, & Ameur, 2006; Pierra, 2008] to model domain knowledge as Event-B theories and use typing to annotate system design models formalised in Event-B allowing an non-intrusive domain-specific behavioural analysis.

Ontologies and Domain Modelling

Ontologies, as explicit knowledge models [Gruber, 1995], have been extensively studied in the literature and applied in several domains spanning semantic web, artificial intelligence, information systems, system engineering etc.

Several approaches for describing, designing and formalising ontologies for these application domains have been proposed. Models [OWL Working Group, 2009; Pierra & Wiedmer, 1996], browsers like Protégé [2] [Knublauch et al., 2004] or PlibEditor [3], query languages like RQL [Karvounarakis et al., 2002], SPARQL [Prud'hommeaux, 2008], OntoQL [Aït Ameur et al., 2017; Jean, Aït Ameur, & Pierra, 2006], reasoners like Pellet [Sirin & Parsia, 2004], annotators like CREAM [Handschuh & Staab, 2003], Terminae [Despres & Szulman, 2006] or SAWSDL [Kopecký et al., 2007] were proposed.

Domain ontologies have been described for domains such as encyclopedia [Hoffart et al., 2013], logistics and rivers and canals [Bjørner, 2006], transportation systems [Bjørner, 2006; Zayas et al., 2010], electronic components [IEC-61360-4, 1999], bio-informatics [Barrell et al., 2009] etc. Examples of general purpose ontologies are [Maio et al., 2012; Niles & Terry, 2004]. Most of mentioned approaches rely on XML-based formats and pay lot of attention to *web knowledge* and to the *automatisation* characteristic (which may limit the scope of addressed knowledge models). To the best of our knowledge, *principled domain-specific non-intrusive behavioural analyses of formal design models* has not been addressed with above mentioned approaches.

[2] http://protege.stanford.edu/

[3], https://www.iso.org/standard/43423.html

5.4 Domain-Specific Behavioural Analysis

The proposed proposal consists in expressing generic domain knowledge properties, particularly behavioral ones, as Event-B model concepts. Such properties are commonly found in domain requirements or standards.

The proposed framework (see Fig. 5.2) is composed of two basic blocks: ontology modelling language (see Section 5.3.1) and meta-Event-B language (see Section 5.3.2). The first component provides primitives to write domain concepts and constraints as ontologies, while the second component allows for the abstraction of a system as an instance of the meta-Event-B language, allowing for a reasoning extension. Moreover, all the behavioural properties encoded in first-order logic can be written in this proposed framework and validated on models using the proposed methodology. In addition to purely temporal properties, the framework allows expressing enriched analyses that take the domain knowledge into account. A mechanism for referencing domain knowledge in design models is also defined. The framework is composed of three parts, and the resulting step-by-step methodology for analysing Event-B models is divided into four major steps.

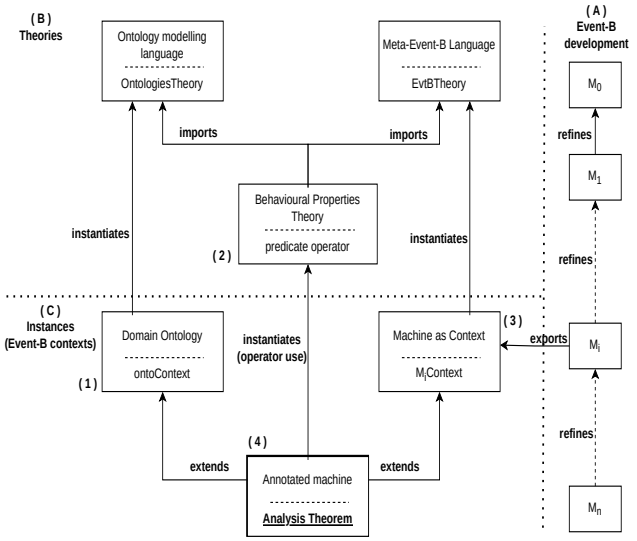


Figure 5.2: Event-B-based framework for domain behavioural properties analysis

5.4.1 Components of the methodology

Three main components have been identified: Event-B development Fig. 5.2.(A), theories Fig. 5.2.(B) and instances as Event-B contexts Fig. 5.2.(C).

(A) Event-B development represents the Event-B model development, including the refinement chain for the system design with an abstract machine $\mathtt{M}_i$

to be analysed;

(B) Theories enabling a designer to manipulate and analyse Event-B models. Three theories have been introduced :

- `EvtBTheory` Event-B theory, introduced in section 5.3.2, allowing to formalise an Event-B model as a context and to define proof obligations;
- An ontology modelling language (`OntologiesTheory` in Listing 5.1 of Section 5.3.1) for describing domain knowledge concepts and constraints.
- A theory importing both theories in order to 1) annotate Event-B models with ontological concepts and to 2) define new POs as predicate operators expressing specific behavioural domain properties and used as theorems on specific Event-B models expressed as instances.

(C) Instances (Event-B contexts). The third part describes the contexts instantiating the theories of Part (B). The context (4) describes the annotated model, which extends the ontology context, the Event-B model context, and defines theorems corresponding to the behavioural properties to be checked. The proof of these theorems guarantees that the properties hold on the analysed machine $\texttt{M}_i$ of part (A).

5.4.2 A Methodology for defining Event-B models domain knowledge based analyses

Our methodology is divided into four steps (see Fig. 5.2).

Step 1: *Ontology definition.* This step consists in describing the domain ontology with the domain concepts and properties as an instance, represented by an Event-B context, of the theory presented in Section 5.3.1.

Step 2: *Express a domain-specific behavioural analysis.* It consists in defining a predicate based on an ontology and applying it to an Event-B model

Step 3: *Export the Event-B machine.* Export the Event-B machine as an instance of the theory *EvtBTheory* to be analysed. This instance allows the machine elements to be explicitly manipulated by the operators of Section 5.3.2 theory.

Step 4: *Annotate the Event-B machine.* Annotate the Event-B machine with the ontology concepts from Step 1. Machine concepts (variables and events) are linked to ontology concepts (tags). The annotated machine is an instance of the theory defining the behavioural properties with a theorem, formalising this property requirement by a theorem to be proved.

The remainder of the article demonstrates the approach using Event-B method.

5.5 Case Study

5.5.1 Informal Description

A critical interactive system is modelled to demonstrate the proposed approach: the user interface of an automatic teller machine (ATM), where the primary requirement is that an authenticated client withdraws banknotes safely. The important requirements associated with HMI are presented below.

- **REQ-1** A user can exclusively use a keyboard or a screen.
- **REQ-2** To withdraw banknotes, a user must be authenticated.
- **REQ-3** A user can adjust the brightness a finite number of times.
- **REQ-4** Any entered passcode must be followed by a confirmation.

First, a user inserts a credit card and chooses an input device to enter a passcode. The user must confirm the entered passcode. Before performing this operation, the user may adjust the brightness of the screen. When the user confirms the input, validation starts. It may result in the acceptance or refusal of the passcode. If the passcode is correct, the ATM delivers banknotes and ejects the card. Otherwise, the user may try again to enter the correct passcode. A user makes new attempts a fixed number of times only. **REQ-4** is of particular interest to this work. Indeed, it is issued from the standard independently of any particular system. In addition, it includes two characteristics: (1) domain-specific knowledge based requirements and (2) expression of a behavioural property that must hold for any system complying the standard.

We would like to draw the reader's attention to the fact that a complete development of the ATM system would be intractable with a frontal approach. Relying on system model refinement and decomposition is more appropriate.

5.5.2 Formal Description in Event-B

This section presents the formal development of the ATM user interface corresponding to Fig. 5.2.(A). It consists of a context and a machine defining static dynamic properties, respectively. Listing 5.6 shows relevant types and constants for modelling the ATM interface. In `axm1` and `axm2`, two enumerated types, `IPT_MOD` and `IRT_STS`, are defined to select possible input devices (keyboard, screen) and credit card modes (in, out), respectively. `MAX_ATP` represents the maximum number of attempts. Contextual information for managing the brightness limit and brightness levels are defined using axioms (`axm7-axm11`). Moreover `axm4` to `axm6` defines the string type with two constants: `EPT_STR` for the empty string and `CRT_KYW` representing the correct password.

CONTEXT $ATMEnvironment$
SETS IPT_MOD, IRT_STS, STR
CONSTANTS $MAX\_ATP, CRT\_KYW, KBD,$
$SCR, IN, OUT, BRT\_LVS, BRT\_MIN, B$
$RT\_MAX, EPT\_STR, MAX\_BRT\_UPD$

AXIOMS
- ***axm1***: $partition(IPT\_MOD, \{KBD\}, \{SCR\})$
- ***axm2***: $partition(IRT\_STS, \{IN\}, \{OUT\})$
- ***axm3***: $MAX\_ATP \in \mathbb{N}1$
- ***axm4***: $CRT\_KYW \in STR$
- ***axm5*** $EPT\_STR \in STR$
- ***axm6***: $CRT\_KYW \neq EPT\_STR$
- ***axm7–8***: $BRT\_MIN \in \mathbb{N} \land BRT\_MAX \in \mathbb{N}$
- ***axm9***: $BRT\_MAX > BRT\_MIN$
- ***axm10***: $BRT\_LVS = BRT\_MIN..BRT\_MAX$
- ***axm11***: $MAX\_BRT\_UPD \in \mathbb{N}$

END

Listing 5.6: Context of the ATM

Listing 5.7 shows an extract of the machine where 14 variables and safety properties are introduced to formalize ATM interactions. Several Events are introduced; `chnBrt` is defined to cover **REQ-3**, for adjusting the brightness of the screen. The guard of `chnBrt` ensures that the maximum number of updates is not exceeded.

MACHINE ATMUserInterface
SEES ATMEnvironment
VARIABLES $str, scrReg, kbdReg, atp, cnfSts, valSts, dlvSts, isStrVis, iptMod, crdSts, brt, brtUpd, cnfKBDstr, cnfSCRstr$
INVARIANTS
- ***inv1–14***: ...
- ***inv15***: $str = scrReg \lor str = kbdReg$
- ***inv16***: $isStrVis = \bot$

EVENTS
- **INITIALISATION**...
- **etrKBDStr**
  - **WHERE**
    - grd1: $0 \leq atp \land atp < MAX\_ATP$
    - grd2–3: $iptMod = KBD \land crdSts = IN$
    - grd4: $cnfKBDstr = \bot$
  - **THEN**
    - $act1 : str, kbdReg :\mid kbdReg' \in STR \land str' = kbdReg'$
    - $act2 : brtUpd, cnfKBDstr := 0, \top$
  - **END**
- **chnBrt**
  - **WHERE**
    - grd1: $crdSts = IN$
    - grd2: $brtUpd < MAX\_BRT\_UPD$
  - **THEN**
    - act1: $brt :\in BRT\_LVS$
    - act2: $brtUpd := brtUpd + 1$
  - **END**
- **cnfKBDStr**
  - **WHERE**
    - grd1: $0 \leq atp \land atp < MAX\_ATP$
    - grd2–3: $iptMod = KBD \land crdSts = IN$
  - **THEN**
    - act1: $atp := atp + 1$
    - act2: $cnfSts := \top$
    - act3: $cnfKBDstr := \bot$
  - **END**

...

Listing 5.7: ATM machine

Other requirements are checked by invariant proving like `inv16`: the entered password is never displayed. **REQ-4** embeds a different kind of properties since (1) it references domain knowledge concepts (2) it is a behavioural property. Due to space limitation, only important variables and safety properties are presented.

5.6 Methodology at work

The methodology's first step (see Section 5.4) is to define an ontology of events which is the basis for modelling domain analyses. The main objective is to check if the ATM model satisfies **REQ-4** stating that when an event annotated as input is activated, a confirmation event will be triggerable. Verifying this requirement necessitates the instantiation of the triptych: `OntologiesTheory`, `EvBTheory` and `predicate operator` (see Figure 5.2). The structure of this section follows the steps of the methodology set up in Section 5.4.

5.6.1 Step 1 - Event Ontology Instantiation (Fig. 5.2.(1))

The ontology modelling language (see. Section 5.3.1) is used to describe Event tags. `input`, `confirmation`, and `finite` are particularly relevant in the ATM case study. The first two tags are used to denote interaction events that provide user input information and formalise a user response. Finally, `finite` designates events that does not occur indefinitely. Listing 5.8 (corresponding to `ontoContext of Fig. 5.2.(1)`) contains the instantiation of the ontology modelling theory. It provides 3 type parameters: `tags` for ontology classes, `Ps` for tag properties, and instances for model events. The other ontology contents are not provided here as they are not relevant for this development.

CONTEXT $EventTagOntology$
SETS $Tags, Ps, Ev$
CONSTANTS $eventOntology, tag, input, confirmation, \ldots, finite$
AXIOMS
axm1: $partition(Tags, \{tag\}, \{input\}, \{confirmation\}, \ldots, \{finite\})$
axm2: $eventOntology \in Ontology(Tags, Ps, Ev)$
axm3–4: $classes(eventOntology) = Tags \land$
$instances(eventOntology) = Ev$
...
END

Listing 5.8: Context for event ontology instantiation

5.6.2 Step 2 - Behaviour Analysis definition (Fig. 5.2.(2))

The definition of the the analysis is composed of 2 phases. First, the terms defining the analysis are specified and then the predicate formed by the conjunction of these conditions is parameterised by the domain ontology. This subsection is divided accordingly and starts with the latter phase.

Domain-Specific Analysis Operator Definition.

The theory for expressing behavioural properties is presented in Listing 5.9. It corresponds to the `predicate operator` in Fig. 5.2.(2). For example, **REQ-4** is formalised using two operators, `isNecFollowedByWD` defining the WD condition of the second one defining the property analysis `isNecFollowedBy`. Indeed, `isNecFollowedBy` verifies whether events annotated by tags in `srcTg` are always followed by events annotated by tags in `trgTg` passing through the intermediate

events annotated with `internalTg`. It asserts that each `EvtInst` event annotated as `srcTg` is reachable in the sense of `is_Reachable` predicate operator.

This operator has 6 arguments: `m` - machine to be analysed, `eo` - ontology to represent the domain concepts and constraints, `srcTg` - source tags, `internalTg` - transit tags, `trgTg` - target tags, and `v` - a list of variants. Note that `insOfC` returns events annotated by tags. To ensure the correct application of this operator, a WD condition is provided: `isNecFollowedByWD` predicate. This operator has six arguments similar to the previous operator. The direct definition of this operator ensures the well-defined ontology ($isWDOntology$), reachability conditions (WD_reach) and satisfies the given variant for each departing event.

THEORY $BehaviouralPropertiesTheory$
IMPORT THEORY $Theo4Reachability, OntologiesTheory$
TYPE PARAMETERS $St, Ev, Tg, Prop$
OPERATORS
isNecFollowedByWD $<predicate>$
$(m : Mach(St, Ev), eo : Ontology(Tg, Prop, Ev),$
$srcTg : \mathbb{P}(Tg), internalTg : \mathbb{P}(Tg), trgTg : \mathbb{P}(Tg),$
$v : \mathbb{P}(E \times \mathbb{P}(St \times \mathbb{Z})))$
direct definition
$isWDOntology(eo) \land srcTg \cup internalTg \cup trgTg \subseteq cls(eo) \land$
$srcTg \neq \emptyset \land trgTg \neq \emptyset \land internalTg \neq \emptyset \land$
$srcTg \cap internalTg = \emptyset \land trgTg \cap internalTg = \emptyset \land$
$(\forall ti \cdot ti \in srcTg \cup internalTg \cup trgTg \Rightarrow insOfC(eo, ti) \neq \emptyset) \land$
$v \in insOfC(eo, srcTg) \rightarrow \mathbb{P}(St \times \mathbb{Z}) \land$
$(\forall i, t \cdot i \in insOfC(eo, srcTg) \land t \in insOfC(eo, trgTg) \Rightarrow$
$WD\_reach(m, i, insOfC(eo, trgTg),$
$insOfC(eo, internalTg), v(i)))$
isNecFollowedBy $<predicate>$
$\boldsymbol{(m : Mach(St, E), eo : Ontology(Tg, Prop, Ev),}$
$\boldsymbol{srcTg : \mathbb{P}(Tg), internalTg : \mathbb{P}(Tg), trgTg : \mathbb{P}(Tg),}$
$\boldsymbol{v : \mathbb{P}(Ev \times \mathbb{P}(St \times \mathbb{Z})))}$
well−definedness $isNecFollowedByWD(m, eo, srcTg, internalTg, trgTg, v)$
direct definition
$\boldsymbol{\forall EvtInst \cdot EvtInst \in insOfC(eo, srcTg) \Rightarrow}$
$\boldsymbol{Is\_Reachable(m, EvtInst, insOfC(eo, trgTg),}$
$\boldsymbol{insOfC(eo, internalTg), v(EvtInst))}$
END

Listing 5.9: Domain Behavioural Properties Theory

Analysis Terms Definition

Listing 5.10, based on `EvtBTheory` in Fig. 5.2.(2), shows reachability theory where several operators are defined for analysing the reachability properties. `WD_reach` operators's definition ensures that the machine m is well constructed (`Machine_WellCons`), the machine invariants are preserved (`Mch_INV`), the target event is not the initialisation event (`Init(m)`), the variant is defined for all reachable states.

Given a source event `src`, a set of intermediary events `es`, and a target event `trg`, `Is_Reachable(m, src, trg, es, v)` holds if, when `src` is activated, then some intermediate events in `es` may be observed finitely many times before `trg` is activated.

The definition of `Is_Reachable` operator states that first, the `src` event must imply either the guards of the events in `es` or the guard of the events in `trg` (`Init_Local_Inv`). Second, the local invariant `Local_Inv_Preserved` must be

preserved by intermediary events. Note that the set `Grd(m)[{trg}]` defined by the local invariant guarantees that the guard of the `trg` always holds. Then, the definition excludes the events not in `es` (`No_Exit` operator). Last, the intermediary events `es` must not be indefinitely active (variants operator `TGMch_NAT` and `TGMch_VARIANT`).

The analysis is strong since it requires that the events imply the guards of the target events. This can be applied for analysis to systems that do not satisfy this condition without losing generality; indeed, by refinement, this can address systems that reach the desired states after many steps; the refinement of the intermediary events does not break the analysis.

THEORY Theo4Reachability **IMPORT THEORY** $EvtBTheory$
TYPE PARAMETERS St, Ev
OPERATORS
next_states <*expression*> $(m : M(St, Ev))$ **direct definition** ...
WD_reach <*predicate*>
$(m : M(St, Ev), src : Ev, trg : \mathbb{P}, es : \mathbb{P}(Ev), v : \mathbb{P}(St \times \mathbb{Z}))$
direct definition
$Machine\_WellCons(m) \land trg \subseteq Progress(m) \land src \in Event(m) \land$
$Inv(m) \lhd v \in Inv(m) \rightarrow \mathbb{Z} \land Mch\_INV(m) \land es \subseteq Progress(m)$
Init_Local_Inv <*predicate*> $(m : M(St, Ev), src : Ev, lInv : \mathbb{P}(St)) \ldots$
Local_Inv_Preserved <*predicate*>
$(m : M(St, Ev), initE : Ev, evs : \mathbb{P}(Ev), lInv : \mathbb{P}(St))$
...
No_Exit <*predicate*> $(m : M(St, Ev), yesE : \mathbb{P}(Ev), noE : \mathbb{P}(Ev),$
$trg : \mathbb{P}(Ev), v : \mathbb{P}(St \times \mathbb{Z}))$
...
TGMch_VARIANT <*predicate*> $(m : M(St, Ev), v : \mathbb{P}(St \times \mathbb{Z}), es : \mathbb{P}(Ev)) \ldots$
TGMch_NAT <*predicate*> $(m : M(St, Ev), v : \mathbb{P}(St \times \mathbb{Z}), es : \mathbb{P}(Ev)) \ldots$
Is_Reachable <*predicate*>
$(m : M(St, Ev), src : Ev, trg : Ev, es : \mathbb{P}(Ev), v : \mathbb{P}(St \times \mathbb{Z}))$
well−definedness $WD\_reach(m, src, trg, es, v)$
direct definition
$Init\_Local\_Inv(m, src, Grd(m)[trg]) \land$
$Local\_Inv\_Preserved(m, src, es, Grd(m)[trg]) \land$
$No\_Exit(m, es, Progress(m) \setminus (es \cup src \cup trg), trg, v)$
$TGMch\_NAT(m, v, es) \land$
$TGMch\_VARIANT(m, v, es) \land$
END

Listing 5.10: Reachability Theory

5.6.3 Step 3 - Exporting Event-B models as instances of the Meta-Event-B theory (Fig. 5.2.(3))

CONTEXT $ATMmEBModel$
EXTENDS $ATMEnvironment, EventTagOntology$
CONSTANTS $istCrd, KBDStr, SCRStr, dlvBnkNts, cnfKBDStr,$
$chnBrt, cnfSCRStr, chkStrCrt, chkStrWrg, ATM, init$
AXIOMS
axm1: $partition(Ev, \{KBDStr\}, \{chnBrt\}, \{cnfKBDStr\},$
$\ldots, \{chkStrWrg\}, \{dlvBnkNts\})$
axm2: $ATM \in Machine(STR \times STR \times STR \times \mathbb{Z} \times BOOL \times BOOL \times$
$BOOL \times BOOL \times IPT\_MOD \times IRT\_STS \times \mathbb{Z} \times$
$\mathbb{Z} \times BOOL \times AMT, Ev)$
axm3: $Event(ATM) = Ev$
axm4: $Grd(ATM) = \{e \mapsto (str \mapsto scrReg \mapsto kbdReg \mapsto atp \mapsto$
$cnfSts \mapsto valSts \mapsto dlvSts \mapsto isStrVis \mapsto iptMod \mapsto$
$crdSts \mapsto brt \mapsto brtUpd \mapsto nStr \mapsto sum) \mid$
$(e = istCrd \land crdSts = OUT) \lor$
$(e = KBDStr \land 0 \leq atp \land atp < MAX\_ATP \land$

$iptMod = KBD \land crdSts = IN \land nStr = \bot) \lor$
$(e = chnBrt \land brtUpd \leq MAX\_BRT\_UPD \land crdSts = IN) \lor$
$(e = cnfKBDStr \land 0 \leq atp \land atp < MAX\_ATP \land$
$iptMod = KBD \land crdSts = IN \land$
$cnfSts = \bot \land valSts = \bot \land nStr = \bot) \lor \ldots\}$
axm5 $BAP(ATM) = \{e \mapsto ((str \mapsto \ldots. \mapsto sum) \mapsto$
$(strp \mapsto \ldots \mapsto sump)) \mid$
$(e = KBDStr \land kbdRegp \in STR \land strp = kbdRegp \land$
$brtUpdp = 0 \land nStrp = \top \land$
$scrReg \mapsto \ldots \mapsto sum = scrRegp \mapsto \ldots \mapsto sump) \lor$
$(e = cnfKBDStr \land atpp = atp + 1 \land cnfStsp = \top \land$
$str \mapsto \ldots \mapsto sum = strp \mapsto \ldots \mapsto sump) \lor$
$(e = chnBrt \land brtp \in BRT\_LVS \land brtUpdp = brtUpd + 1 \land$
$str \mapsto \ldots sum = strp \mapsto \ldots sump) \lor \ldots\}$

THEOREMS
thm : $check\_Machine\_Consistency(ATM)$
END

Listing 5.11: Annotation and analysis context

Each Event-B machine can be formalised as an instance of the Event-B meta-theory (Machine $\mathtt{M}_i$ on Fig. 5.2.(A)) . To define an Event-B machine as an instance, it is enough to instantiate (give values) the `Machine(St, Ev)` attributes at instantiation (see Listing 5.2). The `St` type parameter is substituted by a Cartesian product of the set types of `ATMUserInterface` machine state variables (14 in total) and `Ev` by the set of the events of this machine. Listing 5.11, corresponding to $\mathtt{M}_i$`Context` in Fig. 5.2.(3), shows an extract of the `ATMUserInterface` machine exported as an instance of the meta-Event-B theory. Guards and actions of the events are formalised, as instances, in the `Grd(ATM)` and `BAP(ATM)` sets (axioms `axm4` and `axm5`).

5.6.4 Step 4 - Annotation & analysis (Fig. 5.2.(4))

The final step before checking the property is *annotation*. It links the domain concepts and constraints to the design model. The events are assigned to tags satisfying the subsumption relation. For example, the `cnfKBDStr` is assigned to `textualConfirmation`, `confirmation` and `Tag`.

CONTEXT $AnnotatedMachine$
EXTENDS $EventTagOntology, ATMmEBModel$
CONSTANTS $annotationDef, variantsDef$
AXIOMS
axm1 : $annotationDef = (\{bounded\} \times \{chnBrt\}) \cup$
$(\{inputByKeyboard\} \times \{KBDStr\}) \cup \ldots \cup$
$(\{input\} \times \{KBDStr, SCRStr\}) \cup$
$(\{textualConfirmation\} \times \{cnfKBDStr\}) \cup$
$(\{confirmation\} \times \{cnfSCRStr, cnfKBDStr\}) \cup$
$(\{interaction\} \times \{KBDStr, SCRStr, cnfKBDStr, cnfSCRStr\}) \cup$
$(\{tag\} \times Ev)$
axm2: $classInstances(eventOntology) = annotationDef$
axm3: $variantsDef = \{KBDStr \mapsto \{p \mapsto bright \mapsto ck \mapsto cs \mapsto v$
$\mid p \mapsto bright \mapsto ck \mapsto cs \in State(ATM) \land$
$v = MAX\_BRT\_UPD - bright\}\} \cup$
$\{SCRStr \mapsto \{p \mapsto bright \mapsto ck \mapsto cs \mapsto v$
$\mid p \mapsto bright \mapsto ck \mapsto cs \in State(ATM) \land$
$v = MAX\_BRT\_UPD - bright\}\}$

THEOREMS
isWDOntologyThm: $isWDOntology(eventOntology)$
vThm: $variantsDef \in annotationDef[\{input\}] \rightarrow \mathbb{P}(State(ATM) \times \mathbb{Z})$
anaThm: ***isNecFollowedBy(ATM, eventOntology, {input}, {bounded}, {confirmation, abortion}, variantsDef)***

```
END
```

Listing 5.12: Annotation and analysis context

The model events are annotated using the `annotationDef` relation. They are related to the `eventOntology` via `classInstances`. Indeed, the events are instances of the ontology classes of tags. In Listing 5.12, which materialises `Analysis Theorem` in Fig. 5.2.(4), there are 3 main theorems `isWDOntologyThm`, `vThm` and `anaTh`. The first proves that the `eventOntology` is well-defined as described in Section 5.3.1. The second is important to establish the WD condition of the analysis operator; it ensures that variants are supplied for all events annotated with `input`. The last theorem is the most important, it ensures the correctness of the analysis performed by discharging the generated proof obligations associated to theorems. The proof of application of the predicate operator `isNecFollowedBy` states that the requirement **REQ-4** is satisfied for the `ATMUserInterface`.

5.7 Assessment

This section discusses the evaluation of the framework described in section 5.4 corroborated by the observations of section 5.6. All the models are available via this link [4].

5.7.1 Principled Methodology vs. Ad hoc Analysis.

The prime objective of the methodology is to define a set of principles that can be used for describing domain-specific behavioural analyses. A direct approach would be to follow a shallow paradigm analysis, which consists of incorporating the analysis elements into the model. The methodology provides two modules: (1) the ontology modelling language which allows defining domain knowledge as an ontology, and (2) the Meta-Event-B modelling language which provides a handle to reason on Event-B concepts. Furthermore, refinement is used when possible to overcome the definition of complex analyses (see. section 5.6.2).

5.7.2 Domain-Specific Analyses and Reusability & Sharability.

The ability to parameterise the behavioural analyses with domain-specific constraints and concepts is a significant aspect of the analyses discussed in this article. This enables precise analyses based on concepts and rules drawn from a domain knowledge description. This was the case for the analysis presented in this article, which is based on an event ontology. The framework architecture (see Fig. 5.2) improves the reusability and shareability of the analysis.

[4] https://www.irit.fr/~Ismail.Mendil/recherches/

5.7.3 The Methodology Is Non-Intrusive.

Integrating domain knowledge concepts and constraints directly into the system model could be one approach to investigating the behaviour properties. Such an approach suffers from a lack of generalisation and is intrusive when modelling a system is mixed with analysis details and it may not be scalable. The approach presented in this article allows to avoid these difficulties; indeed, the methodology uses the model as an argument, and the annotation is not intrusive.

5.7.4 Proof-Based Verification.

Another approach would be to boil down the domain knowledge behaviour properties into temporal properties and then use model checking to verify the resulting set of temporal properties. The model is passed as an argument to the analysis procedure alongside the domain knowledge model, which has the advantage of being non-intrusive. Yet, this method meets quickly its limitation when the systems are large and complex due to the classical problem of state explosion. Furthermore, manual translation of domain knowledge constraints is fastidious. The methodology proposed in this article goes beyond this limitation thanks to the proof-based verification. Indeed, the analysis is established by proving a predicate operator where variant and invariant proof obligations are discharged.

5.7.5 Proof & Modelling Effort Reduction.

The methodology proposed in this article reduces proof effort by factorising out common parts such as the proof of the well-definedness analysis and the definition of a collection of lemmas useful at the model side. In addition, the framework components are described once and for all, and they are only proved before being deployed and used for multiple system models. All the proofs performed on the theory side (see Fig. 5.2.(A)) are achived once and for all. They are reused at the machine instance level.

5.8 Conclusion

This article addressed the issue of analysing domain-specific behavioural properties over formal models of systems. The proposal begins by identifying several intermediate subgoals for achieving the main goal and consists of an integrated framework and methodology centred around the Event-B method for investigating non-intrusively behavioural properties mined from domain knowledge. Several challenges are identified: (1) formalising domain knowledge, (2) accessing and manipulating Event-B concepts (3) defining domain-specific behavioural analyses, and (4) annotating and analysing Event-B models. Solutions for all subgoals are proposed in the Event-B setting, indeed (1) domain knowledge is formalised using an ontology modelling language; (2) meta-Event-B is used as a handle for expressing properties on Event-B machines; (3) a methodology for formalising analyses based on the two former theories is presented; and (4) annotating Event-B models

for analysis purposes. The methodology is illustrated through a concrete analysis applied to a real-world case study.

This framework will be used in future work to generalize the ontology of events to include associations between events. In addition, this approach can be exploited for certification purposes. Indeed, a non-intrusive analysis may be carried out for such purposes if certification standards are formalised as theories formalising certification properties. Last, other case studies may be analysed to draw quantitative assessment of our methodology.

Acknowledgment

This study was undertaken as part of the FORMEDICIS (FORmal MEthods for the Development and the engineering of Critical Interactive Systems) ANR-16-CE25-0007 and EBRP (EventB-Rodin-Plus) ANR-19-CE25-0010.

Assessment

This chapter plays a vital role in assessing the usability, scalability, and expressivity of the EB4EB framework, in particular for handling domain knowledge requirements and system modelling behaviour analyses by annotating Event-B models. In this study, we performed domain specific behaviour analyses using the algebraically defined operators of EB4EB and its temporal extension, which includes LTL operators. All generated proof obligations are discharged successfully to ensure the behavioural correctness.

The key advantages of this approach are that it allows for non-intrusive behaviour analysis, which can help to meet standard criteria as well as certification requirements considering domain knowledge. It should be noted that most of the proofs are accomplished interactively; nevertheless, there is some good support for proof rules supplied within the development of the EB4EB framework and its other theory extensions.

Chapter 6

Empowering the Event-B Method

Overview

This chapter presents the results of combining various modelling techniques addressing different classes of problems as well as systems to determine how important it is to carefully define well-definedness conditions for the defined operators and other modelling constructs such as partial functions and relations to ensure invariant preservation through inductive proofs over execution traces and prevent ill-defined state changes. To support this claim, we describe transitions explicitly as partial functions in an Event-B theory, and the associated well-defined conditions of these functions prevent ill-defined transitions during system modelling. Based on our findings, well-definedness conditions are adequate to preserve inductive invariants and the specified safety properties. Furthermore, well-definedness conditions assist in the proving process and encapsulate the required properties to ensure the correct uses of defined operators and modelling constructs. This work is carried out in the Event-B modelling language with the *Theory plugins* and Rodin IDE. Several proof rules are defined for simplifying and rewriting well-defined predicates. It can be used to define reusable higher-order data types and their properties.

Associated paper of this chapter:

- Aït Ameur, Y., Dupont, G., Mendil, I., Méry, D., Pantel, M., Riviere, P., & Singh, N. K. [2022]. Empowering the Event-B Method Using External Theories. *IFM*, *13274*, 18–35

Empowering the Event-B Method Using External Theories

Y. Aït-Ameur[1], G. Dupont[1], I. Mendil[1], D. Méry[2], M. Pantel[1], P. Rivière[1], N. K. Singh[1]

[1] INPT-ENSEEIHT/IRIT, University of Toulouse, France
{yamine,guillaume.dupont, ismail.mendil, marc.pantel, peter.riviere, nsingh}@enseeiht.fr
[2] LORIA, Université de Lorraine and Telecom Nancy, Nancy, France
dominique.mery@loria.fr

Event-B offers a rigorous state-based framework for designing critical systems. Models describe state changes (transitions), and invariant preservation is ensured by inductive proofs over execution traces. In a correct model, such changes transform safe states into safe states, effectively defining a partial function, whose domain prevents ill-defined state changes. Moreover, a state can be formalised as a complex data type, and as such it is accompanied by operators whose correct use is ensured by well-definedness (WD) conditions (partial functions).

This paper proposes to define transitions explicitly as partial functions in an Event-B theory. WD conditions associated to these functions prevent ill-defined transitions in a more effective way than usual Event-B events. We advocate that these WD conditions are sufficient to define transitions that preserve (inductive) invariants and safety properties, thus providing easier and reusable proof methods for model invariant preservation. We rely on the finite automata example to illustrate our approach.

6.1 Introduction

Our proposal stems from the following two extensive research observations:

First, formal state-based methods have demonstrated their ability to model complex systems and reason about them to establish properties reflecting the modelled requirements. In particular, they have proven to be effective in ensuring system safety through the verification of invariant properties. This ensures that each reachable state of the modelled system fulfills these invariants, i.e. the system state is always in a safe region and never leaves it. In general, invariants verification is based on an induction principle over traces of transition systems, i.e. invariants hold in the initial state and if they hold in any state, then they hold in the next state (deterministic) or next states (non-deterministic). The proof is carried out on the formalised model using the associated proof system.

Second, the modelling of complex systems in system engineering relies on domain knowledge that is shared and reused in system models. It contains definitions as well as domain-specific properties. In general, this domain knowledge is formalised as theories with data types, operators, axioms and theorems proved

using the associated proof system, *independently* of the designed models. In these theories, a Well-Definedness (WD) condition is associated to each operator expressing the constraints to be fulfilled for its application (partial function). The theories are used to type concepts in system models, to manipulate them with operators, and finally to establish system specific properties with the help of the axioms and theorems issued from these theories.

Our claim. From our observations, we claim that it is possible to exploit externally defined theories and rely on the associated WD conditions to establish system properties, in particular, safety ones. *The idea consists in formalising state changes (transitions) explicitly as partial function expressed by operators defined in external theories.* The WD conditions associated with each theory operator when discharged as proof obligations (PO) prevent ill-defined transitions.

Objective of this work. In the presence of theories that axiomatise domain specific data types, our approach defines another modelling and proof technique for invariant preservation in Event-B [2]. It relies on the use of automatically generated WD proof obligations associated with operators coded as partial functions, to circumscribe the states of the system under design to a given safety domain.

Organisation of the paper. Next section discusses invariant and WD proof obligations with respect to related work. Section 3 overviews the Event-B concepts needed for our approach. Section 4 describes the formalism of *finite automata*, used to illustrate our approach and Section 5 shows their formalisation as an Event-B model. Section 6 presents our approach, and its correctness is justified in Section 7. Section 8 shows its application on finite automata. Last, a conclusion and future research directions are presented in Section 9.

6.2 Invariants and Well-Definedness (WD)

State-based methods are characterised by the explicit definition of a state, usually characterised by variables as well as a set of actions that modify them. These actions rely on the generalised assignment operation based on the *"becomes such that"* before-after predicate (for deterministic and non deterministic assignments) introduced, in particular, by the seminal work of [J.-R. Abrial, 1996; Dijkstra, 1975; Floyd, 1967; Hoare, 1969]. This operation defines a state transition and it is encapsulated in ASM rules [Börger & Stärk, 2003], substitutions or events in B and Event-B [J.-R. Abrial, 2010], Hoare triples [Hoare, 1969], Guarded Commands (GCL) [Dijkstra, 1975], operations in RSL [George, 1991] and VDM [C. B. Jones, 1986], actions in TLA$^+$ [Lamport, 2002b], schemas in Z [Spivey, 1992] and so on. All these methods provide a proof obligation (PO) generation mechanism that generates proof goals submitted to the underlying method's proof system. These ones are involved in the description and verification of invariants defining safety properties resulting from requirements.

Invariants.

The before-after predicate (BAP) allows to observe the state of a system and state changes in traces describing system behaviours. Inductive-based reasoning defined on such traces establishes properties, in particular invariant preservation. Informally, it states that if a property holds for the initial state and that, for any transition, this property holds before and after this transition, then it holds for every state of the system.

Without loss of generality, let us consider an Event-B guarded event: `WHEN` $G(x)$ `THEN` $x :\mid BAP(x, x')$ `END`. x is a state variable, $G(x)$ a guard (predicate) and $BAP(x, x')$ a BAP relating before x and after x' state variable values. Under $A(s, c)$ axiomatisation of sets and constants definitions, the invariant $I(x)$ preservation PO for such event is $A(s, c) \wedge G(x) \wedge BAP(x, x') \wedge I(x) \implies I(x')$. This PO shall be proved **for each** event of the model.

Well-Definedness (WD).

According to [J. Abrial & Mussat, 2002], Well-Definedness describes the *circumstances under which it is possible to introduce new term symbols by means of conditional definitions in a formal theory as if the definitions in question were unconditional, thus recovering completely the right to subsequently eliminate these symbols without bothering about the validity of such an elimination.* It avoids describing ill-defined operators, formulas, axioms, theorems, and invariants.

In Event-B, each formula is associated to well-definedness POs [Leuschel, 2020] that ensure that the formula is well-defined and that two-valued logic can be used. A WD predicate $WD(f)$ is associated with each formula f. This predicate is defined inductively on the structure of f. For example, if we consider a and b being two integers, P and Q two predicates, f of type $\mathbb{P}(D \times R)$, the following WD definitions can be written as $WD(a \div b) \equiv WD(a) \wedge WD(b) \wedge b \neq 0$, $WD(P \wedge Q) \equiv WD(P) \wedge (P \Rightarrow WD(Q))$, $WD(P \vee Q) \equiv WD(P) \wedge (P \vee WD(Q))$ or $WD(f(a)) \equiv WD(f) \wedge WD(a) \wedge a \in \mathrm{dom}(f) \wedge f \in D \nrightarrow R$ where $\nrightarrow$ denotes a partial function. Once the WD POs are proved, they are added as hypotheses in the proofs of the other POs [J.-R. Abrial, 2010].

Invariants and WD.

When reporting an error in a proof by J-P. Verjus, A. J. M. van Gasteren and G. Tel [van Gasteren & Tel, 1990] identified the concepts of "always-true" and "invariant". In Event-B, "always-true" is expressed using theorems on variables, while "invariant" is expressed as inductive invariants. In addition, invariant properties shall be expressive enough to derive safety properties. Our approach is illustrated on Event-B. We consider a state change as a transformation function on state variables. As this function is partial, it is associated with WD conditions.

Handling WD conditions and partial functions ($\nrightarrow$) definitions in proofs and proof systems is not new. The paper of C.B. Jones [C. B. Jones, 1995] clearly highlights the importance of dealing with such definitions. In formal proof systems, it has been addressed in different manners using two-valued and three-valued logic

(with weak and strong equality), subset types, denotational approaches, type-correct conditions of total functions, etc. [J. Abrial & Mussat, 2002; Barringer et al., 1984; C. B. Jones & Middelburg, 1994; Leuschel, 2020; Nipkow et al., 2002; Owre et al., 1992; Stoddart et al., 1999].

Our proposal.

Our research focuses on state-based modelling with Event-B but may be transferred to other state-based methods. We view a state change (transition) as a partial function $Trans : State \nrightarrow \mathbb{P}(State)$ (or $Trans : State \nrightarrow State$ for a deterministic system). Here, *State* denotes the Cartesian product of the type of each state variable. As an invariant must restrict state changes to safe states, this function can be seen as a partial function, *well-defined* on the set of safe states $Safe_{St}$ as $Trans_{Inv} : Safe_{St} \nrightarrow \mathbb{P}(State)$. To preserve the invariant, one has to prove that: $\mathbf{ran}(Trans_{Inv}) \subseteq \mathbb{P}(Safe_{St})$.

Based on the definition of such function, our proposal consists in describing an alternative approach to Event-B invariant preservation based on the definition, in an Event-B theory, of a data type T describing a *State* with a set of well-founded operators (well-defined partial functions). An operator $Op(x_1 : T_1, x_2 : T_2, \ldots, x_n : T_n)$ with n arguments returns an expression of type T and is associated to a logical condition of the form $WD(x_1, x_2, \ldots, x_n)$ stating that $x_1, x_2, \ldots, x_n \in \mathbf{dom}(Op)$. Each operator describes safe state changes according to a given *reusable* property *independently* of any model. Below, we show how this approach works for Event-B models.

6.3 Overview of Event-B

Event-B [J.-R. Abrial, 2010] is a *correct-by-construction* method based on set theory and first order logic (FOL). It relies on an expressive state-based modelling language where a set of events models state changes.

6.3.1 Contexts and machines (Tables 6.1.b and 6.1.c)

A `Context` component describes the static properties of a model. It introduces the definitions, axioms and theorems needed to describe the required concepts using *carrier sets* s, *constants* c, *axioms* A and *theorems* T_{ctx}. A `Machine` describes the model behaviour as a transition system. A set of guarded events is used to modify a set of state variables using Before-After Predicates (BAP) to record state changes. Machines are made of *variables* x, *invariants* $I(x)$, *theorems* $T_{mch}(x)$, *variants* $V(x)$ and *events* evt (possibly guarded by G and/or parameterized by α).

Refinements. Refinement (not used in this paper) decomposes a *machine* into a less abstract one with more design decisions (refined states and events) moving from an abstract level to a less abstract one (simulation relationship). Gluing

Theory	Context	Machine
THEORY Th **IMPORT** Th1, ... **TYPE PARAMETERS** E, F, ... **DATATYPES** **Type1**(E, ...) **constructors** **cstr1**(p_1: T_1, ...) **OPERATORS** **Op1** <nature> (p_1: T_1, ...) **well−definedness** $WD(p_1, ...)$ **direct definition** D_1 **AXIOMATIC DEFINITIONS** **TYPES** A_1, ... **OPERATORS** **AOp2** <nature> (p_1: T_1, ...): T_r **well−definedness** $WD(p_1, ...)$ **AXIOMS** A_1, ... **THEOREMS** T_1, ... **END**	**CONTEXT** Ctx **SETS** s **CONSTANTS** c **AXIOMS** A **THEOREMS** T_{ctx} **END**	**MACHINE** M **SEES** Ctx **VARIABLES** x **INVARIANTS** $I(x)$ **THEOREMS** $T_{mch}(x)$ **VARIANT** $V(x)$ **EVENTS** **EVENT** evt **ANY** α **WHERE** $G(x, \alpha)$ **THEN** $x :\mid BAP(\alpha, x, x')$ **END** ... **END**
(a)	(b)	(c)

Table 6.1: Global structure of Event-B Theories, Contexts and Machines

invariants relating abstract and concrete variables ensure property preservation. We do not give more details on refinement as the approach we propose applies to any Event-B machine being either a root machine or a refinement machine.

Proof Obligations (PO) and Property Verification. Table 6.2 provides a set of automatically generated POs to guarantee Event-B machines consistency.

(1)	Ctx Theorems (ThmCtx)	$A(s, c) \Rightarrow T_{ctx}$ `(For contexts)`
(2)	Mch Theorems (ThmMch)	$A(s, c) \wedge I(x) \Rightarrow T_{mch}(x)$ `(For machines)`
(3)	Initialisation (Init)	$A(s, c) \wedge G(\alpha) \wedge BAP(\alpha, x') \Rightarrow I(x')$
(4)	Invariant preservation (Inv)	$A(s, c) \wedge I(x) \wedge G(x, \alpha) \wedge BAP(x, \alpha, x') \Rightarrow I(x')$
(4)	Event feasibility (Fis)	$A(s, c) \wedge I(x) \wedge G(x, \alpha) \Rightarrow \exists x' \cdot BAP(x, \alpha, x')$
(5)	Variant progress (Var)	$A(s, c) \wedge I(x) \wedge G(x, \alpha) \wedge BAP(x, \alpha, x') \Rightarrow V(x') < V(x)$

Table 6.2: Relevant Proof Obligations

Core Well-definedness (WD). In addition, WD POs are associated to all Event-B built-in operators of the Event-B modelling language. Once proved, these WD conditions are used as hypotheses to prove further proof obligations.

6.3.2 Event-B extensions with Theories

In order to handle more complex and abstract concepts beyond set theory and first-order logic, an Event-B extension for supporting externally defined mathematical objects has been proposed in [J.-R. Abrial et al., 2009; M. J. Butler & Maamria, 2013]. This extension offers the capability to introduce new data types by defining new types, operators, theorems and associated rewrite and inference rules, all bundled in so-called *theories*. Close to proof assistants like Coq [Bertot & Castéran, 2010], Isabelle/HOL [Nipkow et al., 2002] or PVS [Owre et al., 1992], this capability is convenient to model *concepts unavailable in core Event-B*, using data types.

Theory description (See Table 6.1.a). Theories define and make available new data types, operators and theorems. Data types (`DATATYPES` clause) are associated to *constructors*, i.e. operators to build inhabitant of the defined type. These ones may be inductive. A theory may define various *operators* further used in Event-B expressions. They may be FOL *predicates*, or *expressions* producing actual values (<nature> tag). Operator application can be used in other Event-B theories, contexts and/or machines. They *enrich the modelling language* as they occur in the definition of axioms, theorems, invariants, guards, assignments, etc.

Operators may be defined explicitly in the `DIRECT DEFINITION` clause (case of a constructive definition), or defined axiomatically in the `AXIOMATIC DEFINITIONS` clause (a set of axioms). Last, a theory defines a set of axioms (`AXIOMS` clause), completing the definitions, and theorems (`THEOREMS` clause). Theorems are proved from the definitions and axioms. Many theories have been defined for sequences, lists, groups, reals, differential equations, etc.

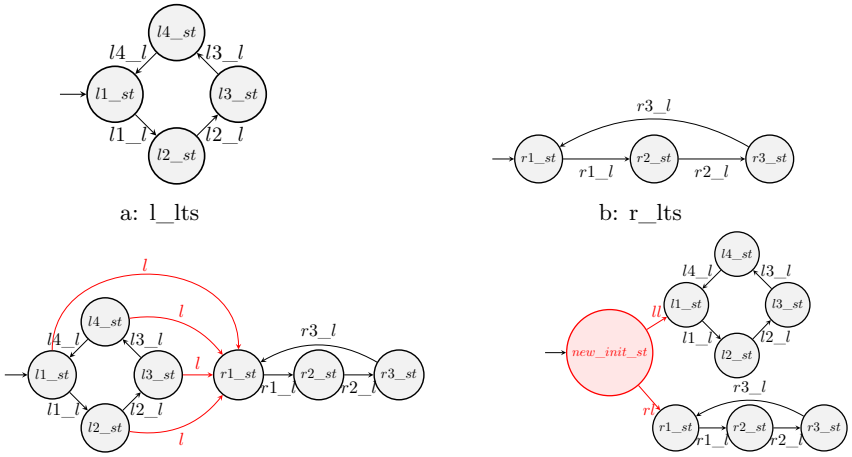
Well-definedness (WD) in Theories. An important feature provided by Event-B theories is the possibility to define *Well-Definedness* (WD) conditions. Each defined operator (thus partially defined) is associated to a condition ensuring its correct definition. When it is applied (in the theory or in an Event-B machine or context), this WD condition generates a proof obligation requiring to establish that this condition holds, i.e. the use of the operator is correct. The theory developer defines these WD conditions for the partially defined operators. All the WD POs and theorems are proved using the Event-B proof system.

Event-B proof system and its IDE Rodin. Rodin[1] is an open source IDE for modelling in Event-B. It offers resources for model editing, automatic PO generation, project management, refinement and proof, model checking, model animation and code generation. Event-B's theories extension is available under the form of a plug-in. Theories are tightly integrated in the proof process. Depending on their definition (direct or axiomatic), operator definitions are expanded either using their direct definition (if available) or by enriching the set of axioms (hypotheses in proof sequents) using their axiomatic definition. Theorems may be imported as hypotheses and, like other theorems, they may be used in proofs. Many provers like first-order logic, or SMT solvers, are plugged to Rodin as well.

6.4 An Illustrative Case Study

We illustrate our approach for invariant preservation by using *finite automata* as a running example. We define finite automata using a set of operators, and consider the *deterministic* property as an invariant property we wish to study. Finite automata are modelled as labelled state transitions systems (LTS). A set of operators are defined on LTS together with a logical property formalising the *deterministic* property.

[1]Rodin Integrated Development Environment http://www.event-b.org/index.html



a: l_lts b: r_lts

c: *PrefixedMerge* application d: *PrefixedUnion* application

Figure 6.1: Examples of LTS operators applications

A $\mathcal{L}ts \in \mathcal{LTS}$ is defined as a tuple $\mathcal{L}ts = (s_0, S, \Sigma, \rightarrow)$ where $s_0 \in S$ is an initial state belonging to the set of states S, Σ an alphabet and $\rightarrow \subseteq S \times \Sigma \times S$ is a transition relation. $\epsilon \in \Sigma$ denotes the empty label.

In order to keep the paper in reasonable length, we focus on two operators: *PrefixedUnion* (Fig. 6.1d) builds the union of two $\mathcal{L}ts$, each of which is prefixed by discriminating labels ll and rl and linked to a new initial state, and *PrefixedMerge* (Fig. 6.1c) merges two $\mathcal{L}ts$ using an intermediate label l. We require that each operation preserves the deterministic property of $\mathcal{L}ts$: if the $\mathcal{L}ts$ fed into these operators are deterministic, then so are their output.

Remark. It is worth noticing that finite automata are our objects of study, and should not be confused with the state-based semantics of Event-B expressed as transition systems.

Next steps. Below, we present two Event-B developments: a classical one (Section 6.5) relying on inductive proofs of the invariant using core Event-B and a second one (Section 6.8), corresponding to the proposed approach relying on the use of externally defined Event-B theories and on WD conditions. This enables us to compare both modelling approaches by highlighting

6.5 Invariant Preservation: Core Event-B

Modelling finite automata in Event-B follows the classical development process of defining the `context` axiomatising the concepts required to model these automata

and the `machine` modelling transformations on them through a set of `events` while ensuring `invariants` (here, determinism) are preserved.

As the purpose of the paper is to show that invariant preservation can be guaranteed using theories and associated WD conditions, only extracts of the model based on the classical approach using Event-B are shown.

An Event-B context for LTS definition.

The `Ltsbasic` context of Listing 6.1 is a set of axioms defining LTS constructs. They introduce a *ConsLts* constructor (`axm5` bijection $\rightarrowtail\!\!\!\!\!\rightarrow$) and accessors to handle any $\mathcal{L}ts \in \mathcal{LTS}$. Last, `axm8` defines a specific LTS, namely *InitLts*, that will be used in the machine for initialisation.

CONTEXT Ltsbasic
SETS S Σ $\mathcal{LTS}$
CONSTANTS init transition state UsedAlphabet ϵ ConsLts init_state
AXIOMS
axm1: $init \in LTS \rightarrow \mathbb{P}(S)$
axm2: $transition \in LTS \rightarrow \mathbb{P}(S \times \Sigma \times S)$
axm3: $state \in LTS \rightarrow \mathbb{P}(S)$
axm4: $UsedAlphabet \in LTS \rightarrow \mathbb{P}(\Sigma)$
axm5: $ConsLts \in (\mathbb{P}(S) \times \mathbb{P}(S) \times \mathbb{P}(\Sigma) \times \mathbb{P}(S \times \Sigma \times S)) \rightarrowtail\!\!\!\!\!\rightarrow LTS$
axm6: $\forall lts, init\_st, tr, s, a \cdot lts = ConsLts(init\_st \mapsto s \mapsto a \mapsto tr) \Leftrightarrow init(lts) = init\_st \wedge transition(lts) = tr \wedge state(lts) = s \wedge UsedAlphabet(lts) = a$
axm7: $\epsilon \in \Sigma \wedge init\_state \in S$
axm8: $InitLts = ConsLts(\{init\_state\} \mapsto \{init\_state\} \mapsto \emptyset \mapsto \emptyset)$
END

Listing 6.1: Basic $\mathcal{L}ts$ constructs.

An Event-B machine to manuipulate LTS.

The objective is to define a set of transformations formalised by events to build a deterministic automaton. The idea is to use a trace of events leading to a deterministic LTS. For this purpose, we use a correct-by-construction method relying on a set of events to build deterministic LTS, preserving the invariant stating LTS determinism.

MACHINE ltsDeterm
SEES Ltsbasic
VARIABLES *lts*
INVARIANTS
inv1: $lts \in LTS$
inv2: $init(lts) \neq \emptyset$ // Init state exists
inv3: $state(lts) = init(lts) \cup dom(dom(transition(lts))) \cup ran(transition(lts))$ //states
inv4: $UsedAlphabet(lts) = ran(dom(transition(lts)))$ // well built Used Alphabet
inv5: $\exists i \cdot init(lts) = \{i\}$ // Unique initial state
inv6: $\epsilon \notin ran(dom(transition(lts)))$ // No ϵ transition
inv7: $transition(lts) \in S \times \Sigma \nrightarrow S$ // Deterministic transition (function)

Listing 6.2: $\mathcal{L}ts$ determinism invariants.

Listing 6.2 shows the list of invariants stating that an LTS is deterministic. `inv2-4` define constraints on the states and labels while `inv5-7` define determinism with single initial state, absence of ϵ label and finally the transition relation is a function (single image $\nrightarrow$).

EVENTS

INITIALISATION $\hat{=}$
THEN
act1: $lts := InitLts$
END
PrefixedMergeEvt $\hat{=}$
ANY $l\_lts, r\_lts, l, \ l\_init\_st, r\_init\_st, l\_st, \ r\_st, l\_UsedAlpha, r\_UsedAlpha$
WHERE
grd1: $l\_lts \in LTS \land r\_lts \in LTS \land l \in \Sigma$
grd2−3: $init(l\_lts) = \{l\_init\_st\} \land init(r\_lts) = \{r\_init\_st\}$
grd4−5: $l\_st = (\{l\_init\_st\} \cup dom(dom(transition(l\_lts))) \cup ran(transition(l\_lts))) \land$
$r\_st = (\{r\_init\_st\} \cup dom(dom(transition(r\_lts))) \cup ran(transition(r\_lts)))$
grd6−7: $l\_UsedAlpha = ran(dom(transition(l\_lts))) \land r\_UsedAlpha = ran(\ldots)$
grd8: $(l\_st \cap r\_st) = \emptyset$
grd9−10: $l \notin ran(dom(transition(l\_lts))) \land l \neq \epsilon$
grd11−12: $\epsilon \notin r\_UsedAlpha \land \epsilon \notin l\_UsedAlpha$
grd13−14: $transition(r\_lts) \in S \times \Sigma \nrightarrow S \land transition(l\_lts) \in S \times \Sigma \nrightarrow S$
THEN
act1: $lts := ConsLts($
$\{l\_init\_st\} \mapsto l\_st \cup r\_st \mapsto$
$(l\_UsedAlpha \cup r\_UsedAlpha \cup \{l\}) \mapsto$
$(transition(l\_lts) \cup transition(r\_lts) \cup \{s \cdot s \in l\_st \mid s \mapsto l \mapsto r\_init\_st\}))$
END
PrefixedUnionEvt $\hat{=}$ ...
...
END

Listing 6.3: Model events building a deterministic $\mathcal{L}ts$.

Listing 6.3 shows the set of events building a LTS (Figure 6.1). Due to space constraints, only the *PrefixedMergeEvt* event is shown. It is parameterised by two LTS (left l_lts and right r_lts) and a connecting label l. It is guarded by conditions ensuring that l_lts and r_lts are well-built and that $l \neq \epsilon$ (`grd1-14`). Action `act1` builds the resulting LTS by updating the state variable lts.

In this approach, it is necessary to describe the invariants ensuring that the state variable defines a deterministic LTS so that the invariant preservation PO (4 of Table 6.2) is discharged.

Although the studied example is well-known and simple, writing these invariants may be a difficult task for the system designer.

6.6 Data type theory-based invariant preservation

The invariant-preservation approach of Section 6.5 relies on an inductive proof where invariants and state variables are directly modelled by the designer using Event-B set theory and its type system (which can be seen as a weak typing system compared to proof assistants like Coq or Isabelle/HOL).

In this approach, the designer describes explicitly safe states, invariants and mandatory guards in the models. The designer has to prove invariant preservation POs for each event. Moreover, this invariant has to be written and proven in further developments, so reusability is compromised. It is possible to design models of systems which exploit externally defined theories to type system features and to manipulate these features using the operators associated to these types *defined as partial functions*. When these operators are used, the WD conditions, associated with them, generate POs on the system design model.

We claim that it is possible to use these WDs to enforce invariants and therefore ensure the safety requirements of the system being designed. In the context of state-based formal methods, this claim is based on the view of invariants as conditions for *well-defined partial functions/transformations*, defined in external theories, with system state type corresponding to one parameter of each of these functions. In addition, the proofs performed on the theory side, achieved once and for all, are reused in system design models verification.

Three main steps are identified. The first one (`Step 1`) is to produce, once and for all, the relevant theories formalising the data types and operators used in the models. The second step (`Step 2`) requires to instantiate the defined theories for the specific types used by the model. Finally, the third step (`Step 3`) uses the defined types and operators for typing and manipulating the state variables.

6.6.1 An Event-B datatype based domain-specific theory (Step 1)

Theories conforming to the template of Listing 6.4 are built and *proved once and for all*. These theories provide generic and parameterised data types, with operators (partial functions) associated to WD conditions and relevant theorems.

THEORY Theo
TYPE PARAMETERS $ArgsTypes$
DATA TYPES
$T(ArgsTypes)$
$Cons(args : ArgsTypes)$
OPERATORS
$\mathbf{Op_1}$ <Predicate> $(el : T(ArgsType), args : ArgsTypes)$
well−definedness condition $\mathbf{WD\_Op_1}(args)$
direct definition $Op\_Exp_1(el, args)$
...
$\mathbf{Op_n}$ <Predicate> $(el : T(ArgsTypes), args : ArgsTypes)$ ...
well−definedness condition $\mathbf{WD\_Op_n}(args)$
direct definition $Op\_Exp_n(el, args)$
Properties <Predicate> $(el : T(ArgsTypes))$
direct definition $properties(el)$
THEOREMS
$\mathbf{ThmTheoOp_1}$: $\forall x, args \cdot x \in T(ArgsTypes) \land args \in ArgsTypes \land$
$\mathbf{WD\_Op_1}(args) \land \mathbf{Op_1}(x, args) \Rightarrow \mathbf{Properties}(x)$
...
$\mathbf{ThmTheoOp_n}$: $\forall x, args \cdot x \in T(ArgsTypes) \land args \in ArgsTypes \land$
$\mathbf{WD\_Op_n}(args) \land \mathbf{Op_n}(x, args) \Rightarrow \mathbf{Properties}(x)$

Listing 6.4: Data type theory template

Listing 6.4 shows a template of theory where the data type T is built from type parameters *ArgsType* and a set of predicate operators Op_i defining relations between concepts of type T and other parameters. *These predicates are used in a model to define before-after predicates* as one of their type parameters $T(ArgsTypes)$ corresponds to the type of the model state.

The predicate *Properties* is defined to capture properties on data of type $T(ArgsTypes)$. It formalises requirements and many of them can be defined.

Last, the central theorems $ThmTheoOp_i$ state that, for each operator Op_i, if its *WD* condition holds then its application implies properties expressed on any element x of type $T(ArgsTypes)$ using the predicate *Properties*. *It can be used to check invariant preservation.* It is worth noticing that these theorems encode

a *structural induction principle*. It is proved *once and for all, independently of any model behaviour description*. Additional theorems, for other properties, characterising the defined data type may be expressed.

6.6.2 An Event-B instantiation context (Step 2)

The theory of Section 6.6.1 is generic. Next step instantiates it with the specific objects of interest, i.e. state manipulated by the Event-B machine.

The context Ctx of Listing 6.5 instantiates the theory of Listing 6.4. Type synthesis and matching of Event-B is used to instantiate the generic data type T with sets s as data type $T(s)$. Then constants and axioms are defined classically.

```
CONTEXT      Ctx
SETS         s
CONSTANTS    c
AXIOMS       ...
THEOREMS
  ThmTheoOp1Inst : ∀x, args · x ∈ T(s) ∧ args ∈ s∧
      WD_Op1(args) ∧ Op1(x, args)  ⇒  Properties(x)
  ...
  ThmTheoOpnInst : ∀x, args · x ∈ T(s) ∧ args ∈ s∧
      WD_Opn(args) ∧ Opn(x, args)  ⇒  Properties(x)
```

Listing 6.5: Context instantiation.

Here again, the important theorems $ThmTheoOp_iInst$ are introduced. They instantiate the generic theorems $ThmTheoOp_i$. As the generic theorems $ThmTheoOp_i$ are already proved, the proofs of these theorems are trivial provided that type checking succeeds. These theorems are the reduction of the polymorphic type of the theory to the concrete type of the model, here the set s.

6.6.3 A domain-specific Event-B machine (Step 3)

At this level, a machine template that exploits the defined theory and the instantiated context is built. Listing 6.6 depicts an Event-B machine with a single state variable x of type $T(s)$ by typing invariant $TypingInv$. Then, a set of events Evt_i, including the initialisation event[2], possibly parameterised, manipulate state variable x. Action `act1` of each event uses operators of the theory in the before-after predicate (BAP) for state variables changes.

```
MACHINE      Machine
SEES         Ctx
VARIABLES    x
INVARIANTS
     TypingInv    :   x ∈ T(s)
     AllowedOper  :   ∃args · args ∈ s ∧ (WD_Op1(args) ∧ Op1(x, args))∨
                                         (WD_Opn(args) ∧ Opn(x, args) ∨ ...)
THEOREMS
     SafThm       :   Properties(x)
EVENTS
     Evt1 ≙          ...          Evtn ≙
       ANY α                        ANY α
       WHEN                         WHEN
         grd1 : α ∈ s ∧ WD_Op1(α)     grd1 : α ∈ s ∧ WD_Opn(α)
       THEN                         THEN
```

[2]For the initialisation event, the guard does not involve state variables.

```
        act1 : x :| Op1(x', α)                    act1 : x :| Opn(x', α)
    END                                       END
END
```

Listing 6.6: An Event-B machine with domain-specific properties

A useful consequence of the use of operators as before-after predicates of events, is the identification of event guards. Indeed, operators application is only possible if their WD conditions in the guards hold.

Two main properties are formalised. First, invariant *AllowedOper* expresses that state variable x is only handled using the operators of the theory when their WD hold; this is *crucial* in the method as it excludes any other type of event from altering x (completeness). Second, theorem *SafThm* captures that the properties *Properties* hold. According to the *ThmMch* PO (Table 6.2), this theorem results from invariants *TypingInv* and *AllowedOper*, together with the axioms and theories of the context Ctx.

6.7 The Proof Process

We have presented a revisited model relying on a data-type-based approach, similar to proof assistants like Coq or Isabelle/HOL, in which a data type and a set of operators are defined to manipulate system states through their type. Then, we have encoded, in the invariant clause, the typing (*TypingInv*) and constraints (*AllowedOper*) corresponding to closure and well-foundedness of operators to strongly type the state variable with respect to the needed operators. Doing so, we embed a stronger type system than Event-B provides. Last, as invariant guarantees typing, safety property *SafThm* can be proved deductively.

Unlike the invariant-based approach of Section 6.5, this approach offers a systematic way to prove invariant preservation. Indeed, proving *SafThm* is straightforward, it is a direct use of the instantiated theorems *ThmTheoOpInst* proved in the context as an instantiation of the generic theorems *ThmTheoOp* of the theory using the modus-ponens proof rule ($\Rightarrow$-elimination rule). Concretely, the proof effort is concentrated on the *reusable* proof *once and for all* of $ThmTheoOp_i$. Other POs are straightforward: *AllowedOper* consists in identifying which allowed operator is used in the disjunction, and *SafThm* is proven deductively using $ThmTheoOp_iInst$ for each allowed operator.

The correctness of this approach consists in establishing that any property P proven true in this approach can be proven true in the classical, invariant-based approach. Concretely, correctness is captured by the $ThmTheo \Rightarrow PO\_Inv$ meta-theorem. We have formalised the proof of this theorem in the Coq [Bertot & Castéran, 2010] proof assistant.

Finally, the approach presented here is similar to encapsulation and application programming interfaces available in programming languages with modules. The *Theo* theory offers a set of generic operators used by models (interface) and the *AllowedOper* invariant encodes encapsulation as the defined state variable is manipulated with the theory-based operators of its type only.

6.8 Revisited Event-B Models for LTS

Back to the case study of deterministic finite automata (Section 6.5), we describe how the proposed approach is applied. We follow the steps identified in Section 6.6.

6.8.1 A data type for LTS (Step 1)

```
THEORY
 TYPE PARAMETERS S , L
 DATA TYPES
  LTS(S, L)
  ConsLts(init : ℙ(S), state : ℙ(S),
     alphabet : ℙ(L),
     transition : ℙ(S × L × S), eps : L)
```

Listing 6.7: A theory of $\mathcal{L}ts$: data-type and constructor.

Listings 6.7, 6.8 and 6.9 describe the theory of $\mathcal{LTS}$ that we have formalised.

Listing 6.7 describes a theory of LTS defining a data type $\mathcal{LTS}(S, L)$ where S and L are types for states and labels, respectively. A constructor *ConsLts* is defined together with accessors to retrieve LTS components (*init*, *state*, *alphabet*, *transition* and ϵ).

```
OPERATORS
   UniqueLabelTransition <predicate> ...
   InitUnique <predicate> ...
   NoEpsTransition <predicate> ...
   GetUniqueInit ...
   WellBuilt <predicate> (lts : LTS(S, L))
     direct definition
       init(lts) ≠ ∅ ∧ alphabet(lts) = ran(dom(transition(lts)))∧
       state(lts) = init(lts) ∪ dom(dom(transition(lts))) ∪ ran(transition(lts))
   IsDeter <predicate> (lts : LTS(S, L))
     direct definition
       WellBuilt(lts) ∧ InitUnique(lts) ∧ NoEpsTransition(lts) ∧ UniqueLabelTransition(lts)
   Wd_ConsLtsDeter <predicate> ...
   ConsLtsDeter <predicate> ...
   ConsSingleStateLts <predicate> ...
   Wd_PrefixedUnion <predicate> ...
   PrefixedUnion <predicate> ...
   Wd_PrefixedUnionDeter <predicate> ...
   PrefixedUnionDeter <predicate> ...
   Wd_PrefixedMerge <predicate> (l_lts : LTS(S, L), l : L, r_lts : LTS(S, L))
     direct definition
       InitUnique(r_lts) ∧ state(l_lts) ∩ state(r_lts) = ∅ ∧ eps(l_lts) = eps(r_lts)
   PrefixedMerge <predicate> (lts : LTS(S, L), l_lts : LTS(S, L), l : L, r_lts : LTS(S, L))
     well−definedness Wd_PrefixedMerge(l_lts, l, r_lts)
     direct definition
       lts = ConsLts
       (init(l_lts), state(l_lts) ∪ state(r_lts), alphabet(l_lts) ∪ alphabet(r_lts) ∪ {l},
       transition(l_lts) ∪ transition(r_lts)∪
             {s, init_r_lts · s ∈ state(l_lts) ∧ init_r_lts = GetUniqueInit(r_lts) |
                 s ↦ l ↦ init_r_lts}, eps(l_lts))
   Wd_PrefixedMergeDeter <predicate> (l_lts : LTS(S, L), l : L, r_lts : LTS(S, L))
     direct definition
       IsDeter(r_lts) ∧ isDeter(l_lts) ∧ state(l_lts) ∩ state(r_lts) = ∅∧
       eps(l_lts) = eps(r_lts) ∧ l ∉ alphabet(l_lts) ∧ l ≠ eps(l_lts)
   PrefixedMergeDeter <predicate> (
       lts : LTS(S, L), l_lts : LTS(S, L), l : L, r_lts : LTS(S, L))
     well−definedness Wd_PrefixedMergeDeter(l_lts, l, r_lts)
     direct definition
       PrefixedMerge(lts, l_lts, l, r_lts)
```

Listing 6.8: A theory of $\mathcal{L}ts$: operators, WD conditions and theorems.

Listing 6.8 shows a subset of operators associated with the $\mathcal{LTS}(S, L)$ data type. In particular, we show the *PrefixedMerge* and *PrefixedUnion* operators,

used in the development of our example (Section 6.5). Each operator is associated to a relevant WD conditions for excluding wrong arguments (partial function). General operators for $\mathcal{LTS}$ are defined: *WellBuilt*, stating that an LTS is correctly built from the constructor using the defined acessors, *IsDeter* asserting that a LTS is deterministic, other operators required to manipulate LTS, such as *UniqueLabelTransition*, *InitUnique*, *NoEpsTranstion*, etc. and other operators required for our case study:

- *ConsLtsDeter*, a derived constructor for deterministic LTS. It restricts the constructor *ConsLts* of $\mathcal{LTS}(S, L)$ with a WD condition *Wd_ConsLtsDeter* to build deterministic LTS only;
- *consSingleStateLts*, a specific operator, used for initialisation, building a LTS with a single state;
- *PrefixedUnion* and *PrefixedMerge* applied to not necessarily deterministic LTS with *Wd_PrefixedUnion* and *Wd_PrefixedMerge* WD conditions;
- deterministic union (*PrefixedUnionDeter*) and merge (*PrefixedMergeDeter*) build deterministic LTS. Their WD conditions (*Wd_PrefixedUnionDeter* and *Wd_PrefixedMergeDeter* respectively) express, in particular, that both l_lts and r_lts parameters shall be deterministic;

Note that each operator outputs the lts parameter from two input parameters l_lts and r_lts. This definition style defines a transformation allowing to write Event-B before-after predicates.

This theory does not **guarantee** that the produced LTS are deterministic if the operators are not applied in the appropriate manner.

THEOREMS
thm1-5 : ...
ThmTheoConsOneSt : $\forall lts, new\_init\_st, \epsilon \cdot lts \in \mathcal{LTS}(S, L) \land \epsilon \in L \land new\_init\_st \in S \land$
ConsSingleStateLts$(lts, new\_init\_st, \epsilon) \Rightarrow$ **IsDeter**(lts)
ThmTheoUnion : $\forall lts, l\_lts, r\_lts, ll, rl, new\_init\_st \cdot$
$l\_lts \in \mathcal{LTS}(S, L) \land r\_lts \in \mathcal{LTS}(S, L) \land ll \mapsto rl \in L \times L \land new\_init\_st \in S \land$
Wd_PrefixedUnionDeter$(new\_init\_st, ll, rl, l\_lts, r\_lts) \land$
PrefixedUnionDeter$(lts, new\_init\_st, ll, rl, l\_lts, r\_lts) \Rightarrow$ **IsDeter**(lts)
ThmTheoMerge : $\forall lts, l\_lts, r\_lts, l \cdot$
$l\_lts \in \mathcal{LTS}(S, L) \land r\_lts \in \mathcal{LTS}(S, L) \land l \in L \land eps(l\_lts) = eps(r\_lts) \land$
Wd_PrefixedMergeDeter$(l\_lts, l, r\_lts) \land$
PrefixedMergeDeter$(lts, l\_lts, l, r\_lts) \Rightarrow$ **IsDeter**(lts)
END

Listing 6.9: A theory of LTS: operators, WD conditions and theorems.

The remaining part of the defined theory (Listing 6.9) contains a set of theorems useful to prove model properties that use the defined types.

In particular, the proven theorems *ThmTheoOp* state that all the operators manipulating deterministic LTS (by the WD condition as hypotheses) produce deterministic automata. Thus, starting with a deterministic automaton, the correct application of any number of operators will always produce a deterministic automaton. As mentioned in Section 6.6.1, these theorems encode structural induction. They guarantee that LTS are effectively deterministic.

6.8.2 An instanciation context for LTS (Step 2)

Next step, following Section 6.6.2, leads to an Event-B context describing specific LTS through theory instantiation. It is obtained by instantiating theory type parameters S and L with *States* and Σ, respectively.

```
CONTEXT    CtxLts
SETS       States    Σ
CONSTANTS  init_state     ϵ
AXIOMS
   axm1: init_state ↦ ϵ ∈ States × Σ // Initialisation
THEOREMS
   ThmTheoConsOneStInst: ∀lts, new_init_st, ϵ · lts ∈ LTS(States, Σ)∧
         ϵ ∈ Σ ∧ new_init_st ∈ States∧
               ConsSingleStateLts(lts, new_init_st, ϵ) ⇒ IsDeter(lts)
   ThmTheoUnionInst: ∀lts, l_lts, r_lts, ll, rl, new_init_st · l_lts ∈ LTS(States, Σ)∧
      r_lts ∈ LTS(States, Σ) ∧ ll ↦ rl ∈ Σ × Σ ∧ new_init_st ∈ States∧
         Wd_PrefixedUnionDeter(new_init_st, ll, rl, l_lts, r_lts)∧
               PrefixedUnionDeter(lts, new_init_st, ll, rl, l_lts, r_lts) ⇒ IsDeter(lts)
   ThmTheoMergeInst: ∀lts, l_lts, r_lts, l · l_lts ∈ LTS(States, Σ) ∧ r_lts ∈ LTS(States, Σ)∧
         l ∈ Σ ∧ eps(l_lts) = eps(r_lts)∧
         Wd_PrefixedMergeDeter(l_lts, l, r_lts)∧
               PrefixedMergeDeter(lts, l_lts, l, r_lts) ⇒ IsDeter(lts)
END
```

Listing 6.10: An instantiated context of LTS.

Useful constants are defined and typed in axiom `axm1`. Theorem *ThmTheoInst* corresponding to the instantiation of the generic theorem *ThmTheo* is also described. Its proof is straightforward by instantiation of the hypotheses.

6.8.3 A data type specific machine for LTS (Step 3)

Last step, corresponding to Section 6.6.3, describes an Event-B machine in Listing 6.11 that is equivalent to the Event-B machine of Listing 6.3.

```
MACHINE  MachineLts
SEES     CtxLts

VARIABLES  lts

INVARIANTS
   TypingInv: lts ∈ LTS(S, L)
   AllowedOper: ∃l_lts, r_lts, ll, rl, l, new_init_st, eps·
         l_lts ∈ LTS(S, L) ∧ r_lts ∈ LTS(S, L) ∧ l ↦ ll ↦ rl ∈ L × L × L∧
         new_init_st ∈ S ∧ eps ∈ L ∧ (
         consOneStateLts(lts, new_init_st, eps)∨
         (Wd_PrefixedUnionDeter(new_init_st, ll, rl, l_lts, r_lts)∧
               PrefixedUnionDeter(lts, new_init_st, ll, rl, l_lts, r_lts))∨
         (Wd_PrefixedMergeDeter(l_lts, l, r_lts)∧
               PrefixedMergeDeter(lts, l_lts, l, r_lts))  )

THEOREMS
   SafThm: IsDeter(lts)
EVENTS
   INITIALISATION ≙
   THEN
      act1: lts :| ConsSingleStateLts(lts′, init_state, ϵ)
   END
   PrefixedMergeEvt ≙
   ANY  l_lts,  r_lts,  l
   WHERE
      grd1: l_lts ↦ r_lts ↦ l ∈ LTS(S, L) × LTS(S, L) × L
      grd2: Wd_PrefixedMergeDeter(l_lts, l, r_lts)
```

```
  THEN
    act1 :  lts :| PrefixedMergeDeter(lts', l_lts, l, r_lts)
  END
  PrefixedUnionEvt  ≙  ...
  ...
END
```

Listing 6.11: A Machine of LTS with a type state variable.

The *MachineLts* of Listing 6.11 describes a single state variable *lts* and two invariants. The first one *TypingInv* types the state variable *lts* with the data type of the instantiated theory.

The second invariant *AllowedOper* states that, once initialised with *ConsSingleStateLts* operator, the *lts* variable can be manipulated with *PrefixedMerge* and *PrefixedUnion* operators only. None of the other operators are allowed to manipulate the state variables (closure and well-foundedness). The *SafThm* theorem states that the *lts* state transition system is deterministic. Finally, the events that manipulate the state variables are defined, they implement machine state changes.

6.8.4 Proof process.

The development presented in this section is a reformulation of the one presented in Section 6.5. However, the mechanism of proof of invariants provided by these two developments differs. In this development, the proofs are eased thanks to the WD conditions associated with each operator and used as guards, together with the proved theorem instantiated in the context. All of the proofs follow and reuse the proof schema shown in Section 6.7.

6.9 Conclusion

In this paper, we have presented an alternative approach for checking invariants and thus safety properties in the Event-B state-based formal method. In the same spirit as proof assistants like Coq or Isabelle/HOL, this approach relies on the extension of Event-B with typing that enforces the checking of conditions related to partial functions using well-definedness (WD) conditions. It consists in encapsulating data types and allowing behavioural models to manipulate typed state variables within events using a subset of operators as long as their WD conditions are met. Furthermore, we demonstrated that it is possible to satisfy invariant preservation proof obligations by using theorems expressed at the data type level and instantiated at the model level.

The defined approach combines algebraically defined data types and their useful properties expressed as WD conditions and theorems (Event-B theories) with behavioural models based on state-transitions systems expressed as a set of state changes using guarded events (Event-B machines). This approach complements the invariant-based approach by enabling inductive proofs at the machine level, or proof of the structural induction principle on the data type theory side. The designer can choose the most appropriate one depending on the difficulty of the modelling and proving activities.

Finally, the proposed approach has been implemented in many different cases: data types, operators and relevant properties (WD and theorems) have been defined for hybrid systems [Dupont et al., 2021] (mathematical extension with a data type for differential equations), interactive critical systems [Mendil et al., 2020][Mendil, Aït Ameur, et al., 2021] [Mendil, Ameur, et al., 2021] (a domain model with a data type for aircraft cockpits widgets) and Event-B models analysis [Riviere, 2021] (Event-B reflexive meta-model with a data type for Event-B states and events).

In the future, we plan to integrate the proposed approach into a refinement chain. On the theory side, we intend to describe refinement of data types so that gluing invariants can be proved using the proposed approach. Studying liveness properties by introducing variants is also targeted. Last, we intend to develop a Rodin plug-in to automate the whole approach.

Assessment

This chapter highlighted essential concepts for identifying and exploiting well-definedness conditions during theory development and system modelling. The usage of well-definedness conditions allows us to define new proof obligations and ensures that inductive invariants and the given safety features are preserved. Furthermore, it aids in the proof mechanisms by giving the necessary hypothesis for the defined theorems and operators. The introduction of well-definedness conditions has considerably boosted the automatic generation of proof obligations and proof automation. Furthermore, such proofs are done once and for all on the theory side. It should be noted that all the defined operators in our EB4EB framework define well-definedness conditions as well as other extensions, such as new types of proof obligations and temporal features, that have been used in their development. It plays a key role in the design of the EB4EB framework as well as its extensions.

Moreover, similar to the identification of well-definedness conditions from the core models, we may extract other core modelling components or their properties into theory using the EB4EB framework that can assist in modelling process as well as simplifying the proving process.

Chapter 7

Conclusion

Contributions

The work presented in this thesis consists in designing a formalised framework for reasoning on state-transitions systems based models expressed as Event-B machines. This framework allows users to manipulate formal models expressed as Event-B machines and define model analyses. It also enables the definition of extensions for Event-B models, providing the ability to conduct additional model analyses beyond what is supported in the core Event-B framework. All the developments we designed have been realised in Event-B, demonstrating that this approach is capable of accommodating modelling and reasoning extensions while still maintaining its ability to support refinement and inductive reasoning for invariant preservation.

A Theory for Event-B. Our main contribution is to formalise every Event-B concept (state variables, guards, before-after predicates, events, invariants, and so on) using an algebraic theory, the EB4EB theory, which manipulates two basic type parameters: *states* and *events*. A constructor and a set of operators are formalised to model and manipulate Event-B concepts as first-order objects, using set theory and first-order logic. Each operator is linked to well-definedness conditions, which generate proof obligations when the operators are used. Finally, a set of proven theorems and proof rules are provided, which can be used to demonstrate the consistency of Event-B models.

Event-B machines as instances of the EB4EB theory: shallow or deep modelling The theory mentioned above can be instantiated to describe Event-B machines. Two instantiation mechanisms have been identified. The first one describes an Event-B context and introduces all the sets that define the various machine components. Theorems relating to machine consistency are also written, and they employ the predicate operators defined in the EB4EB theory. The second mechanism is to define an Event-B machine as a refinement of a "root" generic

machine that includes two events: *initialisation* and *progress*. This mechanism allows you to use Event-B's induction principle, whereas the first involves proving first-order logic theorems.

Extensions for Event-B. The establishment of the EB4EB framework enabled the defining of numerous extensions for Event-B. As Event-B concepts can now be explicitly manipulated, it enables the expression of more advanced, higher-order properties and operations involving these concepts.

- **Reasoning extensions.** The first extension that appears in the definition of EB4EB is related to reasoning. New proof obligations can be expressed and generated automatically for Event-B machines. Here, the logical properties expressed on Event-B machines are formalised. Using this approach, we formalised deadlock freeness, invariant weakness, and reachability proof obligations.

- **Event-B models analyses.** In addition to proof obligation generation, the same framework allowed to define model analyses by querying a machine and checking its robustness. In our work, we introduced additional events (other events external to the Event machine, not to be confused with added events at refinement), known as bad or hidden events, that may be corresponded to non required behaviours (holes in the model) but still satisfy the invariant. This kind of analysis can be used in the security domain to determine whether an attack (described as a bad event) can be triggered. To strengthen the model, we employ the reasoning capabilities extension for invariant weakness checking.

- **Modelling extensions.** The EB4EB framework proved useful in defining other semantic formalisations by extending the EB4EB theory. First, a theory of traces that correspond to Event-B machine traces has been formalised. We used this theory to demonstrate the soundness of the EB4EB theory, specifically that the defined proof obligations are correctly defined on the machine traces. The second modelling extension focuses on domain knowledge modelling. It has been defined in terms of human-computer interaction, where the EB4EB theory has been linked to an ontology theory. This association demonstrated that Event-B's semantics can be extended by associating Event-B concepts with other concepts that correspond to semantic features not available in native Event-B.

- **Support of state and events annotations.** This extension distinguishes itself by combining two other extensions. Indeed, we used temporal logic theory as well as ontologies to express properties that could not be expressed in any of the previous extensions. In a collaborative work, we defined another theory that extends both the temporal logic theory and the ontology theory, allowing us to annotate events with ontology references and use temporal logic property expressions to formalise domain-specific temporal properties on critical interactive systems.

Tool support. Finally, all the development we have done is supported by the Rodin platform and the associated theory plug-in, and all the generated proof obligations have been successfully discharged. In addition, to increase the automation rate of our proofs, we have updated the Rodin tactics by defining new proof rules and rewrite rules. All developments and associated proofs are publicly available and can be found on the https://www.irit.fr/~Peter.Riviere/models/ web page.

Perspectives

The EB4EB framework provides advanced modelling and reasoning capabilities to Event-B, extending its outreach and interoperability.

Below, we highlight new directions made available by the work in this thesis. We have organised them into two categories: *improvements* and *extensions* to Event-B, as well as *development operations* involving Event-B model manipulation.

Event-B improvements and extensions

- **Instantiation and validation.** The definition of contexts, and more generally of logical expressions require checking their consistency, i.e., being inhabited. During our work, we identified three scenarios where the EB4EB framework we developed can be used to address consistency.

  First, *context consistency* consists in checking if their formalisation actually allows some behaviour to exist. In particular, contexts describe elements and hypotheses needed by the system, often represented as abstract sets (akin to types) and axioms. Although abstract sets are non-empty by definition, the method does not require axioms to be consistent. Currently, when model checking fails due to state number explosion, consistency of contexts is expressed manually and in an *ad hoc* manner. When contexts are formalised in the EB4EB framework, elements become accessible as first-class objects, and the consistency of the context can be expressed and thus proven.

  Second, *context validation* is a key issue in the Event-B method, which does not differentiate between definition/prescriptive axioms and properties/descriptive axioms. Checking that so-called *instances* of a context abide by the axioms defined in said context is done in a completely ad hoc way. Continuing the context consistency verification approach, and in the same vein as the context instantiation plug-in defined for Event-B, we can use EB4EB's context formalisation to express properties encoding the correct instantiation of a given context.

  Last, the definition of witness for parameterised events raises the same issue as context extension consistency. Event parameters are limited by the constraints set by the event's guards. Similar to axioms in a context, the Event-B method does not distinguish between *prescriptive* guards, which define the

expected properties of the event parameter, and other guards that only determine when the event is activated. As a result, even if a witness is correct, it may not satisfy any of the guards, effectively disabling the event (which in itself is a perfectly correct refinement, but may not be what the designer is expecting). As with context consistency, the user can perform manual and ad hoc checks that the witness satisfies some guards; however, by leveraging the EB4EB framework's ability to access specific model elements, it is possible to encode and proof "correct witness instances" properties, thereby alleviating this issue.

Other development operations The perspectives that were previously discussed focused on Event-B machines. Here, we outline the viewpoints associated with manipulating Event-B machines.

- **Refinement.** EB4EB currently does not handle Event-B's refinement operation, although its foundations are already present. The formalisation of refinement into the framework supplements its coverage of the method, but also allows to reason on refinement itself. Once a general definition of refinement is incorporated into the EB4EB framework, it becomes possible to write *refinement-related analyses*, i.e. check the properties of a refinement relation between two machines. Moreover, different refinement relationships can be defined. The possibility to *specialise* the refinement operation, restricting its use to some domain elements, while extending its capabilities and semantics may be allowed.

- **Composition/decomposition.** In [Silva & Butler, 2012], the authors propose a mechanism for the decomposition of Event-B machines. The core idea is to enable splitting a machine in parts that may share events and/or states. Composition and decomposition performed in this way is associated to particular proof obligations, and the authors demonstrate, *on paper*, that the correctness of the (de)composition entails the preservation of interesting properties, and in particular that refinement of the composed machine may be transferred to the refinement of its components (and vice versa). The interest of deploying the EB4EB framework here is twofold. First, the framework makes it possible to encode the composition/decomposition mechanism as operators on EB4EB machines, and to provide the proposed proof obligations as predicates to be used in theorems, effectively making the approach available in the framework. Second, the availability of trace-based semantics enables proving the correctness of the mechanism, i.e. to mechanise the soundness proof proposed in the paper. Similarly, the work in [T. S. Hoang et al., 2017] lays out a mechanism for component modularity, empowering machine re-usability and interfacing (through so-called "machine inclusion"). Such an approach is provided in the form of a Rodin plug-in and is associated with proof obligations. Again, the EB4EB can encode this mechanism and verify its correctness, essentially providing a mechanised proof of the plug-in's soundness.

- **Formal model transformations.** Recent advancements in the usage of Event-B show that this method is perfectly capable of handling different types of semantics. Currently, new semantics are usually embedded in an *ad hoc* manner into Event-B models, and the correctness of such embedding is established manually, outside of the method. The EB4EB framework can be extended to propose new natures of *variables* and *events*, effectively encoding, directly in the method, new kinds of semantics, provided they can be expressed using FOL and set theory (which is usually the case). In addition, once these semantics are formalised within the EB4EB framework, model transformations can be formalised using a transformation function formalised on top of the defined semantics. Moreover, since the framework already features the possibility to express traces, meaning it is possible to link the new semantics' embedding in EB4EB with new types of traces, and thus to characterise these new semantics, it becomes possible to check the consistency of the defined transformations as well.

Accessibility and ease of use. The use of the EB4EB framework is generally associated to complex and large proofs. This is due to the encoding of properties to be proved as theorems and well-definedness conditions, which are not decomposed by the proof obligation generator (unlike machine invariants, which are proved for each event rather than for the entire machine). To alleviate this issue, we had to extensively use the proof rules definition capabilities of the theory plug-in, as well as Rodin's rudimentary tactic system. The development of an environment allowing the designers to define domain specific modelling languages and their semantics together with relevant proof rules and tactics will undoubtedly improve the accessibility for non Event-B experts.

Bibliography

Abrial, J.-R. (2010). *Modeling in Event-B: System and software engineering*. Cambridge University Press.

Abrial, J.-R., Butler, M., Hallerstede, S., Leuschel, M., Schmalz, M., & Voisin, L. (2009). *Proposals for mathematical extensions for Event-B* (tech. rep.). http://deploy-eprints.ecs.soton.ac.uk/216/

Abrial, J.-R., Butler, M. J., Hallerstede, S., Hoang, T. S., Mehta, F., & Voisin, L. (2010). Rodin: An open toolset for modelling and reasoning in Event-B. *STTT*, *12*(6), 447–466.

Abrial, J., & Mussat, L. (2002). On using conditional definitions in formal theories. In D. Bert, J. P. Bowen, M. C. Henson, & K. Robinson (Eds.), *ZB 2002: Formal specification and development in Z and b, 2nd international conference of B and Z users, grenoble, france, january 23-25, 2002, proceedings* (pp. 242–269, Vol. 2272). Springer.

Abrial, J.-R. (1996, August). *The B book - assigning programs to meanings*. Cambridge University Press.

Aït Ameur, Y., Baron, M., Bellatreche, L., Jean, S., & Sardet, E. (2017). Ontologies in engineering: The ontodb/ontoql platform. *Soft Comput.*, *21*(2), 369–389.

Aït Ameur, Y., Dupont, G., Mendil, I., Méry, D., Pantel, M., Riviere, P., & Singh, N. K. (2022). Empowering the Event-B Method Using External Theories. *IFM*, *13274*, 18–35.

Aït Ameur, Y., & Méry, D. (2016). Making explicit domain knowledge in formal system development. *Sci. Comput. Program.*, *121*, 100–127.

Aït Ameur, Y., Nakajima, S., & Méry, D. (2021). *Implicit and explicit semantics integration in proof-based developments of discrete systems*. Springer. https://www.springer.com/gp/book/9789811550539

Anand, A., Boulier, S., Cohen, C., Sozeau, M., & Tabareau, N. (2018). Towards certified meta-programming with typed template-coq. In J. Avigad & A. Mahboubi (Eds.), *9th international conference, ITP. part of floc 2018* (pp. 20–39, Vol. 10895). Springer.

Anand, A., Appel, A., Morrisett, G., Paraskevopoulou, Z., Pollack, R., Belanger, O. S., Sozeau, M., & Weaver, M. (2017). CertiCoq: A verified compiler for Coq. *CoqPL Workshop*.

Arcaini, P., Gargantini, A., & Riccobene, E. (2010a). Asmetasmv: A way to link high-level ASM models to low-level nusmv specifications. In M. Frappier, U. Glässer, S. Khurshid, R. Laleau, & S. Reeves (Eds.), *2nd international conference, ABZ* (pp. 61–74, Vol. 5977). Springer.

Arcaini, P., Gargantini, A., & Riccobene, E. (2010b). Automatic review of abstract state machines by meta property verification. In C. A. Muñoz (Ed.), *2nd NASA formal methods symposium - NFM* (pp. 4–13, Vol. NASA/CP-2010-216215).

Arcaini, P., Melioli, R., & Riccobene, E. (2018). Asmetaf: A flattener for the ASMETA framework. In P. Masci, R. Monahan, & V. Prevosto (Eds.), *4th workshop on formal integrated development environment, f-ide@floc* (pp. 26–36, Vol. 284).

Barrell, D., Dimmer, E., Huntley, R. P., Binns, D., O'Donovan, C., & Apweiler, R. (2009). The GOA database in 2009 - an integrated gene ontology annotation resource. *Nucleic Acids Res.*, *37*(Database-Issue), 396–403.

Barringer, H., Cheng, J. H., & Jones, C. B. (1984). A logic covering undefinedness in program proofs. *Acta Informatica*, *21*, 251–269.

Behrmann, G., David, A., & Larsen, K. G. (2004). A tutorial on uppaal. In *Formal methods for the design of real-time systems: International school on formal methods for the design of computer, communication, and software systems, bertinora, italy* (pp. 200–236). Springer.

Bertot, Y., & Castéran, P. (2010). *Interactive theorem proving and program development: Coq'art the calculus of inductive constructions*. Springer Publishing.

Bicarregui, J. C., & Ritchie, B. (1991). Reasoning about VDM developments using the VDM support tool in Mural. In S. Prehn & W. J. Toetenel (Eds.), *Vdm'91 formal software development methods* (pp. 371–388). Springer Berlin Heidelberg.

Bjørner, D. (2006). *Software engineering 3 - domains, requirements, and software design*. Springer.

Bjørner, D. (2017). Manifest domains: Analysis and description. *Formal Aspects Comput.*, *29*(2), 175–225.

Bjørner, D. (2019). Domain analysis and description principles, techniques, and modelling languages. *ACM Trans. Softw. Eng. Methodol.*, *28*(2), 8:1–8:67.

Bodeveix, J., & Filali, M. (2021). Event-B Formalization of Event-B Contexts. *International Conference on Rigorous State-Based Methods (ABZ)*, *12709*, 66–80.

Bodeveix, J., Filali, M., Garnacho, M., Spadotti, R., & Yang, Z. (2015). Towards a verified transformation from AADL to the formal component-based language FIACRE. *Elsevier SCP*, *106*, 30–53.

Boespflug, M., Carbonneaux, Q., Hermant, O., & Saillard, R. (2012). Dedukti: A Universal Proof Checker. *Journées communes LTP - LAC*.

Bonfanti, S., Gargantini, A., & Mashkoor, A. (2018). AsmetaA: Animator for abstract state machines. In M. J. Butler, A. Raschke, T. S. Hoang, & K. Reichl (Eds.), *6th international conference, ABZ* (pp. 369–373, Vol. 10817).

Börger, E., & Stärk, R. F. (2003). *Abstract state machines. A method for high-level system design and analysis*. Springer. http://www.springer.com/computer/swe/book/978-3-540-00702-9

Boulton, R. J., Gordon, A., Gordon, M. J. C., Harrison, J., Herbert, J., & Tassel, J. V. (1992). Experience with embedding hardware description languages in HOL. *IFIP TC10/WG 10.2 International Conference on Theorem Provers in Circuit Design: Theory, Practice and Experience*, 129–156.

Brat, G., Navas, J. A., Shi, N., & Venet, A. (2014). IKOS: A framework for static analysis based on abstract interpretation. *Software Engineering and Formal Methods - 12th International Conference, SEFM 2014*, *8702*, 271–277.

Bühler, D. (2017). *Structuring an abstract interpreter through value and state abstractions:eva, an evolved value analysis for frama-c.* [Doctoral dissertation, University of Rennes 1, France].

Butler, M., & Maamria, I. (2010). Mathematical extension in Event-B through the Rodin theory component.

Butler, M. J., Dghaym, D., Fischer, T., Hoang, T. S., Reichl, K., Snook, C. F., & Tummeltshammer, P. (2017). Formal modelling techniques for efficient development of railway control products. *International Conference on Reliability, Safety, and Security of Railway Systems (RSSRail)*, *10598*, 71–86.

Butler, M. J., Körner, P., Krings, S., Lecomte, T., Leuschel, M., Mejia, L., & Voisin, L. (2020). The first twenty-five years of industrial use of the b-method. *International Conference on Formal Methods for Industrial Critical Systems (FMICS)*, *12327*, 189–209.

Butler, M. J., & Maamria, I. (2013). Practical theory extension in Event-B. *Theories of Programming and Formal Methods - Essays Dedicated to Jifeng He on the Occasion of His 70th Birthday*, *8051*, 67–81.

Carioni, A., Gargantini, A., Riccobene, E., & Scandurra, P. (2008). A scenario-based validation language for asms. In E. Börger, M. J. Butler, J. P. Bowen, & P. Boca (Eds.), *First international conference, ABZ* (pp. 71–84, Vol. 5238). Springer.

Castéran, P. (2021). An explicit semantics for event-b refinements. In Y. Ait-Ameur, S. Nakajima, & D. Méry (Eds.), *Implicit and explicit semantics integration in proof-based developments of discrete systems: Communications of nii shonan meetings* (pp. 155–173). Springer Singapore.

Cimatti, A., Clarke, E. M., Giunchiglia, E., Giunchiglia, F., Pistore, M., Roveri, M., Sebastiani, R., & Tacchella, A. (2002). NuSMV 2: An OpenSource tool for symbolic model checking. *International Conference on Computer Aided Verification (CAV)*, 359–364.

Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., & Rival, X. (2005). The astreé analyzer. *Programming Languages and Systems, 14th European Symposium on Programming, ESOP 2005*, *3444*, 21–30.

Csertán, G., Huszerl, G., Majzik, I., Pap, Z., Pataricza, A., & Varró, D. (2002). VIATRA - visual automated transformations for formal verification and validation of UML models. *ASE*, 267–270.

Despres, S., & Szulman, S. (2006). Terminae method and integration process for legal ontology building. In M. Ali & R. Dapoigny (Eds.), *Advances in applied artificial intelligence* (pp. 1014–1023). Springer Berlin Heidelberg.

Dijkstra, E. W. (1975). Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, *18*(8), 453–457.

Dowek, G. (2015). Deduction modulo theory. *CoRR*, *abs/1501.06523*. http://arxiv.org/abs/1501.06523

Dupont, G., Ameur, Y. A., Singh, N. K., & Pantel, M. (2021). Event-b hybridation: A proof and refinement-based framework for modelling hybrid systems. *ACM Trans. Embed. Comput. Syst.*, *20*(4), 35:1–35:37.

Dupont, G., Yamine Aït Ameur, Pantel, M., & Singh, N. K. (2020). Formally verified architecture patterns of hybrid systems using proof and refinement with Event-B. *Rigorous State-Based Methods - 7th International Conference, ABZ*, *12071*, 169–185.

Ebner, G., Ullrich, S., Roesch, J., Avigad, J., & de Moura, L. (2017). A metaprogramming framework for formal verification. *Proc. ACM Program. Lang.*, *1*(ICFP).

Fallenstein, B., & Kumar, R. (2015). Proof-producing reflection for HOL - With an application to model polymorphism. *Interactive Theorem Proving - 6th International Conference, ITP 2015*, *9236*, 170–186.

Farrell, M., Monahan, R., & Power, J. F. (2016). An institution for Event-B. *Recent Trends in Algebraic Development Techniques (IFIP) WG 1.3 International Workshop (WADT)*, *10644*, 104–119.

Farrell, M., Monahan, R., & Power, J. F. (2017). Combining event-b and CSP: an institution theoretic approach to interoperability. In Z. Duan & L. Ong (Eds.), *19th international conference ICFEM* (pp. 140–156, Vol. 10610). Springer.

Farrell, M., Monahan, R., & Power, J. F. (2022). Building specifications in the Event-B institution. *Springer LMCS*, *18*(4).

Ferrarotti, F., Rivière, P., Schewe, K.-D., Singh, N. K., & Aït Ameur, Y. (2024). A complete fragment of LTL(EB). *13th International Symposium on Foundations of Information and Knowledge Systems FoIKS 24*.

Floyd, R. W. (1967). Assigning meanings to programs. *Reprinted from Proceedings of Symposium in Applied Mathematics , Volume 19 - Mathematical aspects of computer science*, *19*, 19–32.

Fürst, A., Hoang, T. S., Basin, D., Desai, K., Sato, N., & Miyazaki, K. (2014). Code Generation for Event-B. *Integrated Formal Methods*, 323–338.

Gargantini, A., Riccobene, E., & Scandurra, P. (2008). A metamodel-based language and a simulation engine for abstract state machines. *J. Univers. Comput. Sci.*, *14*(12), 1949–1983.

George, C. (1991). The RAISE specification langiage: A tutorial. In S. Prehn & W. J. Toetenel (Eds.), *VDM '91 - formal software development, 4th*

international symposium of VDM europe, proceedings, volume 2: Tutorials (pp. 238–319, Vol. 552). Springer.

Gibson, J. P., & Raffy, J.-L. (2021). Modelling an e-voting domain for the formal development of a software product line: When the implicit should be made explicit. In Y. Ait-Ameur, S. Nakajima, & D. Méry (Eds.), *Implicit and explicit semantics integration in proof-based developments of discrete systems: Communications of nii shonan meetings* (pp. 3–18). Springer Singapore.

Gruber, T. R. (1995). Toward principles for the design of ontologies used for knowledge sharing? *International Journal of Human-Computer Studies*, *43*(5), 907–928. https://doi.org/https://doi.org/10.1006/ijhc.1995.1081

Hacid, K., & Aït Ameur, Y. (2016). Strengthening MDE and formal design models by references to domain ontologies. A model annotation based approach. In T. Margaria & B. Steffen (Eds.), *Leveraging applications of formal methods, verification and validation: Foundational techniques - 7th international symposium, isola 2016, greece, october 10-14, 2016, proceedings, part I* (pp. 340–357, Vol. 9952). https://doi.org/10.1007/978-3-319-47166-2_24

Hacid, K., & Ameur, Y. A. (2017). Handling domain knowledge in design and analysis of engineering models. *Electron. Commun. Eur. Assoc. Softw. Sci. Technol.*, *74*. https://doi.org/10.14279/tuj.eceasst.74.1045

Halchin, A., Ameur, Y. A., Singh, N. K., Ordioni, J., & Feliachi, A. (2020). Handling B models in the PERF integrated verification framework: Formalised and certified embedding. *Science of Computer Programming, Elsevier*, *196*, 102477.

Hallerstede, S., & Hoang, T. S. (2014). Refinement of decomposed models by interface instantiation. *Elsevier SCP*, *94*, 144–163.

Handschuh, S., & Staab, S. (2003). Cream: Creating metadata for the semantic web [The Semantic Web: an evolution for a revolution]. *Computer Networks*, *42*(5), 579–598. https://doi.org/https://doi.org/10.1016/S1389-1286(03)00226-3

He, J., & Hoare, C. A. R. (1998). Unifying theories of programming. *RelMiCS*, 97–99.

Henderson-Sellers, B. (2012). *On the mathematics of modelling, metamodelling, ontologies and modelling languages*. Springer.

Hoang, S., Schneider, S. A., Treharne, H., & Williams, D. M. (2016). Foundations for using linear temporal logic in event-b refinement. *Formal Aspects of Computing*, *28*, 909–935.

Hoang, T. S., & Abrial, J. (2011). Reasoning about liveness properties in event-b. In S. Qin & Z. Qiu (Eds.), *13th international conference on formal engineering methods, ICFEM 2011* (pp. 456–471, Vol. 6991). Springer.

Hoang, T. S., Dghaym, D., Snook, C. F., & Butler, M. J. (2017). A composition mechanism for refinement-based methods. *22nd International Conference on Engineering of Complex Computer Systems, ICECCS*, 100–109.

Hoang, T. S., Schneider, S. A., Treharne, H., & Williams, D. M. (2016). Foundations for using linear temporal logic in Event-B refinement. *Springer FAOC*, *28*(6), 909–935.

Hoare, C. A. R. (1969). An axiomatic basis for computer programming. *Commun. ACM*, *12*(10), 576–580.

Hoare, C. A. R. (1978). Communicating sequential processes. *Commun. ACM*, *21*(8), 666–677.

Hoffart, J., Suchanek, F. M., Berberich, K., & Weikum, G. (2013). YAGO2: A spatially and temporally enhanced knowledge base from wikipedia. *Artif. Intell.*, *194*, 28–61.

Holzmann, G. (2003). *Spin model checker, the: Primer and reference manual* (First). Addison-Wesley Professional.

IEC-61360-4. (1999). *Standard data element types with associated classification scheme for electric components - part 4 : Iec reference collection of standard data element types, component classes and terms* (tech. rep.). International Organization for Standardization.

Jean, S., Aït Ameur, Y., & Pierra, G. (2006). Querying ontology based database using ontoql (an ontology query language). In R. Meersman & Z. Tari (Eds.), *On the move to meaningful internet systems 2006: Coopis, doa, gada, and odbase, OTM confederated international conferences, coopis, doa, gada, and ODBASE 2006, montpellier, france, october 29 - november 3, 2006. proceedings, part I* (pp. 704–721, Vol. 4275). Springer. https://doi.org/10.1007/11914853_43

Jean, S., Pierra, G., & Ameur, Y. A. (2006). Domain ontologies: A database-oriented analysis. In J. Filipe, J. Cordeiro, & V. Pedrosa (Eds.), *Web information systems and technologies, international conferences, WEBIST 2005 and WEBIST 2006. revised selected papers* (pp. 238–254, Vol. 1). Springer. https://doi.org/10.1007/978-3-540-74063-6_19

Jézéquel, J., Barais, O., & Fleurey, F. (2009). Model driven language engineering with kermeta. *GTTSE*, *6491*, 201–221.

Jones, C. B. (1995). Partial functions and logics: A warning. *Inf. Process. Lett.*, *54*(2), 65–67.

Jones, C. B., & Middelburg, C. A. (1994). A typed logic of partial functions reconstructed classically. *Acta Informatica*, *31*(5), 399–430.

Jones, C. B. (1986). *Systematic software development using VDM*. Prentice Hall.

Jones, C. B., Jones, K. D., Lindsay, P. A., & Moore, R. C. (1991). *Mural - a formal development support system*. Springer.

Jouault, F., Allilaire, F., Bézivin, J., Kurtev, I., & Valduriez, P. (2006). ATL: a qvt-like transformation language. *OOPSLA Companion*, 719–720.

Karvounarakis, G., Alexaki, S., Christophides, V., Plexousakis, D., & Scholl, M. (2002). RQL: a declarative query language for RDF. In D. Lassner, D. D. Roure, & A. Iyengar (Eds.), *Proceedings of the eleventh international world wide web conference, WWW 2002, may 7-11, 2002, honolulu, hawaii, USA* (pp. 592–603). ACM.

Knublauch, H., Fergerson, R. W., Noy, N. F., & Musen, M. A. (2004). The protégé OWL Plugin: An open development environment for semantic web appli-

cations. In S. A. McIlraith, D. Plexousakis, & F. van Harmelen (Eds.), *The semantic web – iswc 2004* (pp. 229–243). Springer Berlin Heidelberg.

Kopecký, J., Vitvar, T., Bournez, C., & Farrell, J. (2007). SAWSDL: semantic annotations for WSDL and XML schema. *IEEE Internet Comput.*, *11*(6), 60–67. https://doi.org/10.1109/MIC.2007.134

Kwiatkowska, M., Norman, G., & Parker, D. (2011). PRISM 4.0: Verification of probabilistic real-time systems. In G. Gopalakrishnan & S. Qadeer (Eds.), *International conference on computer aided verification (CAV)* (pp. 585–591, Vol. 6806). Springer.

Lamport, L. (1977). Proving the correctness of multiprocess programs. *IEEE TSE*, *3*(2), 125–143.

Lamport, L. (2002a). Specifying a simple clock. In *Specifying systems, the TLA+ language and tools for hardware and software engineers* (pp. 15–22). Addison-Wesley.

Lamport, L. (2002b). *Specifying systems, the TLA+ language and tools for hardware and software engineers*. Addison-Wesley.

Leroy, X., Blazy, S., Kästner, D., Schommer, B., Pister, M., & Ferdinand, C. (2016). CompCert - A formally verified optimizing compiler. *Embedded Real Time Software and Systems (ERTS)*.

Leuschel, M. (2020). Fast and effective well-definedness checking. In B. Dongol & E. Troubitsyna (Eds.), *Integrated formal methods - 16th international conference, IFM 2020, lugano, switzerland, november 16-20, 2020, proceedings* (pp. 63–81, Vol. 12546). Springer.

Leuschel, M., & Butler, M. (2003). ProB: A Model Checker for B. In *Fme 2003: Formal methods* (pp. 855–874). Springer.

Leuschel, M., & Butler, M. J. (2008). ProB: An automated analysis toolset for the B method. *Springer International Journal STTT*, *10*(2), 185–203.

Maio, C. D., Fenza, G., Furno, D., Loia, V., & Senatore, S. (2012). OWL-FC: an upper ontology for semantic modeling of fuzzy control. *Soft Comput.*, *16*(7), 1153–1164.

Manna, Z., & Pnueli, A. (1984). Adequate proof principles for invariance and liveness properties of concurrent programs. *Elsevier SCP*, *4*(3), 257–289.

Mendil, I., Singh, N. K., Aït-Ameur, Y., Méry, D., & Palanque, P. (2020, December). An Integrated Framework for the Formal Analysis of Critical Interactive Systems. In Y. Liu, S.-P. Ma, S. Chen, & J. Sun (Eds.), *The 27th Asia-Pacific Software Engineering Conference* (p. 10). IEEE.

Mendil, I., Aït Ameur, Y., Singh, N. K., Méry, D., & Palanque, P. A. (2021). Leveraging event-b theories for handling domain knowledge in design models. In S. Qin, J. Woodcock, & W. Zhang (Eds.), *7th international symposium, SETTA* (pp. 40–58, Vol. 13071). Springer.

Mendil, I., Ameur, Y. A., Singh, N. K., Méry, D., & Palanque, P. A. (2021). Standard conformance-by-construction with event-b. In A. Lluch-Lafuente & A. Mavridou (Eds.), *26th international conference, FMICS* (pp. 126–146, Vol. 12863). Springer.

Mendil, I., Riviere, P., Aït Ameur, Y., Singh, N. K., Méry, D., & Palanque, P. A. (2022). Non-intrusive annotation-based domain-specific analysis to cer-

tify event-b models behaviours. *29th Asia-Pacific Software Engineering Conference, APSEC*, 129–138.

Méry, D., & Poppleton, M. (2017). Towards an integrated formal method for verification of liveness properties in distributed systems: With application to population protocols. *SoSyM*, *16*(4), 1083–1115.

Méry, D., & Singh, N. K. (2011). Automatic code generation from Event-B models. *Symposium on Information and Communication Technology*, 179–188.

Mitra, S., & Archer, M. (2004). PVS strategies for proving abstraction properties of automata. *International Workshop on Strategies in Automated Deduction*, *125*, 45–65.

Mitra, S., & Archer, M. (2005). PVS strategies for proving abstraction properties of automata [Proceedings of the 5th International Workshop on Strategies in Automated Deduction (Strategies 2004)]. *Electronic Notes in Theoretical Computer Science*, *125*(2), 45–65.

Mossakowski, T. (2016). The distributed ontology, model and specification language - DOL. In *23rd IFIP WG 1.3 international workshop, WADT* (pp. 5–10, Vol. 10644). Springer.

Muñoz, C., & Rushby, J. (1999). Structural embeddings: Mechanization with method. *International Symposium on FFormal Methods*, 452–471.

Niles, I., & Terry, A. (2004). The MILO: A general-purpose, mid-level ontology. In H. R. Arabnia (Ed.), *Proceedings of the international conference on information and knowledge engineering. ike'04, june 21-24, 2004, nevada, USA* (pp. 15–19). CSREA Press.

Nipkow, T., Paulson, L. C., & Wenzel, M. (2002). *Isabelle/HOL - A proof assistant for higher-order logic* (Vol. 2283). Springer.

OWL Working Group, W. (2009). *OWL 2 Web Ontology Language: Document Overview* [Available at http://www.w3.org/TR/owl2-overview/].

Owre, S., Rushby, J. M., & Shankar, N. (1992). PVS: A prototype verification system. *Automated Deduction - CADE-11, 11th International Conference on Automated Deduction*, *607*, 748–752.

Paul van der Walt. (2012). *Reflection in Agda* [Master's thesis, University of Utrecht, Department of Computing Science].

Pierra, G. (2008). Context representation in domain ontologies and its use for semantic integration of data. *Journal on Data Semantics*, *10*, 174–211.

Pierra, G., & Wiedmer, H. (1996). *Industrial automation systems and integration parts library part 42: Methodology for structuring part families* (tech. rep.). Technical Report ISO DIS 13584-42, International Organization for Standardization, 30 May 1996. ISO/TC 184/SC4/WG2.

Pnueli, A., Siegel, M., & Singerman, E. (1998). Translation validation. *International Conference on Tools and Algorithms for Construction and Analysis of Systems(TACAS)*, *1384*, 151–166.

Prud'hommeaux, E. (2008). Sparql query language for rdf, w3c recommendation. *http://www. w3. org/TR/rdf-sparql-query/*.

Riccobene, E., & Scandurra, P. (2004). Towards an interchange language for ASMs. *International Workshop on Abstract State Machines, Advances in Theory and Practice (ASM)*, *3052*, 111–126.

Riviere, P. (2021). Formal Meta Engineering Event-B: Extension and Reasoning The EB4EB Framework. In A. Rashke & D. Méry (Eds.), *8th international conference, ABZ. proceedings* (Vol. 12709). Springer.

Riviere, P., Kobayashi, T., Singh, N. K., Ishikawa, F., Aït Ameur, Y., & Dupont, G. (2024,(Submitted)). On-the-Fly Proof-Based Verification of Reachability in Autonomous Vehicle Controllers Relying on Goal-Aware RSS.

Riviere, P., Singh, N. K., & Aït Ameur, Y. (2021). Data-types definitions: Use of Theory and Context instantiations Plugins. *9th Rodin User and Developer Workshop collocated with the ABZ 2021 Conference*, 1–6.

Riviere, P., Singh, N. K., & Aït Ameur, Y. (2022a). EB4EB: A Framework for Reflexive Event-B. *International Conference on Engineering of Complex Computer Systems, ICECCS 2022*, 71–80.

Riviere, P., Singh, N. K., & Aït Ameur, Y. (2022b). Reflexive Event-B: Semantics and Correctness the EB4EB Framework. *IEEE Transactions on Reliability*, 1–16.

Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. (2023a). Formalising liveness properties in Event-B. *NASA Formal Methods 2023*.

Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. (2023b). Proof automation for Event-B theories. *10th Rodin User and Developer Workshop collocated with the ABZ 2023 Conference*.

Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. (2023c). Standalone Event-B models analysis relying on the EB4EB meta-theory. *International Conference on Rigorous State Based Methods, ABZ 2023*.

Riviere, P., Singh, N. K., Aït Ameur, Y., & Dupont, G. (2024,(Submitted)). Extending the EB4EB framework with parameterised events.

Scandurra, P., Arnoldi, A., Yue, T., & Dolci, M. (2012). Functional requirements validation by transforming use case models into abstract state machines. In S. Ossowski & P. Lecca (Eds.), *ACM symposium SAC* (pp. 1063–1068). ACM.

Schneider, S. A., Treharne, H., & Wehrheim, H. (2011). A CSP account of event-b refinement. In J. Derrick, E. A. Boiten, & S. Reeves (Eds.), *15th international refinement workshop, refine@fm* (pp. 139–154, Vol. 55).

Schneider, S. A., Treharne, H., & Wehrheim, H. (2014). The behavioural semantics of event-b refinement. *Formal Aspects Comput.*, *26*(2), 251–280. https://doi.org/10.1007/s00165-012-0265-0

Sheard, T., & Jones, S. P. (2002). Template meta-programming for haskell. *SIGPLAN Not.*, *37*(12), 60–75. https://doi.org/10.1145/636517.636528

Silva, R., & Butler, M. (2012). Shared Event Composition/Decomposition in Event-B. In B. K. Aichernig, F. S. de Boer, & M. M. Bonsangue (Eds.), *Formal methods for components and objects* (pp. 122–141). Springer.

Singh, N. K. (2013). *Using Event-B for critical device software systems*. Springer.

Singh, N. K., Aït Ameur, Y., & Méry, D. (2021). Formal ontological analysis for medical protocols. In Y. Aït Ameur, S. Nakajima, & D. Méry (Eds.), *Implicit and explicit semantics integration in proof-based developments of discrete systems: Communications of nii shonan meetings* (pp. 83–107). Springer Singapore.

Singh, N. K., Ameur, Y. A., & Méry, D. (2018). Formal ontology driven model refactoring. *23rd International Conference on Engineering of Complex Computer Systems, ICECCS 2018*, 136–145.

Sirin, E., & Parsia, B. (2004). Pellet: An OWL DL reasoner. In V. Haarslev & R. Möller (Eds.), *Proceedings of the 2004 international workshop on description logics (dl2004), whistler, british columbia, canada, june 6-8, 2004* (Vol. 104). CEUR-WS.org. http://ceur-ws.org/Vol-104/30Sirin-Parsia.pdf

Sozeau, M., Anand, A., Boulier, S., Cohen, C., Forster, Y., Kunze, F., Malecha, G., Tabareau, N., & Winterhalter, T. (2020). The MetaCoq Project. *J. Autom. Reason.*, *64*(5), 947–999.

Spivey, J. M. (1985). *Understanding Z : A specification language and its formal semantics* [Doctoral dissertation, University of Oxford, UK].

Spivey, J. M. (1992). *Z notation - a reference manual (2. ed.)* Prentice Hall.

Stoddart, B., Dunne, S., & Galloway, A. (1999). Undefined expressions and logic in Z and B. *Formal Methods Syst. Des.*, *15*(3), 201–215.

Stump, A. (2016). *Verified functional programming in agda.* Association for Computing Machinery; Morgan & Claypool.

Su, W., & Abrial, J. (2017). Aircraft landing gear system: Approaches with Event-B to the modeling of an industrial system. *Springer International Journal STTT*, *19*(2), 141–166.

Sun, J., Liu, Y., Dong, J. S., & Pang, J. (2009). Pat: Towards flexible verification under fairness. In A. Bouajjani & O. Maler (Eds.), *International conference on computer aided verification (CAV)* (pp. 709–714). Springer.

Taha, W., & Sheard, T. (1997). Multi-stage programming with explicit annotations. *SIGPLAN Not.*, *32*(12), 203–217.

van Gasteren, A. J. M., & Tel, G. (1990). Comments on "on the proof of a distributed algorithm": Always-tru is not invariant. *Inf. Process. Lett.*, *35*(6), 277–279.

Woodcock, J., & Cavalcanti, A. (2004). A tutorial introduction to designs in unifying theories of programming. *IFM*, *2999*, 40–66.

Zayas, D. S., Monceaux, A., & Aït Ameur, Y. (2010). Knowledge models to reduce the gap between heterogeneous models: Application to aircraft systems engineering. In R. Calinescu, R. F. Paige, & M. Z. Kwiatkowska (Eds.), *15th IEEE international conference on engineering of complex computer systems, ICECCS 2010, oxford, united kingdom, 22-26 march 2010* (pp. 355–360). IEEE Computer Society.

Zhu, C., Butler, M., Cirstea, C., & Hoang, T. S. (2023). A fairness-based refinement strategy to transform liveness properties in Event-B models. *Elsevier SCP*, *225*, 102907.

Part III

Appendices

Appendix A

Theories

A.1 EB4EB Core Theories

A.1.1 Core definition of EB4EB:

Listing A.1: Theory of the Syntax representation of Event-B

```
THEORY EvtBStruc
  TYPE PARAMETERS STATE, EVENT
  DATA TYPES
    Machine(STATE, EVENT)
    CONSTRUCTORS
      Cons_machine(
        Event : ℙ(EVENT),
        State : ℙ(STATE),
        Init : EVENT,
        Progress : ℙ(EVENT),
        AP : ℙ(STATE),
        Grd : ℙ(EVENT × STATE),
        BAP : ℙ(EVENT × (STATE × STATE)),
        Inv : ℙ(STATE),
        Thm : ℙ(STATE),
        Ordinary : ℙ(EVENT),
        Variant : ℙ(STATE × ℤ),
        Convergent : ℙ(EVENT))
  OPERATORS
    Grd_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        dom(Grd(m)) = Progress(m)
    BAP_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        dom(BAP(m)) = Progress(m)
```

```
    Event_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        partition(Event(m), {Init(m)}, Progress(m))
    Variant_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ◁ Variant(m) ∈ Inv(m) → ℤ
    Tag_Event_WellCons predicate (m : Machine(EVENT, STATE))
      direct definition
        partition(Event(m), Ordinary(m), Convergent(m))
        ∧Init(m) ∈ Ordinary(m)
    Machine_WellCons predicate (m : Machine(STATE, EVENT))
      direct definition
        BAP_WellCons(m)∧
        Grd_WellCons(m)∧
        Event_WellCons(m)∧
        Tag_Event_WellCons(m)∧
        Variant_WellCons(m)
END
```

Listing A.2: Theory of the Proof obligation definition

```
THEORY EvtBPO
  IMPORT THEORY EvtBStruc
  TYPE PARAMETERS STATE, EVENT
  OPERATORS
    Mch_THM predicate (m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ⊆ Thm(m)
    Mch_INV_Init predicate (m : Machine(STATE, EVENT))
      direct definition
        AP(m) ⊆ Inv(m)
    Mch_INV_One_Ev predicate (m : Machine(STATE, EVENT),
        e : EVENT)
      well−definedness e ∈ Progress(m)
      direct definition
        BAP(m)[{e}][Inv(m) ∩ Grd(m)[{e}]] ⊆ Inv(m)
    Mch_INV predicate (m : Machine(STATE, EVENT))
      direct definition
        Mch_INV_Init(m)∧
        (∀e · e ∈ Progress(m) ⇒ Mch_INV_One_Ev(m, e))
    Mch_FIS_Init predicate (m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ∩ AP(m) ≠ ∅
    Mch_FIS_One_Ev predicate (m : Machine(STATE, EVENT),
        e : EVENT)
      well−definedness e ∈ Progress(m)
      direct definition
```

$Inv(m) \cap Grd(m)[\{e\}] \subseteq dom(BAP(m)[\{e\}])$

Mch_FIS ***predicate*** $(m : Machine(STATE, EVENT))$
direct definition
$Mch\_FIS\_Init(m) \wedge$
$(\forall e \cdot e \in Progress(m) \Rightarrow Mch\_FIS\_One\_Ev(m, e))$

Mch_VARIANT_One_Ev ***predicate*** $($
$m : Machine(STATE, EVENT), e : EVENT, s : STATE)$
well−definedness $Variant\_WellCons(m),$
$Mch\_INV\_One\_Ev(m, e), e \in Progress(m), e \in Convergent(m),$
$s \in Inv(m), s \in Grd(m)[\{e\}]$
direct definition
$\forall sp \cdot sp \in BAP(m)[\{e\}][\{s\}]$
$\Rightarrow (Inv(m) \lhd Variant(m))(s) > (Inv(m) \lhd Variant(m))(sp)$

Mch_VARIANT ***predicate*** $(m : Machine(STATE, EVENT))$
well−definedness $Variant\_WellCons(m), Mch\_INV(m),$
$BAP\_WellCons(m), Tag\_Event\_WellCons(m),$
$Event\_WellCons(m)$
direct definition
$\forall e, s \cdot e \in Event(m) \wedge$
$e \in Convergent(m) \wedge s \in State(m) \wedge s \in Inv(m) \wedge$
$s \in Grd(m)[\{e\}]$
$\Rightarrow Mch\_VARIANT\_One\_Ev(m, e, s)$

Mch_NAT_One_Ev ***predicate*** $(m : Machine(STATE, EVENT),$
$e : EVENT)$
well−definedness $e \in Convergent(m)$
direct definition
$Variant(m)[Inv(m) \cap Grd(m)[\{e\}]] \subseteq \mathbb{N}$

Mch_NAT ***predicate*** $(m : Machine(STATE, EVENT))$
direct definition
$Variant(m)[Inv(m) \cap Grd(m)[Convergent(m)]] \subseteq \mathbb{N}$

check_Machine_Consistency ***predicate*** $($
$m : Machine(STATE, EVENT))$
well−definedness $Machine\_WellCons(m)$
direct definition
$Mch\_THM(m) \wedge$
$Mch\_INV(m) \wedge$
$Mch\_FIS(m) \wedge$
$Mch\_NAT(m) \wedge$
$Mch\_VARIANT(m)$

END

A.1.2 Generic Machine for Shallow modelling

Listing A.3: Context of the generic machine of shallow modelling

CONTEXT

```
    ShallowCtx
SETS
    S
    EV
CONSTANTS
    m
AXIOMS
    axm1: m ∈ Machine(S, EV)
END
```

Listing A.4: Generic machine for the shallow modelling

```
MACHINE
    ShallowMch
SEES
    ShallowCtx
VARIABLES  s , InitDone
INVARIANTS
    inv1: s ∈ S
    inv2: InitDone ∈ BOOL
    inv3: InitDone = TRUE ⇒ s ∈ Inv(m)
EVENTS
    INITIALISATION
    THEN
        act1: s :∈ S
        act2: InitDone := FALSE
    END

    Do_Init
    WHERE
        grd1: InitDone = FALSE
        grd2: Mch_INV_Init(m) ∧ Mch_FIS_Init(m)
    THEN
        act1: s :∈ AP(m)
        act2: InitDone := TRUE
    END

    Do_Ordinary
    ANY  e
    WHERE
        grd1: InitDone = TRUE
        grd2: e ∈ Progress(m) ∧ e ∈ Ordinary(m)
        grd3: s ∈ Grd(m)[{e}]
        grd4: Mch_INV_One_Ev(m, e) ∧ Mch_FIS_One_Ev(m, e)
    THEN
        act1: s :∈ BAP(m)[{e}][{s}]
    END
```

Do_Convergent
ANY e
WHERE
grd1 : $InitDone = TRUE$
grd2 : $e \in Progress(m) \land e \in Convergent(m)$
grd3 : $s \in Grd(m)[\{e\}]$
grd4 : $Mch\_INV\_One\_Ev(m, e) \land Mch\_FIS\_One\_Ev(m, e)$
grd6 : $Variant\_WellCons(m)$
grd5 : $Mch\_VARIANT\_One\_Ev(m, e, s) \land Mch\_NAT\_One\_Ev(m, e)$
THEN
act1 : $s :\in BAP(m)[\{e\}][\{s\}]$
END

END

A.1.3 Helper Theory

Listing A.5: Helper Theory for Event-B machine manipulation

THEORY EvtBManip
IMPORT THEORY
EvtBTheory
TYPE PARAMETERS $EVENT, STATE$
OPERATORS
Get_next_act_state ***expression*** $(m : Machine(STATE, EVENT))$
well−definedness $Event\_WellCons(m)$
direct definition
$\{e, s \cdot$
$((e = Init(m) \land s = AP(m)) \lor$
$(e \in Progress(m) \land s = BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]]))$
$\mid e \mapsto s\}$
THEOREMS
thm1 :
$\forall e, m \cdot m \in Machine(STATE, EVENT) \land e \in Event(m) \land$
$Event\_WellCons(m) \Rightarrow$
$e \in dom(Get\_next\_act\_state(m))$
thm2 :
$\forall m \cdot m \in Machine(STATE, EVENT) \land Event\_WellCons(m) \Rightarrow$
$Get\_next\_act\_state(m) \in EVENT \pfun \mathbb{P}(STATE)$
PROOF RULES
extension_def :
Metavariables
$m : Machine(STATE, EVENT)$
$e : EVENT$
Rewrite Rules

$rew1:\ Get\_next\_act\_state(m)(e)$
rhs1: $e = Init(m) \Rightarrow AP(m)$
rhs2: $e \in Progress(m) \Rightarrow BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]]$
rhs3: $e \notin dom(Get\_next\_act\_state(m)) \Rightarrow \emptyset : \mathbb{P}(STATE)$
END

Listing A.6: Proof rule Definition

THEORY $EvtBTheoryProofRule$
IMPORT THEORY $EvtBPO$
TYPE PARAMETERS $STATE, EVENT$
THEOREMS
$thm1:$
$\forall m, e \cdot m \in Machine(STATE, EVENT) \land e \in Convergent(m) \land$
$e \in Event(m) \land Event\_WellCons(m) \land Tag\_Event\_WellCons(m)$
$\Rightarrow e \in Progress(m)$
PROOF RULES
extension_def:
Metavariables
$m : Machine(STATE, EVENT)$
Rewrite Rules
$rewConsistency:\ check\_Machine\_Consistency(m)$
rhs1: $\top \Rightarrow Mch\_THM(m) \land$
$Mch\_INV(m) \land$
$Mch\_FIS(m) \land$
$Mch\_NAT(m) \land$
$Mch\_VARIANT(m)$
$rewMCH:\ m \in Machine(STATE, EVENT)$
rhs1: $\top \Rightarrow \exists p\_E, p\_S, p\_Ie, p\_P, p\_V, p\_C,$
$p\_O, p\_AP, p\_B, p\_G, p\_In, p\_T \cdot m =$
$Cons\_machine(p\_E, p\_S, p\_Ie, p\_P, p\_AP, p\_G, p\_B,$
$p\_In, p\_T, p\_V, p\_O, p\_C)$
$rewWCons:\ Machine\_WellCons(m)$
rhs1: $\top \Rightarrow BAP\_WellCons(m) \land$
$Grd\_WellCons(m) \land$
$Event\_WellCons(m) \land$
$Tag\_Event\_WellCons(m) \land$
$Variant\_WellCons(m)$
$rewBAPWC:\ BAP\_WellCons(m)$
rhs1: $\top \Rightarrow \forall b, p \cdot Progress(m) = p \land BAP(m) = b \Rightarrow dom(b) = p$
$rewGRDWC:\ Grd\_WellCons(m)$
rhs1: $\top \Rightarrow \forall g, p \cdot Progress(m) = p \land Grd(m) = g \Rightarrow dom(g) = p$
$rewEvtWC:\ Event\_WellCons(m)$
rhs1: $\top \Rightarrow \forall Ev, In, Pro \cdot$
$Init(m) = In \land Event(m) = Ev \land Progress(m) = Pro$
$\Rightarrow partition(Ev, \{In\}, Pro)$
$rewVarWC:\ Variant\_WellCons(m)$

$$
\begin{array}{l}
\quad \mathrm{rhs1}: \ \top \Rightarrow \forall variant, inv \cdot Inv(m) = inv \wedge Variant(m) = variant \\
\qquad \Rightarrow inv \lhd variant \in inv \to \mathbb{Z} \\
rewTagWC: \ Tag\_Event\_WellCons(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow \forall ordinary, init, ev, convergent \cdot \\
\qquad Init(m) = init \wedge Ordinary(m) = ordinary \wedge \\
\qquad Convergent(m) = convergent \wedge Event(m) = ev \\
\qquad\quad \Rightarrow partition(ev, ordinary, convergent) \wedge init \in ordinary \\
rewThm: \ Mch\_THM(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow \\
\qquad \forall inv, thm \cdot Inv(m) = inv \wedge Thm(m) = thm \Rightarrow inv \subseteq thm \\
rewInvInit: \ Mch\_INV\_Init(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow \forall inv, ap \cdot Inv(m) = inv \wedge AP(m) = ap \Rightarrow ap \subseteq inv \\
reFISInit: \ Mch\_FIS\_Init(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow \forall inv, ap \cdot Inv(m) = inv \wedge AP(m) = ap \Rightarrow ap \cap inv \neq \emptyset \\
rewFIS: \ Mch\_FIS(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow Mch\_FIS\_Init(m) \wedge \\
\qquad (\forall e, pro, inv, grd, bap \cdot Progress(m) = pro \wedge e \in pro \wedge \\
\qquad\quad Inv(m) = inv \wedge Grd(m) = grd \wedge BAP(m) = bap \\
\qquad\quad\quad \Rightarrow inv \cap grd[\{e\}] \subseteq dom(bap[\{e\}])) \\
rewInv: \ Mch\_INV(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow Mch\_INV\_Init(m) \wedge \\
\qquad (\forall e, pro, inv, grd, bap \cdot Progress(m) = pro \wedge e \in pro \wedge \\
\qquad\quad Inv(m) = inv \wedge Grd(m) = grd \wedge BAP(m) = bap \\
\qquad\quad\quad \Rightarrow bap[\{e\}][inv \cap grd[\{e\}]] \subseteq inv) \\
rewNat: \ Mch\_NAT(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow \forall variant, inv, grd, convergent \cdot \\
\qquad Variant(m) = variant \wedge Inv(m) = inv \wedge \\
\qquad Convergent(m) = convergent \wedge Grd(m) = grd \\
\qquad\quad \Rightarrow variant[inv \cap grd[convergent]] \subseteq \mathbb{N} \\
rewVar: \ Mch\_VARIANT(m) \\
\quad \mathrm{rhs1}: \ \top \Rightarrow \forall e, s, ev, conv, st, inv, grd, sp, bap, var \cdot \\
\qquad Event(m) = ev \wedge Convergent(m) = conv \wedge Inv(m) = inv \wedge \\
\qquad State(m) = st \wedge Grd(m) = grd \wedge e \in ev \wedge e \in conv \wedge \\
\qquad s \in st \wedge s \in inv \wedge BAP(m) = bap \wedge Variant(m) = var \wedge \\
\qquad s \in grd[\{e\}] \wedge sp \in bap[\{e\}][\{s\}] \\
\qquad\quad \Rightarrow (inv \lhd var)(s) > (inv \lhd var)(sp)
\end{array}
$$

END

A.2 EB4EB Analyses Properties

A.2.1 Core definition of analyses

Listing A.7: Theory of the deadlock freeness analysis

THEORY Theo4DeadlockFree

```
IMPORT THEORY
    EvtBTheory
TYPE PARAMETERS EVENT, STATE
OPERATORS
    DeadlockFreeness_Definition predicate (
        m : Machine(STATE, EVENT))
      direct definition
        Inv(m) ⊆ Grd(m)[Progress(m)]
    check_Machine_DeadLockFreeness predicate (
        m : Machine(STATE, EVENT))
      well−definedness Machine_WellCons(m)
      direct definition
        DeadlockFreeness_Definition(m)
PROOF RULES
    extension_def:
    Metavariables
      m : Machine(STATE, EVENT)
    Rewrite Rules
      rew1 : check_Machine_DeadLockFreeness(m)
        rhs1 : ⊤ ⇒ DeadlockFreeness_Definition(m)
      rew2 : DeadlockFreeness_Definition(m)
        rhs1 : ⊤ ⇒
            ∀g, i, p · Progress(m) = p ∧ Grd(m) = g ∧ Inv(m) = i ⇒ i ⊆ g[p]
END
```

Listing A.8: Theory of the reachability analysis

```
THEORY Theo4Reachability
  IMPORT THEORY
    EvtBManip
  TYPE PARAMETERS EVENT, STATE
  OPERATORS
    At_Least_One_Triggerable_Evt predicate (
        m : Machine(STATE, EVENT), src : EVENT, trgSet : ℙ(EVENT))
      well−definedness trgSet ⊆ Progress(m), src ∈ Event(m),
          Machine_WellCons(m)
      direct definition
        Get_next_act_state(m)(src) ∩ Grd(m)[trgSet] ≠ ∅
    VariantDecrease predicate (m : Machine(STATE, EVENT),
        variant : ℙ(STATE × ℤ), SubSetEvt : ℙ(EVENT))
      well−definedness Inv(m) ◁ variant ∈ Inv(m) → ℤ, Mch_INV(m)
          , BAP_WellCons(m), SubSetEvt ⊆ Progress(m)
      direct definition
        ∀e, s · e ∈ Event(m) ∧ e ∈ SubSetEvt ∧ s ∈ State(m) ∧ s ∈ Inv(m)∧
            s ∈ Grd(m)[{e}] ⇒ (∀sp · sp ∈ BAP(m)[{e}][{s}]
              ⇒ (Inv(m) ◁ variant)(s) > (Inv(m) ◁ variant)(sp))
    NaturalVariant predicate (m : Machine(STATE, EVENT),
```

$variant : \mathbb{P}(STATE \times \mathbb{Z})$, $SubSetEvt : \mathbb{P}(EVENT)$)

well−definedness $Inv(m) \lhd variant \in Inv(m) \to \mathbb{Z}$,
$BAP\_WellCons(m)$, $SubSetEvt \subseteq Progress(m)$

direct definition
$variant[Inv(m) \cap Grd(m)[SubSetEvt]] \subseteq \mathbb{N}$

One_Next_Evt_Is_Triggerable ***predicate*** (
$m : Machine(STATE, EVENT)$, $variant : \mathbb{P}(STATE \times \mathbb{Z})$,
$SubSetEvt : \mathbb{P}(EVENT)$)

well−definedness $Inv(m) \lhd variant \in Inv(m) \to \mathbb{Z}$,
$BAP\_WellCons(m)$, $SubSetEvt \subseteq Progress(m)$, $Mch\_INV(m)$

direct definition
$\forall e, s \cdot e \in SubSetEvt \land$
$s \in BAP(m)[\{e\}][Inv(m) \cap Grd(m)[\{e\}]] \land$
$(Inv(m) \lhd variant)(s) \in \mathbb{N} \Rightarrow$
$s \in Grd(m)[SubSetEvt]$

Evt_Is_Reachable_From_Definition ***predicate*** (
$m : Machine(STATE, EVENT)$, $src : EVENT$, $trg : EVENT$,
$SubSetEvt : \mathbb{P}(EVENT)$, $variant : \mathbb{P}(STATE \times \mathbb{Z})$)

well−definedness $Machine\_WellCons(m)$, $trg \in Progress(m)$,
$src \in Event(m)$, $Inv(m) \lhd variant \in Inv(m) \to \mathbb{Z}$, $Mch\_INV(m)$,
$SubSetEvt \subseteq Progress(m)$

direct definition
$NaturalVariant(m, variant, SubSetEvt) \land$
$VariantDecrease(m, variant, SubSetEvt) \land$
$One\_Next\_Evt\_Is\_Triggerable(m, variant, SubSetEvt) \land$
$At\_Least\_One\_Triggerable\_Evt(m, src, SubSetEvt) \land$
$variant^{-1}[\mathbb{Z} \setminus \mathbb{N}] \cap Inv(m) \subseteq Grd(m)[\{trg\}]$

check_Machine_Evt_Is_Reachable_From ***predicate*** (
$m : Machine(STATE, EVENT)$, $src : EVENT$, $trg : EVENT$,
$SubSetEvt : \mathbb{P}(EVENT)$, $variant : \mathbb{P}(STATE \times \mathbb{Z})$)

well−definedness $Machine\_WellCons(m)$, $trg \in Progress(m)$,
$src \in Event(m)$, $Inv(m) \lhd variant \in Inv(m) \to \mathbb{Z}$, $Mch\_INV(m)$,
$SubSetEvt \subseteq Progress(m)$

direct definition
$Evt\_Is\_Reachable\_From\_Definition(m, src, trg, SubSetEvt, variant)$

PROOF RULES

extension_def:

Metavariables
$m : Machine(STATE, EVENT)$
$src : EVENT$
$trg : EVENT$
$convergent : \mathbb{P}(EVENT)$
$variant : \mathbb{P}(STATE \times \mathbb{Z})$
$trgs : \mathbb{P}(EVENT)$

Rewrite Rules
$rew2 :$ $One\_Next\_Evt\_Is\_Triggerable(m, variant, convergent)$

rhs1 : $\top \Rightarrow \forall e, s, i, g, b \cdot$
$Inv(m) = i \wedge$
$Grd(m) = g \wedge$
$BAP(m) = b \wedge$
$e \in convergent \wedge$
$s \in b[\{e\}][i \cap g[\{e\}]] \wedge$
$(i \lhd variant)(s) \in \mathbb{N} \Rightarrow$
$s \in g[convergent]$

rew3 : $NaturalVariant(m, variant, convergent)$
rhs1 : $\top \Rightarrow \forall i, g \cdot$
$Inv(m) = i \wedge$
$Grd(m) = g \Rightarrow$
$variant[i \cap g[convergent]] \subseteq \mathbb{N}$

rew4 : $VariantDecrease(m, variant, convergent)$
rhs1 : $\top \Rightarrow \forall e, s, i, g, b \cdot$
$Inv(m) = i \wedge$
$Grd(m) = g \wedge$
$BAP(m) = b \wedge$
$e \in Event(m) \wedge$
$e \in convergent \wedge$
$s \in State(m) \wedge$
$s \in Inv(m) \wedge$
$s \in Grd(m)[\{e\}] \Rightarrow$
$(\forall sp \cdot$
$sp \in BAP(m)[\{e\}][\{s\}] \Rightarrow$
$(Inv(m) \lhd variant)(s) > (Inv(m) \lhd variant)(sp))$

rew5 : $At\_Least\_One\_Triggerable\_Evt(m, src, trgs)$
rhs1 : $\top \Rightarrow \forall g \cdot$
$Grd(m) = g \Rightarrow$
$Get\_next\_act\_state(m)(src) \cap g[trgs] \neq \emptyset$

rew6 :
$Evt\_Is\_Reachable\_From\_Definition(m, src, trg, convergent, variant)$

rhs1 : $\top \Rightarrow \forall i, g \cdot$
$Inv(m) = i \wedge Grd(m) = g \Rightarrow$
$NaturalVariant(m, variant, convergent) \wedge$
$VariantDecrease(m, variant, convergent) \wedge$
$One\_Next\_Evt\_Is\_Triggerable(m, variant, convergent) \wedge$
$At\_Least\_One\_Triggerable\_Evt(m, src, convergent) \wedge$
$variant^{-1}[\mathbb{Z} \setminus \mathbb{N}] \cap i \subseteq g[\{trg\}]$

rew7 : $check\_Machine\_Evt\_Is\_Reachable\_From($
$m, src, trg, convergent, variant)$
rhs1 : $\top \Rightarrow Evt\_Is\_Reachable\_From\_Definition($
$m, src, trg, convergent, variant)$

Listing A.9: Theory of the weak invariant analysis

```
THEORY Theo4WeakInv
  IMPORT THEORY
    EvtBTheory
  TYPE PARAMETERS EVENT, STATE
  OPERATORS
    AllowedMachineHoleSub_Definition predicate (
        m : Machine(STATE, EVENT), nGrd : ℙ(STATE),
        nBAP : ℙ(STATE × STATE))
      direct definition
        nBAP[Inv(m) ∩ nGrd] ⊆ Inv(m)
    check_Machine_AllowedMachineHoleSub predicate (
        m : Machine(STATE, EVENT), nGrd : ℙ(STATE),
        nBAP : ℙ(STATE × STATE))
      well-definedness Machine_WellCons(m)
      direct definition
        AllowedMachineHoleSub_Definition(m, nGrd, nBAP)
  PROOF RULES
    extension_def:
    Metavariables
      m : Machine(STATE, EVENT)
      nGrd : ℙ(STATE)
      nBAP : ℙ(STATE × STATE)
    Rewrite Rules
      rew1 : AllowedMachineHoleSub_Definition(m, nGrd, nBAP)
        rhs1 : ⊤ ⇒ ∀i · i = Inv(m) ⇒ nBAP[i ∩ nGrd] ⊆ i
      rew2 : check_Machine_AllowedMachineHoleSub(m, nGrd, nBAP)
        rhs1 : ⊤ ⇒ AllowedMachineHoleSub_Definition(m, nGrd, nBAP)
END
```

A.3 EB4EB Liveness Properties

A.3.1 Core definition of temporal properties

Listing A.10: Theory of temporal propreties

```
THEORY Theo4Liveness
  IMPORT THEORY EvtBPO
  TYPE PARAMETERS EVENT, STATE
  OPERATORS
    LeadsP1ToP2OneEvt predicate (m : Machine(STATE, EVENT),
        p1 : ℙ(STATE), p2 : ℙ(STATE), e : EVENT)
      well-definedness e ∈ Progress(m)
      direct definition
        BAP(m)[{e}][p1 ∩ Grd(m)[{e}] ∩ Inv(m)] ⊆ p2
```

LeadsP1ToP2 ***predicate*** $(m : Machine(STATE, EVENT)\,,$
$p1 : \mathbb{P}(STATE)\,, p2 : \mathbb{P}(STATE))$
direct definition
$\forall e \cdot e \in Progress(m) \Rightarrow LeadsP1ToP2OneEvt(m, p1, p2, e)$

ConvPOneEvt ***predicate*** $(m : Machine(STATE, EVENT)\,,$
$p : \mathbb{P}(STATE)\,, variant : \mathbb{P}(STATE \times \mathbb{Z})\,, e : EVENT)$
well−definedness $e \in Progress(m)\,, variant \in STATE \to \mathbb{Z}$
direct definition
$variant[Inv(m) \cap p \cap Grd(m)[\{e\}]] \subseteq \mathbb{N} \land$
$(\forall s, sp \cdot s \in Inv(m) \land s \in p \cap Grd(m)[\{e\}] \land sp \in BAP(m)[\{e\}][\{s\}]$
$\Rightarrow variant(s) > variant(sp))$

ConvP ***predicate*** $(m : Machine(STATE, EVENT)\,, p : \mathbb{P}(STATE)\,,$
$variant : \mathbb{P}(STATE \times \mathbb{Z}))$
well−definedness $variant \in STATE \to \mathbb{Z}$
direct definition
$\forall e \cdot e \in Progress(m) \Rightarrow ConvPOneEvt(m, p, variant, e)$

DiverPOneEvt ***predicate*** $(m : Machine(STATE, EVENT)\,,$
$p : \mathbb{P}(STATE)\,, variant : \mathbb{P}(STATE \times \mathbb{Z})\,, e : EVENT)$
well−definedness $e \in Progress(m)\,, variant \in STATE \to \mathbb{Z}$
direct definition
$ConvPOneEvt(m, STATE \setminus p, variant, e) \land$
$(\forall s, sp \cdot s \in Inv(m) \land s \in p \cap Grd(m)[\{e\}] \land$
$sp \in BAP(m)[\{e\}][\{s\}] \land variant(sp) \in \mathbb{N}$
$\Rightarrow variant(s) \geq variant(sp))$

DiverP ***predicate*** $(m : Machine(STATE, EVENT)\,, p : \mathbb{P}(STATE)\,,$
$variant : \mathbb{P}(STATE \times \mathbb{Z}))$
well−definedness $variant \in STATE \to \mathbb{Z}$
direct definition
$\forall e \cdot e \in Progress(m) \Rightarrow DiverPOneEvt(m, p, variant, e)$

DeadlockFreeP ***predicate*** $(m : Machine(STATE, EVENT)\,,$
$p : \mathbb{P}(STATE))$
direct definition
$p \cap Inv(m) \subseteq Grd(m)[Progress(m)]$

Globaly ***predicate*** $(m : Machine(STATE, EVENT)\,, I : \mathbb{P}(STATE))$
direct definition
$Inv(m) \subseteq I$

ExistenceP ***predicate*** $(m : Machine(STATE, EVENT)\,,$
$p : \mathbb{P}(STATE)\,, variant : \mathbb{P}(STATE \times \mathbb{Z}))$
well−definedness $variant \in STATE \to \mathbb{Z}$
direct definition
$ConvP(m, STATE \setminus p, variant) \land$
$DeadlockFreeP(m, STATE \setminus p)$

P1UntilP2 ***predicate*** $(m : Machine(STATE, EVENT)\,,$
$variant : \mathbb{P}(STATE \times \mathbb{Z})\,, p1 : \mathbb{P}(STATE)\,, p2 : \mathbb{P}(STATE))$
well−definedness $variant \in STATE \to \mathbb{Z}$
direct definition

$LeadsP1ToP2(m, p1 \cap (STATE \setminus p2), p1 \cup p2) \wedge$
$ExistenceP(m, (STATE \setminus p1) \cup p2, variant)$

P1ProgressByP3ToP2 ***predicate*** $(m : Machine(STATE, EVENT)$,
$variant : \mathbb{P}(STATE \times \mathbb{Z})$, $p1 : \mathbb{P}(STATE)$, $p2 : \mathbb{P}(STATE)$,
$p3 : \mathbb{P}(STATE))$
well−definedness $variant \in STATE \rightarrow \mathbb{Z}$
direct definition
$Globaly(m, p3 \cup p2 \cup (STATE \setminus p1)) \wedge$
$P1UntilP2(m, variant, p3, p2)$

PersistenceP ***predicate*** $(m : Machine(STATE, EVENT)$,
$p : \mathbb{P}(STATE)$, $variant : \mathbb{P}(STATE \times \mathbb{Z}))$
well−definedness $variant \in STATE \rightarrow \mathbb{Z}$
direct definition
$DiverP(m, p, variant) \wedge$
$DeadlockFreeP(m, STATE \setminus p)$

END

A.3.2 Helper Theory

Listing A.11: Proof rules for temporal properties

THEORY *Theo4LivenessProofRules*
IMPORT THEORY *Theo4Liveness*
TYPE PARAMETERS $EVENT, STATE$
PROOF RULES
extension_liveness_def:
Metavariables
$m : Machine(STATE, EVENT)$
$p1 : \mathbb{P}(STATE)$
$p2 : \mathbb{P}(STATE)$
$p3 : \mathbb{P}(STATE)$
$v : \mathbb{P}(STATE \times \mathbb{Z})$
Rewrite Rules
rew1 : $LeadsP1ToP2(m, p1, p2)$
rhs1 : $\top \Rightarrow \forall bap, g, i, p \cdot bap = BAP(m) \wedge g = Grd(m) \wedge$
$i = Inv(m) \wedge p = Progress(m)$
$\Rightarrow (\forall e \cdot e \in p$
$\Rightarrow bap[\{e\}][p1 \cap g[\{e\}] \cap i] \subseteq p2)$
rew2 : $ConvP(m, p1, v)$
rhs1 : $\top \Rightarrow \forall bap, g, i, p \cdot bap = BAP(m) \wedge g = Grd(m) \wedge$
$i = Inv(m) \wedge p = Progress(m)$
$\Rightarrow (\forall e \cdot e \in p$
$\Rightarrow (v[Inv(m) \cap p1 \cap Grd(m)[\{e\}]] \subseteq \mathbb{N} \wedge$
$(\forall s, sp \cdot s \in i \wedge s \in p1 \cap g[\{e\}] \wedge sp \in bap[\{e\}][\{s\}]$
$\Rightarrow v(s) > v(sp))))$
rew3 : $DiverP(m, p1, v)$

$$
\begin{array}{l}
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \ \forall bap, g, i, p \cdot bap = BAP(m) \wedge g = Grd(m) \\
\quad \wedge i = Inv(m) \wedge p = Progress(m) \\
\qquad \Rightarrow (\forall e \cdot e \in p \\
\qquad\quad \Rightarrow (v[Inv(m) \cap (STATE \setminus p1) \cap Grd(m)[\{e\}]] \subseteq \mathbb{N} \wedge \\
\qquad\quad (\forall s, sp \cdot s \in i \wedge \\
\qquad\qquad s \in (STATE \setminus p1) \cap g[\{e\}] \wedge sp \in bap[\{e\}][\{s\}] \\
\qquad\qquad\quad \Rightarrow v(s) > v(sp)) \wedge \\
\qquad\quad (\forall s, sp \cdot s \in i \wedge \\
\qquad\qquad s \in p1 \cap g[\{e\}] \wedge sp \in bap[\{e\}][\{s\}] \wedge v(sp) \in \mathbb{N} \\
\qquad\qquad\quad \Rightarrow v(s) \geq v(sp))))
\end{array}
$$

$$
\begin{array}{l}
rew4: \ DeadlockFreeP(m, p1) \\
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \ \forall g, i, p \cdot g = Grd(m) \wedge i = Inv(m) \wedge p = Progress(m) \\
\qquad \Rightarrow (p1 \cap i \subseteq g[Progress(m)]) \\
rew5: \ Globaly(m, p1) \\
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \ \forall i \cdot i = Inv(m) \Rightarrow (i \subseteq p1) \\
rew6: \ ExistenceP(m, p1, v) \\
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \\
\qquad ConvP(m, STATE \setminus p1, v) \wedge DeadlockFreeP(m, STATE \setminus p1) \\
rew7: \ P1UntilP2(m, v, p1, p2) \\
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \ LeadsP1ToP2(m, p1 \cap (STATE \setminus p2), p1 \cup p2) \wedge \\
\quad ExistenceP(m, (STATE \setminus p1) \cup p2, v) \\
rew8: \ P1ProgressByP3ToP2(m, v, p1, p2, p3) \\
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \\
\qquad Globaly(m, p3 \cup p2 \cup (STATE \setminus p1)) \wedge P1UntilP2(m, v, p3, p2) \\
rew9: \ PersistenceP(m, p1, v) \\
\quad \mathrm{rhs1}: \ \top \ \Rightarrow \ DiverP(m, p1, v) \wedge DeadlockFreeP(m, STATE \setminus p1)
\end{array}
$$

END

A.4 Domain specific Analysis

A.4.1 Ontologies Theory

Listing A.12: Ontologies theory

THEORY *OntologiesTheory*
TYPE PARAMETERS C, P, I
DATA TYPES
Ontology(C, P, I)
CONSTRUCTORS
consOntology(
$classes : \mathbb{P}(C)$,
$properties : \mathbb{P}(P)$,
$instances : \mathbb{P}(I)$,
$classProperties : \mathbb{P}(C \times P)$,

$classInstances : \mathbb{P}(C \times I)$,
$classAssociations : \mathbb{P}(C \times P \times C)$,
$instanceAssociations : \mathbb{P}(I \times P \times I))$

OPERATORS

getClasses <***expression***> $(o : Ontology(C, P, I))$
direct definition
$classes(o)$

getProperties <***expression***> $(o : Ontology(C, P, I))$
direct definition
$properties(o)$

getInstances <***expression***> $(o : Ontology(C, P, I))$
direct definition
$instances(o)$

isWDClassProperites <***predicate***> $(o : Ontology(C, P, I))$
direct definition
$classProperties(o) \in getClasses(o) \leftrightarrow getProperties(o)$

getClassProperties <***expression***> $(o : Ontology(C, P, I))$
well−definedness isWDClassProperites(o)
direct definition
$classProperties(o)$

isWDClassInstances <***predicate***> $(o : Ontology(C, P, I))$
direct definition
$classInstances(o) \in getClasses(o) \leftrightarrow getInstances(o)$

getClassInstances <***expression***> $(o : Ontology(C, P, I))$
well−definedness isWDClassInstances(o)
direct definition
$classInstances(o)$

isWDClassAssociations <***predicate***> $(o : Ontology(C, P, I))$
well−definedness $isWDClassProperites(o)$
direct definition
$classAssociations(o) \in getClassProperties(o) \rightarrow classes(o)$

getClassAssociations <***expression***> $(o : Ontology(C, P, I))$
well−definedness $isWDClassAssociations(o)$
direct definition
$classAssociations(o)$

isWDInstancesAssociations <***predicate***> $(o : Ontology(C, P, I))$
well−definedness $isWDClassProperites(o)$,
$isWDClassInstances(o)$, $isWDClassAssociations(o)$
direct definition
$instanceAssociations(o) \subseteq$
$instances(o) \times properties(o) \times instances(o) \wedge$
$instanceAssociations(o) \subseteq \{i1 \mapsto p \mapsto i2 \mid i1 \in I \wedge p \in P \wedge i2 \in I \wedge$
$i1 \mapsto p \mapsto i2 \in instances(o) \times properties(o) \times instances(o) \wedge$
$(\exists c1, c2 \cdot c1 \in C \wedge c2 \in C \wedge \{c1, c2\} \subseteq getClasses(o) \Rightarrow$
$(c1 \mapsto p \mapsto c2 \in getClassAssociations(o) \wedge$
$p \in getClassProperties(o)[\{c1\}] \wedge$

$i1 \in getClassInstances(o)[\{c1\}] \wedge$
$i2 \in getClassInstances(o)[\{c2\}]))$
}

getInstanceAssociations <***expression***> $(o : Ontology(C, P, I))$
well−definedness $isWDInstancesAssociations(o)$
direct definition
$instanceAssociations(o)$

isWDOntology <***predicate***> $(o : Ontology(C, P, I))$
direct definition
$isWDClassProperites(o) \wedge$
$isWDClassInstances(o) \wedge$
$isWDClassAssociations(o) \wedge$
$isWDInstancesAssociations(o)$

isItWDOntology <***predicate***> $(o : Ontology(C, P, I))$
well−definedness $isWDOntology(o)$
direct definition
$\top$

ontologyContainsClasses <***predicate***> $(o : Ontology(C, P, I),$
$cc : \mathbb{P}(C))$
well−definedness $isWDOntology(o), cc \neq \emptyset$
direct definition
$cc \subseteq getClasses(o)$

ontologyContainsProperties <***predicate***> $(o : Ontology(C, P, I),$
$pp : \mathbb{P}(P))$
well−definedness $isWDOntology(o), pp \neq \emptyset$
direct definition
$pp \subseteq getProperties(o)$

ontologyContainsInstances <***predicate***> $(o : Ontology(C, P, I),$
$ii : \mathbb{P}(I))$
well−definedness $isWDOntology(o), ii \neq \emptyset$
direct definition
$ii \subseteq getInstances(o)$

getPropertiesOfaClass <***expression***> $(o : Ontology(C, P, I), c : C)$
well−definedness $isWDOntology(o),$
$ontologyContainsClasses(o, \{c\})$
direct definition
$getClassProperties(o)[\{c\}]$

getInstancesOfaClass <***expression***> $(o : Ontology(C, P, I), c : C)$
well−definedness $isWDOntology(o),$
$ontologyContainsClasses(o, \{c\})$
direct definition
$getClassInstances(o)[\{c\}]$

getInstancesOfClasses <***expression***> $(o : Ontology(C, P, I),$
$cc : \mathbb{P}(C))$
well−definedness $isWDOntology(o),$
$ontologyContainsClasses(o, cc)$

direct definition
$getClassInstances(o)[cc]$
isPropertyOfTheClass <***predicate***> $(o : Ontology(C, P, I), c : C, p : P)$
well−definedness $isWDOntology(o)$,
$ontologyContainsClasses(o, \{c\})$,
$ontologyContainsProperties(o, \{p\})$
direct definition
$p \in getPropertiesOfaClass(o, c)$
getPropertyRangeClasses <***expression***> $(o : Ontology(C, P, I), p : P)$
well−definedness $isWDOntology(o)$,
$ontologyContainsProperties(o, \{p\})$
direct definition
$\{c2 \mid \forall c1 \cdot c1 \in getClasses(o) \Rightarrow c1 \mapsto p \mapsto c2 \in getClassAssociations(o)\}$
getPropertyRangeInstances <***expression***> $(o : Ontology(C, P, I), p : P)$
well−definedness $isWDOntology(o)$,
$ontologyContainsProperties(o, \{p\})$
direct definition
$\{j \mid \forall i \cdot i \in getInstances(o) \Rightarrow i \mapsto p \mapsto j \in getInstanceAssociations(o)\}$
getValueOfAInstanceProperty <***expression***> $(o : Ontology(C, P, I), i : I, p : P)$
well−definedness $isWDOntology(o)$,
$ontologyContainsProperties(o, \{p\})$,
$ontologyContainsInstances(o, \{i\})$
direct definition
$getInstanceAssociations(o)[\{i \mapsto p\}]$
getClassesOfInstance <***expression***> $(o : Ontology(C, P, I), i : I)$
well−definedness $isWDOntology(o)$,
$ontologyContainsInstances(o, \{i\})$
direct definition
$getClassInstances(o)^{-1}[\{i\}]$
classContainsProperties <***predicate***> $(o : Ontology(C, P, I), c : C, pp : \mathbb{P}(P))$
well−definedness $isWDOntology(o)$,
$ontologyContainsClasses(o, \{c\})$,
$ontologyContainsProperties(o, pp)$
direct definition
$pp \subseteq getPropertiesOfaClass(o, c)$
classContainsInstances <***predicate***> $(o : Ontology(C, P, I), c : C, ii : \mathbb{P}(I))$
well−definedness $isWDOntology(o)$,
$ontologyContainsClasses(o, \{c\})$,

$ontologyContainsInstances(o, ii)$

direct definition

$ii \subseteq getInstancesOfaClass(o, c)$

instanceHasPropertyValue <***predicate***> $(o : Ontology(C, P, I)\,,$
$i : I\,, p : P\,, v : I)$

well−definedness $isWDOntology(o)\,,$
$ontologyContainsInstances(o, \{i, v\})\,,$
$ontologyContainsProperties(o, \{p\})$

direct definition

$v \in getValueOfAInstanceProperty(o, i, p)$

isA <***predicate***> $(o : Ontology(C, P, I)\,, c1 : C\,, c2 : C)$

well−definedness $isWDOntology(o)\,,$
$ontologyContainsClasses(o, \{c1, c2\})$

direct definition

$getInstancesOfaClass(o, c1) \subseteq getInstancesOfaClass(o, c2)$

addInstancesToOntology <***expression***> $(o : Ontology(C, P, I)\,,$
$ii : \mathbb{P}(I))$

well−definedness $isWDOntology(o)\,,$
$\neg ontologyContainsInstances(o, ii)$

direct definition

$consOntology($
$getClasses(o),$
$getProperties(o),$
$getInstances(o) \cup ii,$
$getClassProperties(o),$
$getClassInstances(o),$
$getClassAssociations(o),$
$getInstanceAssociations(o))$

addInstancesToAClass <***expression***> $(o : Ontology(C, P, I)\,, c : C\,,$
$ii : \mathbb{P}(I))$

well−definedness $isWDOntology(o)\,,$
$ontologyContainsClasses(o, \{c\})\,,$
$ontologyContainsInstances(o, ii)\,,$
$\neg classContainsInstances(o, c, ii)$

direct definition

$consOntology($
$getClasses(o),$
$getProperties(o),$
$getInstances(o),$
$getClassProperties(o),$
$getClassInstances(o) \cup (\{c\} \times ii),$
$getClassAssociations(o),$
$getInstanceAssociations(o))$

ontologyContainsIpv <***predicate***> $(o : Ontology(C, P, I)\,,$
$ipvs : \mathbb{P}(I \times P \times I))$

well−definedness $isWDOntology(o)$

direct definition
$ipvs \subseteq getInstanceAssociations(o)$
isWDAddValueOfAInstanceProperty <***predicate***> (
$o : Ontology(C, P, I)$, $ipvs : \mathbb{P}(I \times P \times I)$, $i : I$, $p : P$, $v : I$)
direct definition
$\top$
addValueOfAInstanceProperty <***expression***> (
$o : Ontology(C, P, I)$, $ipvs : \mathbb{P}(I \times P \times I)$, $i : I$, $p : P$, $v : I$)
well−definedness
$isWDAddValueOfAInstanceProperty(o, ipvs, i, p, v)$
direct definition
$ipvs \cup \{i \mapsto p \mapsto v\}$
isWDRemoveValueOfAInstanceProperty <***predicate***> (
$o : Ontology(C, P, I)$, $ipvs : \mathbb{P}(I \times P \times I)$, $i : I$, $p : P$, $v : I$)
direct definition
$\top$
removeValueOfAInstanceProperty <***expression***> (
$o : Ontology(C, P, I)$, $ipvs : \mathbb{P}(I \times P \times I)$, $i : I$, $p : P$, $v : I$)
well−definedness
$isWDRemoveValueOfAInstanceProperty(o, ipvs, i, p, v)$
direct definition
$ipvs \setminus \{i \mapsto p \mapsto v\}$
isVariableOfOntology <***predicate***> ($o : Ontology(C, P, I)$,
$ipvs : \mathbb{P}(I \times P \times I)$)
well−definedness $isWDOntology(o)$
direct definition
$ipvs \subseteq getInstanceAssociations(o)$
isWDInstanceHasPropertyValuei <***predicate***> (
$o : Ontology(C, P, I)$, $ipvs : \mathbb{P}(I \times P \times I)$, $i : I$, $p : P$, $v : I$)
direct definition
$isWDOntology(o) \wedge$
$isVariableOfOntology(o, ipvs) \wedge$
$ontologyContainsInstances(o, \{i, v\}) \wedge$
$ontologyContainsProperties(o, \{p\})$
instanceHasPropertyValuei <***predicate***> ($o : Ontology(C, P, I)$,
$ipvs : \mathbb{P}(I \times P \times I)$, $i : I$, $p : P$, $v : I$)
well−definedness
$isWDInstanceHasPropertyValuei(o, ipvs, i, p, v)$
direct definition
$v \in ipvs[\{i \mapsto p\}]$

THEOREMS

thm1 :
$\forall o, c1, c2, c3 \cdot o \in Ontology(C, P, I) \wedge isWDOntology(o) \wedge$
$c1 \in C \wedge c2 \in C \wedge c3 \in C \wedge$
$ontologyContainsClasses(o, \{c1, c2, c3\})$
$\Rightarrow (isA(o, c1, c2) \wedge isA(o, c2, c3) \Rightarrow isA(o, c1, c3))$

$thm2:$
$\forall o, cs1, cs2 \cdot o \in Ontology(C, P, I) \land isWDOntology(o) \land$
$cs1 \subseteq C \land cs2 \subseteq C \land cs1 \neq \emptyset \land cs2 \neq \emptyset \land$
$ontologyContainsClasses(o, cs1) \land ontologyContainsClasses(o, cs2)$
$\Rightarrow (ontologyContainsClasses(o, cs1 \cup cs2))$

END

A.4.2 Behavioural theory of domain specific property

Listing A.13: Theory of behavioural analyses derived from a ontology

THEORY *BehaviouralPropertiesTheory*
IMPORT THEORY
Theo4Reachability
OntologiesTheory
TYPE PARAMETERS $STATEE$, $EVENTT$, $Tags$, Ps
OPERATORS
isPossiblyFollowedByWD <***predicate***> (
$m : Machine(STATEE, EVENTT)$,
$eo : Ontology(Tags, Ps, EVENTT)$, $startTags : \mathbb{P}(Tags)$,
$transitTags : \mathbb{P}(Tags)$, $endTags : \mathbb{P}(Tags)$,
$variants : \mathbb{P}(EVENTT \times \mathbb{P}(STATEE \times \mathbb{Z}))$)
direct definition
$isWDOntology(eo) \land$
$startTags \cup transitTags \cup endTags \subseteq getClasses(eo) \land$
$startTags \cap transitTags = \emptyset \land$
$endTags \cap transitTags = \emptyset \land$
$startTags \neq \emptyset \land$
$endTags \neq \emptyset \land$
$transitTags \neq \emptyset \land$
$(\forall ti \cdot ti \in startTags \cup transitTags \cup endTags$
$\Rightarrow getInstancesOfClasses(eo, \{ti\}) \neq \emptyset) \land$
$variants \in$
$getInstancesOfClasses(eo, startTags) \rightarrow \mathbb{P}(STATEE \times \mathbb{Z}) \land$
$(\forall i \cdot i \in getInstancesOfClasses(eo, startTags)$
$\Rightarrow WD\_reach(m,$
$i,$
$getInstancesOfClasses(eo, endTags),$
$getInstancesOfClasses(eo, transitTags),$
$variants(i)))$
isPossiblyFollowedBy <***predicate***> (
$m : Machine(STATEE, EVENTT)$,
$eo : Ontology(Tags, Ps, EVENTT)$, $startTags : \mathbb{P}(Tags)$,
$transitTags : \mathbb{P}(Tags)$, $endTags : \mathbb{P}(Tags)$,
$variants : \mathbb{P}(EVENTT \times \mathbb{P}(STATEE \times \mathbb{Z}))$)
well−definedness $isPossiblyFollowedByWD($

$m, eo, startTags, transitTags, endTags, variants)$

direct definition

$\forall i \cdot i \in getInstancesOfClasses(eo, startTags)$
$\Rightarrow Evt\_Is\_Sometimes\_Reachable\_From\_Definition($
$m,$
$i,$
$getInstancesOfClasses(eo, endTags),$
$getInstancesOfClasses(eo, transitTags),$
$variants(i))$

isNecessarilyFollowedByWD <***predicate***> (
$m : Machine(STATEE, EVENTT)$,
$eo : Ontology(Tags, Ps, EVENTT)$, $startTags : \mathbb{P}(Tags)$,
$transitTags : \mathbb{P}(Tags)$, $endTags : \mathbb{P}(Tags)$,
$variants : \mathbb{P}(EVENTT \times \mathbb{P}(STATEE \times \mathbb{Z}))$)

direct definition

$isWDOntology(eo) \wedge$
$startTags \cup transitTags \cup endTags \subseteq getClasses(eo) \wedge$
$startTags \cap transitTags = \emptyset \wedge$
$endTags \cap transitTags = \emptyset \wedge$
$startTags \neq \emptyset \wedge$
$endTags \neq \emptyset \wedge$
$transitTags \neq \emptyset \wedge$
$(\forall ti \cdot ti \in startTags \cup transitTags \cup endTags$
$\Rightarrow getInstancesOfClasses(eo, \{ti\}) \neq \emptyset) \wedge$
$variants \in$
$getInstancesOfClasses(eo, startTags) \rightarrow \mathbb{P}(STATEE \times \mathbb{Z}) \wedge$
$(\forall i \cdot i \in getInstancesOfClasses(eo, startTags)$
$\Rightarrow WD\_reach(m,$
$i,$
$getInstancesOfClasses(eo, endTags),$
$getInstancesOfClasses(eo, transitTags),$
$variants(i)))$

isNecessarilyFollowedBy <***predicate***> (
$m : Machine(STATEE, EVENTT)$,
$eo : Ontology(Tags, Ps, EVENTT)$,
$startTags : \mathbb{P}(Tags)$,
$transitTags : \mathbb{P}(Tags)$,
$endTags : \mathbb{P}(Tags)$,
$variants : \mathbb{P}(EVENTT \times \mathbb{P}(STATEE \times \mathbb{Z}))$)

well—definedness $isNecessarilyFollowedByWD($
$m, eo, startTags, transitTags, endTags, variants)$

direct definition

$\forall i \cdot i \in getInstancesOfClasses(eo, startTags)$
$\Rightarrow Evt\_Is\_Always\_Reachable\_From\_Definition($
$m,$
$i,$

$$getInstancesOfClasses(eo, endTags),$$
$$getInstancesOfClasses(eo, transitTags),$$
$$variants(i))$$

THEOREMS

WDNisWDP :

$$\forall m, eo, startTags, transitTags, endTags, variants \cdot$$
$$(m \in Machine(STATEE, EVENTT) \wedge$$
$$eo \in Ontology(Tags, Ps, EVENTT) \wedge$$
$$startTags \cup transitTags \cup endTags \in \mathbb{P}(Tags) \wedge$$
$$variants \in \mathbb{P}(EVENTT \times \mathbb{P}(STATEE \times \mathbb{Z})))$$
$$\Rightarrow (isNecessarilyFollowedByWD($$
$$m, eo, startTags, transitTags, endTags, variants)$$
$$\Rightarrow isPossiblyFollowedByWD($$
$$m, eo, startTags, transitTags, endTags, variants))$$

NisP :

$$\forall m, eo, startTags, transitTags, endTags, variants \cdot$$
$$(m \in Machine(STATEE, EVENTT) \wedge$$
$$eo \in Ontology(Tags, Ps, EVENTT) \wedge$$
$$startTags \cup transitTags \cup endTags \in \mathbb{P}(Tags) \wedge$$
$$variants \in \mathbb{P}(EVENTT \times \mathbb{P}(STATEE \times \mathbb{Z})))$$
$$\Rightarrow (isNecessarilyFollowedByWD($$
$$m, eo, startTags, transitTags, endTags, variants) \wedge$$
$$isNecessarilyFollowedBy($$
$$m, eo, startTags, transitTags, endTags, variants)$$
$$\Rightarrow isPossiblyFollowedBy($$
$$m, eo, startTags, transitTags, endTags, variants))$$

END

A.5 Correctness

A.5.1 Peano theory

Listing A.14: Theory of Peano

```
THEORY Natural
  DATA TYPES
    iNAT
    CONSTRUCTORS
      iZero()
      iSucc(iPrec : iNAT)
  OPERATORS
    mk_int expression (n : iNAT)
      recursive definition
        case n:
          iZero =>
```

```
                0
            iSucc(m) =>
              1 + mk_int(m)
  AXIOMATIC DEFINITIONS  xdb1:
    OPERATORS
      mk_iNAT expression (x : ℤ) : iNAT
        well−definedness x ∈ ℕ
    AXIOMS
      axm1: mk_iNAT(0) = iZero
      axm2: ∀x · x ∈ ℕ ⇒ mk_iNAT(1 + x) = iSucc(mk_iNAT(x))
      axm3: ∀x · x ∈ ℕ ⇒ mk_int(mk_iNAT(x)) = x
  THEOREMS
    thm1: ∀n · n ∈ iNAT ⇒ mk_int(n) ∈ ℕ
    thm2: ∀n · n ∈ iNAT ⇒ mk_iNAT(mk_int(n)) = n
    thm3: ∀n · n ∈ ℕ ⇒ (∃ni · ni ∈ iNAT ∧ mk_iNAT(n) = ni)
    thm4: ∀x · x ∈ ℕ1 ⇒ (∃ni · mk_iNAT(x) = iSucc(ni))
END
```

Listing A.15: Basic operator for peano arithmetic

```
THEORY NaturalOp
  IMPORT THEORY Natural
  OPERATORS
    add associative commutative expression (n1 : iNAT , n2 : iNAT)
      direct definition
        mk_iNAT(mk_int(n1) + mk_int(n2))
    miniNAT predicate (set : ℙ(iNAT))
      well−definedness set ≠ ∅
      direct definition
        ∃i · i ∈ set ∧ (∀j · j ∈ set ⇒ mk_int(i) ≤ mk_int(j))
  THEOREMS
    succ add left:
      ∀n1, n2 · n1 addiSucc(n2) = iSucc(n1 addn2)
    succ add right:
      ∀n1, n2 · iSucc(n1)addn2 = iSucc(n1 addn2)
    thm3:
      ∀set · set ≠ ∅ ⇒ miniNAT(set)
  PROOF RULES
    rulesBlock1:
    Metavariables
      n1 : iNAT
      n2 : iNAT
    Rewrite Rules
      rew1: n1 addiSucc(n2)
        rhs1: ⊤ ⇒ iSucc(n1 addn2)
      rew2: iSucc(n1)addn2
        rhs1: ⊤ ⇒ iSucc(n1 addn2)
```

Inference Rules

$inf1: \quad n1 \in iNAT, n2 \in iNAT \vdash mk\_int(n1) + mk\_int(n2) \in \mathbb{N}$

$inf2: \quad n1 \in iNAT, n2 \in iNAT \vdash mk\_int(n1\,addn2) \geq mk\_int(n1)$

$inf3: \quad n1 \in iNAT, n2 \in iNAT \vdash mk\_int(n1\,addn2) \geq mk\_int(n2)$

END

A.5.2 Event-B traces formalism

Listing A.16: Theory of Event-B traces

THEORY *Traces*

IMPORT THEORY *EvtBStruc*, *Natural*

TYPE PARAMETERS *STATE*, *EVENT*

OPERATORS

IsANextState ***predicate*** $(m : Machine(STATE, EVENT),$
$s : STATE, sp : STATE)$

direct definition

$\exists e \cdot e \in Progress(m) \wedge s \in Grd(m)[\{e\}] \wedge s \mapsto sp \in BAP(m)[\{e\}]$

IsATrace ***predicate*** $(m : Machine(STATE, EVENT),$
$tr : \mathbb{P}(iNAT \times STATE))$

direct definition

$($
$tr \in iNAT \rightarrow STATE \vee$
$(\exists n \cdot n \in iNAT \wedge$
$tr \in \{i \mid mk\_int(i) \in 0..mk\_int(n)\} \rightarrow STATE \wedge$
$tr(n) \notin Grd(m)[Progress(m)])$
$) \wedge$
$tr(iZero) \in AP(m) \wedge$
$(\forall i, j \cdot i \in dom(tr) \wedge j \in dom(tr) \wedge j = iSucc(i)$
$\Rightarrow IsANextState(m, tr(i), tr(j)))$

THEOREMS

tr is a partial fun:
$\forall m, tr \cdot m \in Machine(STATE, EVENT) \wedge IsATrace(m, tr)$
$\Rightarrow tr \in iNAT \nrightarrow STATE$

if succ(n) in tr then n in tr:
$\forall i, j, tr, m \cdot m \in Machine(STATE, EVENT) \wedge IsATrace(m, tr) \wedge$
$j = iSucc(i) \wedge j \in dom(tr)$
$\Rightarrow i \in dom(tr)$

PROOF RULES

type_rules:

Metavariables

$m : Machine(STATE, EVENT)$
$tr : \mathbb{P}(iNAT \times STATE)$
$i : iNAT$
$j : iNAT$

Inference Rules

$inf1:\ IsATrace(m, tr) \vdash tr \in iNAT \nrightarrow STATE$
$inf2:\ IsATrace(m, tr), j \in dom(tr), j = iSucc(i) \vdash i \in dom(tr)$
END

A.5.3 Soundness of Invariant Proof Obligation

Listing A.17: Soundness theorem of the Invariant proof obligations

THEORY $InvCorrectness$
IMPORT THEORY $Traces, EvtBPO$
TYPE PARAMETERS $STATE, EVENT$
THEOREMS
thm1:
$\forall m, tr \cdot m \in Machine(STATE, EVENT) \land Machine\_WellCons(m) \land$
$IsATrace(m, tr) \land$
$Mch\_INV(m)$
$\Rightarrow (\forall n \cdot n \in dom(tr) \Rightarrow tr(n) \in Inv(m))$
END

A.5.4 Soundness of Temporal Properties

Listing A.18: Soundness theorems for all temporal properties

THEORY $LivenessCorrectness$
IMPORT THEORY $InvCorrectness, Theo4Liveness, NaturalOp$
TYPE PARAMETERS $STATE, EVENT$
THEOREMS
thm of correctness of Leads_From_P1_To_P2:
$\forall m, tr, p1, p2 \cdot m \in Machine(STATE, EVENT) \land$
$Machine\_WellCons(m) \land$
$check\_Machine\_Consistency(m) \land$
$IsATrace(m, tr) \land$
$TLLeads\_From\_P1\_To\_P2(m, p1, p2)$
$\Rightarrow (\forall i \cdot i \in dom(tr) \land iSucc(i) \in dom(tr) \land tr(i) \in p1$
$\Rightarrow tr(iSucc(i)) \in p2)$
Lemme Convergent_In_P:
$\forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \land$
$m \in Machine(STATE, EVENT) \land$
$Machine\_WellCons(m) \land$
$check\_Machine\_Consistency(m) \land$
$IsATrace(m, tr) \land$
$TLConvergent\_In\_P(m, p, variant) \land$
$dom(tr) = iNAT$
$\Rightarrow (\forall i \cdot i \in dom(tr) \Rightarrow ((\forall k \cdot tr(iaddk) \in p)$
$\Rightarrow (\forall k \cdot variant(tr(iaddk)) \leq variant(tr(i)) - mk\_int(k))))$
thm of correctness of Convergent_In_P:

$$
\begin{array}{l}
\forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \wedge \\
\quad m \in Machine(STATE, EVENT) \wedge \\
\qquad Machine\_WellCons(m) \wedge \\
\qquad check\_Machine\_Consistency(m) \wedge \\
\qquad IsATrace(m, tr) \wedge \\
\qquad TLConvergent\_In\_P(m, p, variant) \wedge \\
\qquad dom(tr) = iNAT \\
\qquad\quad \Rightarrow (\forall i \cdot i \in dom(tr) \\
\qquad\qquad \Rightarrow (\exists j \cdot j \in iNAT \wedge mk\_int(j) \geq mk\_int(i) \wedge \\
\qquad\qquad\quad j \in dom(tr) \wedge tr(j) \notin p))
\end{array}
$$

lemme 1 Divergent_In_P:

$$
\begin{array}{l}
\forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \wedge \\
\quad m \in Machine(STATE, EVENT) \wedge \\
\qquad Machine\_WellCons(m) \wedge \\
\qquad check\_Machine\_Consistency(m) \wedge \\
\qquad IsATrace(m, tr) \wedge \\
\qquad TLDivergent\_In\_P(m, p, variant) \wedge \\
\qquad dom(tr) = iNAT \\
\qquad\quad \Rightarrow (\forall i \cdot \forall j \cdot variant(tr(iaddj)) \leq variant(tr(i)) \\
\qquad\qquad\qquad \vee variant(tr(iaddj)) \notin \mathbb{N})
\end{array}
$$

lemme 2 Divergent_In_P:

$$
\begin{array}{l}
\forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \wedge \\
m \in Machine(STATE, EVENT) \wedge \\
\qquad Machine\_WellCons(m) \wedge \\
\qquad check\_Machine\_Consistency(m) \wedge \\
\qquad IsATrace(m, tr) \wedge \\
\qquad TLDivergent\_In\_P(m, p, variant) \wedge \\
\qquad dom(tr) = iNAT \\
\qquad\quad \Rightarrow (\forall i \cdot \forall k \cdot \\
\qquad\qquad (\exists j \cdot variant(tr(iaddj)) < variant(tr(i)) - mk\_int(k)) \vee \\
\qquad\qquad (\exists j \cdot variant(tr(iaddj)) \notin \mathbb{N}))
\end{array}
$$

thm of correctness of Divergent_In_P:

$$
\begin{array}{l}
\forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \wedge \\
\quad m \in Machine(STATE, EVENT) \wedge \\
\qquad Machine\_WellCons(m) \wedge \\
\qquad check\_Machine\_Consistency(m) \wedge \\
\qquad IsATrace(m, tr) \wedge \\
\qquad TLDivergent\_In\_P(m, p, variant) \wedge \\
\qquad dom(tr) = iNAT \\
\qquad\quad \Rightarrow (\exists i \cdot \forall j \cdot mk\_int(j) \geq mk\_int(i) \Rightarrow tr(j) \in p)
\end{array}
$$

thm of correctness of Deadlock_Free_In_P:

$$
\begin{array}{l}
\forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \wedge \\
\quad m \in Machine(STATE, EVENT) \wedge \\
\qquad Machine\_WellCons(m) \wedge \\
\qquad check\_Machine\_Consistency(m) \wedge \\
\qquad IsATrace(m, tr) \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad TLDeadlock\_Free\_In\_P(m,p) \\
\qquad \Rightarrow (dom(tr) = iNAT \lor (\exists n \cdot n \in iNAT \land \\
\quad tr \in \{i \mid mk\_int(i) \in 0..mk\_int(n)\} \to STATE \land tr(n) \notin p)) \\
\textbf{\textit{thm of correctness of Globally}}: \\
\quad \forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \land \\
\qquad m \in Machine(STATE, EVENT) \land \\
\qquad\quad Machine\_WellCons(m) \land \\
\qquad\quad check\_Machine\_Consistency(m) \land \\
\qquad\quad IsATrace(m, tr) \land \\
\qquad\quad TLGlobally(m, p) \\
\qquad\qquad \Rightarrow (\forall i \cdot i \in dom(tr) \Rightarrow tr(i) \in p) \\
\textbf{\textit{thm of correctness of Existence}}: \\
\quad \forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \land \\
\qquad m \in Machine(STATE, EVENT) \land \\
\qquad\quad Machine\_WellCons(m) \land \\
\qquad\quad check\_Machine\_Consistency(m) \land \\
\qquad\quad IsATrace(m, tr) \land \\
\qquad\quad TLExistence(m, p, variant) \\
\qquad\qquad \Rightarrow (\forall i \cdot i \in dom(tr) \\
\qquad\qquad\quad \Rightarrow (\exists j \cdot mk\_int(j) \geq mk\_int(i) \land j \in dom(tr) \land tr(j) \in p)) \\
\textbf{\textit{thm of correctness of Until}}: \\
\quad \forall m, tr, variant, p1, p2 \cdot variant \in STATE \to \mathbb{Z} \land \\
\qquad m \in Machine(STATE, EVENT) \land \\
\qquad\quad Machine\_WellCons(m) \land \\
\qquad\quad check\_Machine\_Consistency(m) \land \\
\qquad\quad IsATrace(m, tr) \land \\
\qquad\quad TLUntil(m, variant, p1, p2) \\
\qquad\qquad \Rightarrow (\forall i \cdot i \in dom(tr) \land tr(i) \in p1 \\
\qquad\qquad\quad \Rightarrow (\exists j \cdot mk\_int(j) \geq mk\_int(i) \land j \in dom(tr) \land tr(j) \in p2 \land \\
\qquad\qquad\qquad (\forall k \cdot k \in mk\_int(i)..(mk\_int(j) - 1) \\
\qquad\qquad\qquad\quad \Rightarrow tr(mk\_iNAT(k)) \in p1))) \\
\textbf{\textit{thm of correctness of Progress}}: \\
\quad \forall m, tr, variant, p1, p2, p3 \cdot variant \in STATE \to \mathbb{Z} \land \\
\quad m \in Machine(STATE, EVENT) \land \\
\qquad\quad Machine\_WellCons(m) \land \\
\qquad\quad check\_Machine\_Consistency(m) \land \\
\qquad\quad IsATrace(m, tr) \land \\
\qquad\quad TLProgress(m, variant, p1, p2, p3) \\
\qquad\qquad \Rightarrow (\forall i \cdot i \in dom(tr) \land tr(i) \in p1 \\
\qquad\qquad\quad \Rightarrow (\exists j \cdot mk\_int(j) \geq mk\_int(i) \land \\
\qquad\qquad\qquad j \in dom(tr) \land tr(j) \in p2)) \\
\textbf{\textit{thm of correctness of Persistence}}: \\
\quad \forall m, tr, variant, p \cdot variant \in STATE \to \mathbb{Z} \land \\
\qquad m \in Machine(STATE, EVENT) \land \\
\qquad\quad Machine\_WellCons(m) \land \\
\qquad\quad check\_Machine\_Consistency(m) \land
\end{array}
$$

$$
\begin{array}{l}
\quad IsATrace(m, tr) \wedge \\
\quad TLPersistence(m, p, variant) \\
\qquad \Rightarrow (\exists i \cdot i \in dom(tr) \wedge \\
\qquad\quad (\forall j \cdot mk\_int(j) \geq mk\_int(i) \wedge j \in dom(tr) \Rightarrow tr(j) \in p))
\end{array}
$$

END

Appendix B

Models

B.1 Clock Models

B.1.1 Classical Event-B

Listing B.1: The Clock models

MACHINE
Horloge
VARIABLES m, h
INVARIANTS
inv1: $m \in \mathbb{N}$
inv2: $h \in \mathbb{N}$
inv3: $m < 60$
inv4: $h < 24$
thm: $m < 59 \lor (m = 59 \land h < 23) \lor (m = 59 \land h = 23)$
VARIANT $24 * 60 - 1 - (m + h * 60)$
EVENTS
INITIALISATION
THEN
act1: $m, h :\mid m' = 0 \land h' = 0$
END

tick_min *convergent*
WHERE
grd1: $m < 59$
THEN
act1: $m :\mid m' = m + 1$
END

tick_heure *convergent*
WHERE

grd1 : $m = 59 \land h < 23$
THEN
act1 : $m, h :| m' = 0 \land h' = h + 1$
END

tick_minuit
WHERE
grd1 : $m = 59 \land h = 23$
THEN
act1 : $m, h :| m' = 0 \land h' = 0$
END

END

B.1.2 Instantiation of EB4EB

Deep Modelling

Listing B.2: Instantiation of the Clock in Deep modelling

CONTEXT
ClockDeep
SETS
Ev
CONSTANTS
clock
tick_min
tick_hour
tick_midnight
init
AXIOMS
axm1 : $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm2 : $clock \in Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$
axm3 : $Event(clock) = Ev$
axm4 : $Init(clock) = init$
axm5 : $State(clock) = \mathbb{Z} \times \mathbb{Z}$
axm6 : $Inv(clock) = \{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}$
axm7 : $Progress(clock) = \{tick\_midnight, tick\_hour, tick\_min\}$
axm8 : $Thm(clock) = \{m \mapsto h \mid m < 59 \lor (m = 59 \land (h < 23 \lor h = 23))\}$
axm9 : $AP(clock) = \{m \mapsto h \mid m = 0 \land h = 0\}$
axm10 :
$BAP(clock) = \{t \mapsto ((m \mapsto h) \mapsto (mp \mapsto hp)) \mid ($
$(t = tick\_min \land mp = m + 1 \land hp = h) \lor$
$(t = tick\_hour \land mp = 0 \land hp = h + 1) \lor$
$(t = tick\_midnight \land mp = 0 \land hp = 0))\}$
axm11 :

$Grd(clock) = \{t \mapsto (m \mapsto h) \mid ($
$(t = tick\_min \wedge m < 59) \vee$
$(t = tick\_hour \wedge m = 59 \wedge h < 23) \vee$
$(t = tick\_midnight \wedge m = 59 \wedge h = 23))\}$

axm12 : $Convergent(clock) = \{tick\_min, tick\_hour\}$

axm13 : $Variant(clock) = \{m \mapsto h \mapsto v \mid v = 24 * 60 - 1 - (m + h * 60)\}$

axm14 : $Ordinary(clock) = \{init, tick\_midnight\}$

THEOREMS

axm16 : $check\_Machine\_Consistency(clock)$

END

Shallow Modelling

Listing B.3: Instantiation of the static part of the clock models in shallow modelling

CONTEXT
ClockShallowCtx
EXTENDS
ShallowCtx
CONSTANTS
tick_min
tick_hour
tick_midnight
init
pr
AXIOMS

axm1 : $pr \in (\mathbb{Z} \times \mathbb{Z}) \rightarrowtail S$

axm2 : $partition(EV, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$

axm3 : $Event(m) = EV$

axm4 : $Init(m) = init$

axm5 : $State(m) = S$

axm6 : $Inv(m) = pr[\{mi \mapsto h \mid mi \in \mathbb{N} \wedge h \in \mathbb{N} \wedge mi < 60 \wedge h < 24\}]$

axm7 : $Thm(m) =$
$pr[\{mi \mapsto h \mid mi < 59 \vee (mi = 59 \wedge h < 23) \vee (mi = 59 \wedge h = 23)\}]$

axm8 : $Variant(m) =$
$\{s, n, mi, h \cdot s = pr(mi \mapsto h) \wedge n = (24 * 60 - 1 - (mi + h * 60)) \mid s \mapsto n\}$

axm9 : $Convergent(m) = \{tick\_min, tick\_hour\}$

axm10 : $Ordinary(m) = \{init, tick\_midnight\}$

axm12 : $Progress(m) = \{tick\_midnight, tick\_hour, tick\_min\}$

axm11 : $Anticipated(m) = \emptyset$

thm1 : $Mch\_THM(m)$

thm2 : $Event\_WellCons(m)$

thm3 : $Variant\_WellCons(m)$

thm4 : $Tag\_Event\_WellCons(m)$

END

Listing B.4: Instantiation of the dynamic part of the clock models in shallow modelling

MACHINE
ClockShallow
REFINES
ShallowMch
SEES
ClockShallowCtx
VARIABLES mi, h, $InitDone$
INVARIANTS
inv1: $s = pr(mi \mapsto h)$
EVENTS
INITIALISATION
WITH
s': $s' = pr(mi' \mapsto h')$
THEN
act1: $mi :\in \mathbb{Z}$
act2: $h :\in \mathbb{Z}$
act3: $InitDone := FALSE$
END

Do_Init
REFINES Do_Init
WHERE
grd1: $pr[\{0 \mapsto 0\}] = AP(m)$
grd2: $InitDone = FALSE$
WITH
s': $s' = pr(mi' \mapsto h')$
THEN
act2: $InitDone := TRUE$
act1: $mi, h :| pr(mi' \mapsto h') \in AP(m)$
END

Tick_min
REFINES $Do\_Convergent$
WHERE
grd1: $pr(mi \mapsto h) \in Grd(m)[\{tick\_min\}]$
grd2: $InitDone = TRUE$
grd3: $pr[\{ms, hs \cdot ms < 59 \wedge hs \in \mathbb{Z} \mid ms \mapsto hs\}] = Grd(m)[\{tick\_min\}]$
grd4: $\{ss, ssp, ms, hs, msp, hsp \cdot$
$ss = pr(ms \mapsto hs) \wedge ssp = pr(msp \mapsto hsp) \wedge$
$msp = ms + 1 \wedge hs = hsp$
$\mid ss \mapsto ssp\} = BAP(m)[\{tick\_min\}]$
WITH
e: $e = tick\_min$

s' : $s' = pr(mi' \mapsto h')$
THEN
act1 : $mi, h :| pr(mi' \mapsto h') \in BAP(m)[\{tick\_min\}][\{pr(mi \mapsto h)\}]$
END

Tick_hour
REFINES *Do_Convergent*
WHERE
grd1 : $pr(mi \mapsto h) \in Grd(m)[\{tick\_hour\}]$
grd2 : $InitDone = TRUE$
grd3 :
$pr[\{ms, hs \cdot hs < 23 \wedge ms = 59 \mid ms \mapsto hs\}] = Grd(m)[\{tick\_hour\}]$
grd4 : $\{ss, ssp, ms, hs, msp, hsp \cdot$
$ss = pr(ms \mapsto hs) \wedge ssp = pr(msp \mapsto hsp) \wedge msp = ms \wedge$
$hsp = hs + 1$
$\mid ss \mapsto ssp\} = BAP(m)[\{tick\_hour\}]$
WITH
e : $e = tick\_hour$
s' : $s' = pr(mi' \mapsto h')$
THEN
act1 : $mi, h :| pr(mi' \mapsto h') \in BAP(m)[\{tick\_hour\}][\{pr(mi \mapsto h)\}]$
END

Tick_midnight
REFINES *Do_Ordinary*
WHERE
grd1 : $pr(mi \mapsto h) \in Grd(m)[\{tick\_midnight\}]$
grd2 : $InitDone = TRUE$
grd3 :
$pr[\{ms, hs \cdot ms = 59 \wedge hs = 23 \mid ms \mapsto hs\}] = Grd(m)[\{tick\_midnight\}]$

grd4 : $\{ss, ssp, ms, hs, msp, hsp \cdot$
$ss = pr(ms \mapsto hs) \wedge ssp = pr(msp \mapsto hsp)$
$\wedge msp = 0 \wedge hsp = 0$
$\mid ss \mapsto ssp\} = BAP(m)[\{tick\_midnight\}]$
WITH
e : $e = tick\_midnight$
s' : $s' = pr(mi' \mapsto h')$
THEN
act1 : $mi, h :| pr(mi' \mapsto h') \in BAP(m)[\{tick\_midnight\}][\{pr(mi \mapsto h)\}]$
END

END

B.1.3 Analyses

Listing B.5: Dealock freeness analysis on the Clock models

CONTEXT
ClockDeadlockFree
SETS
Ev
CONSTANTS
clock
tick_min
tick_hour
tick_midnight
init
AXIOMS
axm1: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm2: $clock \in Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$
axm3: $Event(clock) = Ev$
axm4: $Init(clock) = init$
axm5: $State(clock) = \mathbb{Z} \times \mathbb{Z}$
axm6: $Inv(clock) = \{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}$
axm7: $Progress(clock) = \{tick\_midnight, tick\_hour, tick\_min\}$
axm8: $Thm(clock) = State(clock)$
axm9: $AP(clock) = \{m \mapsto h \mid m = 0 \land h = 0\}$
axm10:
$BAP(clock) = \{t \mapsto ((m \mapsto h) \mapsto (mp \mapsto hp)) \mid ($
$(t = tick\_min \land mp = m + 1 \land hp = h) \lor$
$(t = tick\_hour \land mp = 0 \land hp = h + 1) \lor$
$(t = tick\_midnight \land mp = 0 \land hp = 0))\}$
axm11:
$Grd(clock) = \{t \mapsto (m \mapsto h) \mid ($
$(t = tick\_min \land m < 59) \lor$
$(t = tick\_hour \land m = 59 \land h < 23) \lor$
$(t = tick\_midnight \land m = 59 \land h = 23))\}$
axm12: $Convergent(clock) = \{tick\_min, tick\_hour\}$
axm13: $Anticipated(clock) = \emptyset$
axm14: $Variant(clock) = \{m \mapsto h \mapsto v \mid v = 24 * 60 - 1 - (m + h * 60)\}$
axm15: $Ordinary(clock) = \{init, tick\_midnight\}$
thm1: $check\_Machine\_Consistency(clock)$
thmDeadlock: $check\_Machine\_DeadLockFreeness(clock)$
END

Listing B.6: Weak invariant analysis on the Clock models

CONTEXT
ClockInspectInv
SETS

Ev

CONSTANTS

clock
tick_min
tick_hour
tick_midnight
init

AXIOMS

axm1: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$

axm2: $clock \in Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$

axm3: $Event(clock) = Ev$

axm4: $Init(clock) = init$

axm5: $State(clock) = \mathbb{Z} \times \mathbb{Z}$

axm6: $Inv(clock) = \{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}$

axm7: $Progress(clock) = \{tick\_midnight, tick\_hour, tick\_min\}$

axm8: $Thm(clock) = State(clock)$

axm9: $AP(clock) = \{m \mapsto h \mid m = 0 \land h = 0\}$

axm10:

$$BAP(clock) = \{t \mapsto ((m \mapsto h) \mapsto (mp \mapsto hp)) \mid ($$
$$(t = tick\_min \land mp = m + 1 \land hp = h) \lor$$
$$(t = tick\_hour \land mp = 0 \land hp = h + 1) \lor$$
$$(t = tick\_midnight \land mp = 0 \land hp = 0))\}$$

axm11:

$$Grd(clock) = \{t \mapsto (m \mapsto h) \mid ($$
$$(t = tick\_min \land m < 59) \lor$$
$$(t = tick\_hour \land m = 59 \land h < 23) \lor$$
$$(t = tick\_midnight \land m = 59 \land h = 23))\}$$

axm12: $Convergent(clock) = \{tick\_min, tick\_hour\}$

axm13: $Anticipated(clock) = \emptyset$

axm14: $Variant(clock) = \{m \mapsto h \mapsto v \mid v = 24 * 60 - 1 - (m + h * 60)\}$

axm15: $Ordinary(clock) = \{init, tick\_midnight\}$

thm1: $check\_Machine\_Consistency(clock)$

thmInspectInvEvtM5: $check\_Machine\_AllowedMachineHoleSub(clock,$
$\{m \mapsto h \mid m < 55 \land h \in \mathbb{Z}\},$
$\{(m \mapsto h) \mapsto (mp \mapsto hp) \mid mp = m + 5 \land hp = h\})$

thmInspectInvEvtH5: $check\_Machine\_AllowedMachineHoleSub(clock,$
$\{m \mapsto h \mid m = 0 \land h < 19\},$
$\{(m \mapsto h) \mapsto (mp \mapsto hp) \mid mp = 0 \land hp = h + 1 \land m \in \mathbb{Z}\})$

thmInspectInvEvtMH1: $check\_Machine\_AllowedMachineHoleSub(clock,$
$\{m \mapsto h \mid m < 59 \land h < 23\},$
$\{(m \mapsto h) \mapsto (mp \mapsto hp) \mid mp = m + 1 \land hp = h + 1\})$

END

Listing B.7: The Strenghtening Clock machine resulting of the Weak invariant analysis

CONTEXT

ClockInvStrong

SETS

Ev

CONSTANTS

clock
tick_min
tick_hour
tick_midnight
init

AXIOMS

axm1: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$

axm2: $clock \in Machine(\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}, Ev)$

axm3: $Event(clock) = Ev$

axm4: $Init(clock) = init$

axm5: $State(clock) = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$

axm6:

$$Inv(clock) = \{m \mapsto h \mapsto mb \mapsto hb \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24 \land ((m = mb + 1 \land h = hb) \lor (m = 0 \land (h = hb + 1 \lor h = 0)))\}$$

axm7: $Progress(clock) = \{tick\_midnight, tick\_hour, tick\_min\}$

axm8: $Thm(clock) = State(clock)$

axm9:

$$AP(clock) = \{m \mapsto h \mapsto mb \mapsto hb \mid m = 0 \land h = 0 \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\}$$

axm10:

$$BAP(clock) = \{t \mapsto ((m \mapsto h \mapsto mb \mapsto hb) \mapsto (mp \mapsto hp \mapsto mbp \mapsto hbp)) \mid mb \in \mathbb{Z} \land hb \in \mathbb{Z} \land ((t = tick\_min \land mp = m + 1 \land hp = h \land mbp = m \land hbp = h) \lor (t = tick\_hour \land mp = 0 \land hp = h + 1 \land mbp = m \land hbp = h) \lor (t = tick\_midnight \land mp = 0 \land hp = 0 \land mbp = m \land hbp = h))\}$$

axm11:

$$Grd(clock) = \{t \mapsto (m \mapsto h \mapsto mb \mapsto hb) \mid mb \in \mathbb{Z} \land hb \in \mathbb{Z} \land ((t = tick\_min \land m < 59) \lor (t = tick\_hour \land m = 59 \land h < 23) \lor (t = tick\_midnight \land m = 59 \land h = 23))\}$$

axm12: $Convergent(clock) = \{tick\_min, tick\_hour\}$

axm13: $Variant(clock) = \{m \mapsto h \mapsto mb \mapsto hb \mapsto v \mid mb \in \mathbb{Z} \land hb \in \mathbb{Z} \land v = 24 * 60 - 1 - (m + h * 60)\}$

axm14: $Ordinary(clock) = \{init, tick\_midnight\}$

axm15: $Anticipated(clock) = \emptyset$

thm1: $check\_Machine\_Consistency(clock)$

thmInspectInvEvtM5: $\neg check\_Machine\_AllowedMachineHoleSub(clock, \{m \mapsto h \mapsto mb \mapsto hb \mid m < 55 \land h \in \mathbb{Z} \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\}, \{(m \mapsto h \mapsto mb \mapsto hb) \mapsto (mp \mapsto hp \mapsto mbp \mapsto hbp) \mid mp = m + 5 \land hp = h \land mbp = m \land hbp = h \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\})$

thmInspectInvEvtH5: $\neg check\_Machine\_AllowedMachineHoleSub(clock, \{m \mapsto h \mapsto mb \mapsto hb \mid m = 59 \land h < 19 \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\},$

$\{(m \mapsto h \mapsto mb \mapsto hb) \mapsto (mp \mapsto hp \mapsto mbp \mapsto hbp) \mid$
$mp = 0 \land hp = h + 5 \land mbp = m \land hbp = h \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\})$

thmInspectInvEvtMH1:
$\neg check\_Machine\_AllowedMachineHoleSub(clock,$
$\{m \mapsto h \mapsto mb \mapsto hb \mid m < 59 \land h < 23 \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\},$
$\{(m \mapsto h \mapsto mb \mapsto hb) \mapsto (mp \mapsto hp \mapsto mbp \mapsto hbp) \mid$
$mp = m + 1 \land hp = h + 1 \land mbp = m \land hbp = h \land mb \in \mathbb{Z} \land hb \in \mathbb{Z}\})$

END

Listing B.8: Reachability analysis on the Clock models

CONTEXT
ClockReachability
SETS
Ev
CONSTANTS
clock
tick_min
tick_hour
tick_midnight
init
AXIOMS
axm1: $partition(Ev, \{init\}, \{tick\_midnight\}, \{tick\_hour\}, \{tick\_min\})$
axm2: $clock \in Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$
axm3: $Event(clock) = Ev$
axm4: $Init(clock) = init$
axm5: $State(clock) = \mathbb{Z} \times \mathbb{Z}$
axm6: $Inv(clock) = \{m \mapsto h \mid m \in \mathbb{N} \land h \in \mathbb{N} \land m < 60 \land h < 24\}$
axm7: $Progress(clock) = \{tick\_midnight, tick\_hour, tick\_min\}$
axm8: $Thm(clock) = State(clock)$
axm9: $AP(clock) = \{m \mapsto h \mid m = 0 \land h = 0\}$
axm10:
$BAP(clock) = \{t \mapsto ((m \mapsto h) \mapsto (mp \mapsto hp)) \mid ($
$(t = tick\_min \land mp = m + 1 \land hp = h) \lor$
$(t = tick\_hour \land mp = 0 \land hp = h + 1) \lor$
$(t = tick\_midnight \land mp = 0 \land hp = 0))\}$
axm11:
$Grd(clock) = \{t \mapsto (m \mapsto h) \mid ($
$(t = tick\_min \land m < 59) \lor$
$(t = tick\_hour \land m = 59 \land h < 23) \lor$
$(t = tick\_midnight \land ((m \geq 59 \land h = 23) \lor h \geq 24)))\}$
axm12: $Convergent(clock) = \{tick\_min, tick\_hour\}$
axm13: $Variant(clock) = \{m \mapsto h \mapsto v \mid v = 24 * 60 - 2 - (m + h * 60)\}$
axm14: $Ordinary(clock) = \{init, tick\_midnight\}$
axm15: $Anticipated(clock) = \emptyset$
thm1: $check\_Machine\_Consistency(clock)$
thmReach: $check\_Machine\_Evt\_Is\_Reachable\_From(clock,$

$init, tick\_midnight, Convergent(clock), Variant(clock))$

END

B.2 Read/Write system

B.2.1 Classical Event-B

Listing B.9: The Read/Write machine

MACHINE $RdWRMch$
VARIABLES r , w
INVARIANTS
inv1 : $r \in \mathbb{N}$
inv3 : $w \in \mathbb{N}$
inv4 : $0 \leq w - r$
inv2 : $w - r \leq 3$
EVENTS
INITIALISATION
THEN
act1 : $r, w := 0, 0$
END

read
WHERE
grd1 : $r < w$
THEN
act1 : $r := r + 1$
END

write
WHERE
grd1 : $w < r + 3$
THEN
act1 : $w := w + 1$
END

END

B.2.2 Instantiation of EB4EB with analyses

Listing B.10: The instance of read write machine with temporal properties

CONTEXT $RdWR$
SETS Ev
CONSTANTS $rdwr$, $init$, $read$, $write$, L

AXIOMS

axm1: $partition(Ev, \{init\}, \{read\}, \{write\})$
axm2: $rdwr \in Machine(\mathbb{Z} \times \mathbb{Z}, Ev)$
axm3: $Event(rdwr) = Ev$
axm5: $State(rdwr) = \mathbb{Z} \times \mathbb{Z}$
axm6: $Init(rdwr) = init$
axm7: $Inv(rdwr) = \{r \mapsto w \mid r \in \mathbb{N} \land w \in \mathbb{N} \land 0 \leq w - r \land w - r \leq 3\}$
axm8: $AP(rdwr) = \{0 \mapsto 0\}$
axm9: $BAP(rdwr) = \{e \mapsto ((r \mapsto w) \mapsto (rp \mapsto wp)) \mid$
$(e = read \land rp = r + 1 \land wp = w)$
$\lor (e = write \land rp = r \land wp = w + 1)\}$
axm10: $Grd(rdwr) = \{e \mapsto (r \mapsto w) \mid$
$(e = read \land r < w) \lor (e = write \land w < r + 3)\}$
axm11: $Progress(rdwr) = \{read, write\}$
axm12: $Convergent(rdwr) = \emptyset$
axm13: $Ordinary(rdwr) = Ev$
axm14: $Variant(rdwr) \in \mathbb{Z} \times \mathbb{Z} \to \mathbb{N}$
axm15: $Thm(rdwr) = State(rdwr)$
thm1: $check\_Machine\_Consistency(rdwr)$
axm17: $L \in \mathbb{N}$
thmEx1: $TLExistence($
$rdwr,$
$\{r \mapsto w \mid w \in \mathbb{Z} \land r \geq L\},$
$\{(r \mapsto w) \mapsto v \mid v = ((L - r) + (L + 3 - w))\})$
thmEx2: $TLProgress($
$rdwr,$
$\{(r \mapsto w) \mapsto v \mid v = ((L - r) + (L + 3 - w))\},$
$\{r \mapsto w \mid r \in \mathbb{Z} \land w = L\},$
$\{r \mapsto w \mid r = L \land w \in \mathbb{Z}\}, \{r \mapsto w \mid r < L \land w \in \mathbb{Z}\})$
thmEx3: $TLPersistence($
$rdwr,$
$\{r \mapsto w \mid L \leq w \land r \in \mathbb{Z}\},$
$\{(r \mapsto w) \mapsto v \mid v = (L - r) + (L - w)\})$

END

B.3 Peterson

B.3.1 Classical Event-B

Listing B.11: The peterson algorithm models

MACHINE *PetersonMch*
VARIABLES a, b, w_a, w_b, $turn$
INVARIANTS

inv1:
$a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn \in \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\}$
inv0_1: $(w\_a = 0 \Rightarrow a = 0)$
inv0_2: $(w\_b = 0 \Rightarrow b = 0)$
inv0_3: $(a = 0 \vee b = 0)$
inv1_1: $(turn = 0 \wedge w\_a = 1 \Rightarrow b = 0)$
inv1_2: $(turn = 1 \wedge w\_b = 1 \Rightarrow a = 0)$
inv2: $(w\_a = 1 \wedge w\_b = 0 \Rightarrow turn = 1)$

EVENTS

INITIALISATION
THEN
act1: $a := 0$
act2: $b := 0$
act3: $w\_a := 0$
act4: $w\_b := 0$
act5: $turn := 0$
END

wish_a
WHERE
grd1: $w\_a = 0$
THEN
act1: $w\_a := 1$
act2: $turn := 1$
END

enter_a
WHERE
grd1: $w\_a = 1$
grd3: $a = 0$
grd2: $w\_b = 0 \vee turn = 0$
THEN
act1: $a := 1$
END

leave_a
WHERE
grd1: $a = 1$
THEN
act1: $a := 0$
act2: $w\_a := 0$
END

wish_b
WHERE
grd1: $w\_b = 0$

THEN
act1: $w\_b := 1$
act2: $turn := 0$
END

enter_b
WHERE
grd1: $w\_b = 1$
grd3: $b = 0$
grd2: $w\_a = 0 \lor turn = 1$
THEN
act1: $b := 1$
END

leave_b
WHERE
grd1: $b = 1$
THEN
act1: $b := 0$
act2: $w\_b := 0$
END

END

B.3.2 Instantiation of EB4EB

Listing B.12: The instantiation of the peterson algorithm with liveness properties

CONTEXT *Peterson*
SETS *Ev*
CONSTANTS $peterson$, $init$, $wish\_a$, $enter\_a$, $leave\_a$, $wish\_b$, $leave\_b$, $enter\_b$,
AXIOMS
axm1: $partition(Ev, \{init\}, \{enter\_a\}, \{wish\_a\}, \{leave\_a\}, \{enter\_b\}, \{wish\_b\}, \{leave\_b\})$
axm2: $peterson \in Machine(\mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}, Ev)$
axm3: $Event(peterson) = Ev$
axm5: $State(peterson) = \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}$
axm6: $Init(peterson) = init$
axm7: $Inv(peterson) = \{a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn \mid$
$a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn \in \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \times \{0,1\} \land$
$(w\_a = 0 \Rightarrow a = 0) \land (w\_b = 0 \Rightarrow b = 0) \land (a = 0 \lor b = 0) \land$
$(turn = 0 \land w\_a = 1 \Rightarrow b = 0) \land (turn = 1 \land w\_b = 1 \Rightarrow a = 0) \land$
$(w\_a = 1 \land w\_b = 0 \Rightarrow turn = 1)\}$

$$
\begin{array}{ll}
\textbf{axm8}: & AP(peterson) = \{0 \mapsto 0 \mapsto 0 \mapsto 0 \mapsto 0\} \\
\textbf{axm9}: & BAP(peterson) = \\
& \{e \mapsto \\
& \quad ((a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn) \mapsto \\
& \quad (ap \mapsto bp \mapsto w\_ap \mapsto w\_bp \mapsto turnp)) \mid \\
& \quad (e = wish\_a \land \\
& \quad w\_ap = 1 \land turnp = 1 \land ap = a \land bp = b \land w\_bp = w\_b) \\
& \lor (e = enter\_a \land \\
& \quad w\_ap = w\_a \land turnp = turn \land ap = 1 \land bp = b \land w\_bp = w\_b) \\
& \lor (e = leave\_a \land \\
& \quad w\_ap = 0 \land turnp = turn \land ap = 0 \land bp = b \land w\_bp = w\_b) \\
& \lor (e = wish\_b \land \\
& \quad w\_ap = w\_a \land turnp = 0 \land ap = a \land bp = b \land w\_bp = 1) \\
& \lor (e = enter\_b \land \\
& \quad w\_ap = w\_a \land turnp = turn \land ap = a \land bp = 1 \land w\_bp = w\_b) \\
& \lor (e = leave\_b \land \\
& \quad w\_ap = w\_a \land turnp = turn \land ap = a \land bp = 0 \land w\_bp = 0) \\
& \quad \} \\
\textbf{axm10}: & Grd(peterson) = \{e \mapsto (a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn) \mid \\
& \quad (e = wish\_a \land w\_a = 0) \\
& \quad \lor (e = enter\_a \land w\_a = 1 \land a = 0 \land (w\_b = 0 \lor turn = 0)) \\
& \quad \lor (e = leave\_a \land a = 1) \\
& \quad \lor (e = wish\_b \land w\_b = 0) \\
& \quad \lor (e = enter\_b \land w\_b = 1 \land b = 0 \land (w\_a = 0 \lor turn = 1)) \\
& \quad \lor (e = leave\_b \land b = 1)\} \\
\textbf{axm11}: & Progress(peterson) = \\
& \quad \{enter\_a, wish\_a, leave\_a, enter\_b, wish\_b, leave\_b\} \\
\textbf{axm12}: & Convergent(peterson) = \emptyset \\
\textbf{axm13}: & Ordinary(peterson) = Ev \\
\textbf{axm14}: & Variant(peterson) \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \to \mathbb{N} \\
\textbf{axm15}: & Thm(peterson) = State(peterson) \\
\textbf{thm1}: & check\_Machine\_Consistency(peterson) \\
\textbf{thmEx2}: & TLProgress(\\
& \quad peterson, \\
& \quad \{(a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn) \mapsto v \mid \\
& \quad \quad v = 2 * w\_b + 3 * turn - b - a \land w\_a \in \mathbb{Z}\}, \\
& \quad \{a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn \mid \\
& \quad \quad w\_a = 1 \land a \mapsto b \mapsto w\_b \mapsto turn \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}\}, \\
& \quad \{a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn \mid \\
& \quad \quad a = 1 \land w\_a \mapsto b \mapsto w\_b \mapsto turn \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}\}, \\
& \quad \{a \mapsto b \mapsto w\_a \mapsto w\_b \mapsto turn \mid \\
& \quad \quad w\_a = 1 \land a \mapsto b \mapsto w\_b \mapsto turn \in \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z} \times \mathbb{Z}\}) \\
\textbf{END} &
\end{array}
$$

B.4 Calibration

B.4.1 Classical Event-B

Listing B.13: The context of calibration models

CONTEXT $CalibrationCtx$
SETS $Mode$
CONSTANTS on , off , M
AXIOMS
axm1: $partition(Mode, \{on\}, \{off\})$
axm2: $M \in \mathbb{N}$
END

Listing B.14: the calibration models

MACHINE $CalibrationMch$
SEES $CalibrationCtx$
VARIABLES s , t
INVARIANTS
inv1: $s \in Mode$
inv3: $t \in \mathbb{Z}$
inv2: $s = off \Rightarrow t \leq M$
EVENTS
INITIALISATION
THEN
act1: $t := 0$
act2: $s :\in Mode$
END

calibrate_on
WHERE
grd1: $s = off$
grd2: $t \leq M$
THEN
act1: $t := t + 1$
act2: $s := on$
END

calibrate_off
WHERE
grd1: $s = on$
grd2: $t < M$
THEN
act1: $t := t + 1$
act2: $s := off$
END

working
WHERE
grd1: $s = on$
grd2: $M \leq t$
THEN
act1: $t := t + 1$
END

END

B.4.2 Instantiation of EB4EB with analyses

Listing B.15: The instantiation of the calibration models with temporal properties

CONTEXT *Calibration*
SETS Ev, $Mode$
CONSTANTS $calibration$, $init$, $calibrate\_on$, $calibrate\_off$, $working$, on, off, M
AXIOMS
axm1: $partition(Mode, \{on\}, \{off\})$
axm2: $M \in \mathbb{N}$
axm3: $partition(Ev, \{init\}, \{calibrate\_on\}, \{calibrate\_off\}, \{working\})$
axm4: $calibration \in Machine(Mode \times \mathbb{Z}, Ev)$
axm5: $Event(calibration) = Ev$
axm6: $State(calibration) = Mode \times \mathbb{Z}$
axm7: $Init(calibration) = init$
axm8: $Inv(calibration) = \{s \mapsto t \mid s = off \Rightarrow t \leq M\}$
axm9: $AP(calibration) = \{on \mapsto 0, off \mapsto 0\}$
axm10: $BAP(calibration) = \{e \mapsto ((s \mapsto t) \mapsto (sp \mapsto tp)) \mid$
$tp = t + 1 \wedge$
$((e = calibrate\_on \wedge sp = on)$
$\vee (e = calibrate\_off \wedge sp = off)$
$\vee (e = working \wedge sp = s))\}$
axm11: $Grd(calibration) = \{e \mapsto (s \mapsto t) \mid$
$(e = calibrate\_on \wedge s = off \wedge t \leq M)$
$\vee (e = calibrate\_off \wedge s = on \wedge t < M)$
$\vee (e = working \wedge s = on \wedge M \leq t)\}$
axm12: $Progress(calibration) = \{calibrate\_on, calibrate\_off, working\}$
axm13: $Convergent(calibration) = \emptyset$
axm14: $Ordinary(calibration) = Ev$
axm15: $Variant(calibration) \in Mode \times \mathbb{Z} \rightarrow \mathbb{N}$
axm16: $Thm(calibration) = State(calibration)$
thm1: $check\_Machine\_Consistency(calibration)$
thmEx3: $TLPersistence($
$calibration,$

$\{s \mapsto t \mid s = on \land t \in \mathbb{Z}\},$
$\{(s \mapsto t) \mapsto v \mid v = M - t \land s \in Mode\})$

END

B.5 Automatic Teller Machine

B.5.1 Classical Event-B

Listing B.16: The context of the ATM models

CONTEXT
$ATMEnvironment$
SETS
$INPUT\_MODE$
$INSERTION\_STATUS$
$STRINGS$
$AMOUNTS$
CONSTANTS
$MAX\_ATTEMPTS$
$CORRECT\_PASSCODE$
$KEYBOARD$
$SCREEN$
IN
OUT
$BRIGHTNESS\_LEVELS$
$BRIGHTNESS\_MIN$
$BRIGHTNESS\_MAX$
$EMPTY\_STRING$
$MAX\_BRIGHTNESS\_UPDATE$
NO_MONEY
AXIOMS
axm1: $MAX\_ATTEMPTS \in \mathbb{N}1$
axm2: $CORRECT\_PASSCODE \in STRINGS \land$
$EMPTY\_STRING \in STRINGS$
axm3: $CORRECT\_PASSCODE \neq EMPTY\_STRING$
axm4: $partition(INPUT\_MODE, \{KEYBOARD\}, \{SCREEN\})$
axm5: $partition(INSERTION\_STATUS, \{IN\}, \{OUT\})$
axm6: $BRIGHTNESS\_MIN \in \mathbb{N}$
axm7: $BRIGHTNESS\_MAX \in \mathbb{N}$
axm8: $BRIGHTNESS\_MAX > BRIGHTNESS\_MIN$
axm9:
$BRIGHTNESS\_LEVELS = BRIGHTNESS\_MIN..BRIGHTNESS\_MAX$
axm10: $MAX\_BRIGHTNESS\_UPDATE \in \mathbb{N}$
axm11: $NO\_MONEY \in AMOUNTS$
axm12: $AMOUNTS \setminus \{NO\_MONEY\} \neq \emptyset$

END

Listing B.17: The ATM models

MACHINE
$ATMUserInterface$
SEES
$ATMEnvironment$
VARIABLES $string$, $virtualNumpadRegister$, $keyboardRegister$, $attempts$, $confirmationStatus$, $validationStatus$, $deliveryStatus$, $isStringVisible$, $inputMode$, $cardStatus$, $brightness$, $brightnessUpdates$, $newString$, sum
INVARIANTS
inv1: $string \in STRINGS$
inv2: $virtualNumpadRegister \in STRINGS$
inv3: $keyboardRegister \in STRINGS$
inv4: $attempts \in 0..MAX\_ATTEMPTS$
inv5: $validationStatus \in BOOL$
inv6: $deliveryStatus \in BOOL$
inv7: $confirmationStatus \in BOOL$
inv8: $isStringVisible \in BOOL$
inv9: $inputMode \in INPUT\_MODE$
inv10: $cardStatus \in INSERTION\_STATUS$
inv11: $string = virtualNumpadRegister \lor string = keyboardRegister$
inv12: $isStringVisible = FALSE$
inv13: $brightness \in BRIGHTNESS\_LEVELS$
inv14: $brightnessUpdates \in \mathbb{Z}$
inv15: $newString \in BOOL$
inv16: $sum \in AMOUNTS$
EVENTS
INITIALISATION
THEN
act1: $string := EMPTY\_STRING$
act2: $attempts := 0$
act3: $virtualNumpadRegister := EMPTY\_STRING$
act4: $keyboardRegister := EMPTY\_STRING$
act5:
$confirmationStatus, validationStatus, deliveryStatus, isStringVisible := TRUE, FALSE, FALSE, FALSE$
act6: $inputMode :\in INPUT\_MODE$
act7: $cardStatus := OUT$
act8: $brightness :\in BRIGHTNESS\_LEVELS$
act9: $brightnessUpdates := 0$
act10: $newString := FALSE$
act11: $sum := NO\_MONEY$
END

insertCard
WHERE
$grd1$: $cardStatus = OUT$
THEN
$act1$: $cardStatus := IN$
$act2$: $inputMode :\in INPUT\_MODE$
$act3$: $sum := NO\_MONEY$
$act4$: $validationStatus := FALSE$
$act5$: $newString := FALSE$
END

KBDString
WHERE
$grd1$: $cardStatus = IN$
$grd2$: $inputMode = KEYBOARD$
$grd3$: $confirmationStatus = TRUE$
$grd4$: $newString = FALSE$
$grd5$: $0 \leq attempts \wedge attempts < MAX\_ATTEMPTS$
THEN
$act1$: $string, keyboardRegister :|$
$keyboardRegister' \in STRINGS \wedge string' = keyboardRegister'$
$act2$: $brightnessUpdates := 0$
$act3$: $confirmationStatus := FALSE$
$act4$: $newString := TRUE$
$act5$: $validationStatus := FALSE$
END

SCRString
WHERE
$grd1$: $cardStatus = IN$
$grd2$: $inputMode = SCREEN$
$grd3$: $confirmationStatus = TRUE$
$grd4$: $newString = FALSE$
$grd5$: $0 \leq attempts \wedge attempts < MAX\_ATTEMPTS$
THEN
$act1$: $string, virtualNumpadRegister :|$
$virtualNumpadRegister' \in STRINGS \wedge$
$string' = virtualNumpadRegister'$
$act2$: $brightnessUpdates := 0$
$act3$: $confirmationStatus := FALSE$
$act4$: $newString := TRUE$
$act5$: $validationStatus := FALSE$
END

changeBrightness
WHERE

grd1: $cardStatus = IN$
grd2: $brightnessUpdates < MAX\_BRIGHTNESS\_UPDATE$
THEN
act1: $brightness :\in BRIGHTNESS\_LEVELS$
act2: $brightnessUpdates := brightnessUpdates + 1$
END

confirmKBDString
WHERE
grd1: $cardStatus = IN$
grd2: $inputMode = KEYBOARD$
grd3: $confirmationStatus = FALSE$
grd4: $validationStatus = FALSE$
grd5: $0 \leq attempts \wedge attempts < MAX\_ATTEMPTS$
grd6: $newString = TRUE$
THEN
act1: $attempts := attempts + 1$
act2: $confirmationStatus := TRUE$
END

confirmSCRString
WHERE
grd1: $0 \leq attempts \wedge attempts < MAX\_ATTEMPTS$
grd2: $inputMode = SCREEN$
grd3: $cardStatus = IN$
grd4: $confirmationStatus = FALSE$
grd5: $validationStatus = FALSE$
grd6: $newString = TRUE$
THEN
act1: $attempts := attempts + 1$
act2: $confirmationStatus := TRUE$
END

checkStringCorrect
WHERE
grd1: $cardStatus = IN$
grd2: $confirmationStatus = TRUE$
grd3: $string = CORRECT\_PASSCODE$
grd4: $newString = TRUE$
THEN
act1: $validationStatus := TRUE$
act2: $confirmationStatus := FALSE$
END

checkStringWrong
WHERE

```
    grd1 : cardStatus = IN
    grd2 : confirmationStatus = TRUE
    grd3 : string ≠ CORRECT_PASSCODE
    grd4 : newString = TRUE
  THEN
    act1 : validationStatus := FALSE
    act2 : confirmationStatus := FALSE
    act3 : newString := FALSE
  END

  deliverBankNotes
  WHERE
    grd1 : cardStatus = IN
    grd2 : validationStatus = TRUE
    grd3 : deliveryStatus = FALSE
  THEN
    act1 : deliveryStatus := TRUE
    act2 : sum :∈ AMOUNTS \ {NO_MONEY}
    act3 : cardStatus := OUT
  END

END
```

B.5.2 Instantiation of EB4EB

Listing B.18: The instantiation of the calibration models

```
CONTEXT
  ATMmEBModel
EXTENDS
  ATMEnvironment
SETS
  Ev
CONSTANTS
  ATM
  init
  insertCard
  KBDString
  SCRString
  changeBrightness
  confirmKBDString
  confirmSCRString
  checkStringCorrect
  checkStringWrong
  deliverBankNotes
AXIOMS
```

Ev:
$partition(Ev, \{init\}, \{insertCard\}, \{KBDString\}, \{SCRString\},$
$\{changeBrightness\}, \{confirmKBDString\}, \{confirmSCRString\},$
$\{checkStringCorrect\}, \{checkStringWrong\}, \{deliverBankNotes\})$
axm17: $0 \leq MAX\_ATTEMPTS - 1$
axm18: $partition(BOOL, \{FALSE\}, \{TRUE\})$
axm19: $BRIGHTNESS\_MIN + 1 \leq BRIGHTNESS\_MAX$
axm21: $BRIGHTNESS\_MIN \leq BRIGHTNESS\_MIN + 1$
ATM:
$ATM \in Machine(STRINGS \times STRINGS \times STRINGS \times \mathbb{Z} \times$
$BOOL \times BOOL \times BOOL \times BOOL \times$
$INPUT\_MODE \times INSERTION\_STATUS \times$
$\mathbb{Z} \times \mathbb{Z} \times BOOL \times AMOUNTS, Ev)$
State:
$State(ATM) = STRINGS \times STRINGS \times STRINGS \times \mathbb{Z} \times$
$BOOL \times BOOL \times BOOL \times BOOL \times$
$INPUT\_MODE \times INSERTION\_STATUS \times$
$\mathbb{Z} \times \mathbb{Z} \times BOOL \times AMOUNTS$
Event: $Event(ATM) = Ev$
Init: $Init(ATM) = init$
Progress:
$Progress(ATM) = \{insertCard, KBDString, SCRString,$
$changeBrightness, confirmKBDString, confirmSCRString,$
$checkStringCorrect, checkStringWrong, deliverBankNotes\}$
Inv:
$Inv(ATM) = \{string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto$
$attempts \mapsto confirmationStatus \mapsto$
$validationStatus \mapsto deliveryStatus \mapsto$
$isStringVisible \mapsto inputMode \mapsto$
$cardStatus \mapsto brightness \mapsto$
$brightnessUpdates \mapsto$
$newString \mapsto$
$sum \mid$
$string \in STRINGS \wedge$
$virtualNumpadRegister \in STRINGS \wedge$
$keyboardRegister \in STRINGS \wedge$
$attempts \in 0..MAX\_ATTEMPTS \wedge$
$validationStatus \in BOOL \wedge$
$deliveryStatus \in BOOL \wedge$
$confirmationStatus \in BOOL \wedge$
$isStringVisible \in BOOL \wedge$
$inputMode \in INPUT\_MODE \wedge$
$cardStatus \in INSERTION\_STATUS \wedge$
$brightness \in BRIGHTNESS\_LEVELS \wedge$
$brightnessUpdates \in \mathbb{Z} \wedge$
$newString \in BOOL \wedge$

$sum \in AMOUNTS \wedge$
$isStringVisible = FALSE \wedge$
$(string = virtualNumpadRegister \vee string = keyboardRegister)\}$

Thm: $Thm(ATM) = State(ATM)$

AP:

$AP(ATM) = \{string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto$
$attempts \mapsto confirmationStatus \mapsto$
$validationStatus \mapsto deliveryStatus \mapsto$
$isStringVisible \mapsto inputMode \mapsto$
$cardStatus \mapsto brightness \mapsto brightnessUpdates \mapsto$
$newString \mapsto sum \mid$
$string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto$
$attempts \mapsto confirmationStatus \mapsto$
$validationStatus \mapsto deliveryStatus \mapsto$
$isStringVisible \mapsto$
$cardStatus \mapsto$
$brightnessUpdates \mapsto$
$newString \mapsto$
$sum = EMPTY\_STRING \mapsto$
$EMPTY\_STRING \mapsto EMPTY\_STRING \mapsto$
$0 \mapsto TRUE \mapsto FALSE \mapsto$
$FALSE \mapsto FALSE \mapsto$
$OUT \mapsto 0 \mapsto FALSE \mapsto NO\_MONEY \wedge$
$inputMode \in INPUT\_MODE \wedge$
$brightness \in BRIGHTNESS\_LEVELS\}$

Grd: $Grd(ATM) =$

$\{e \mapsto (string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto$
$attempts \mapsto confirmationStatus \mapsto$
$validationStatus \mapsto deliveryStatus \mapsto$
$isStringVisible \mapsto inputMode \mapsto$
$cardStatus \mapsto brightness \mapsto$
$brightnessUpdates \mapsto$
$newString \mapsto$
$sum) \mid$
$(e = insertCard \wedge cardStatus = OUT) \vee$
$(e = SCRString \wedge 0 \leq attempts \wedge$
$attempts < MAX\_ATTEMPTS \wedge$
$inputMode = SCREEN \wedge$
$cardStatus = IN \wedge$
$newString = FALSE) \vee$
$(e = KBDString \wedge 0 \leq attempts \wedge$
$attempts < MAX\_ATTEMPTS \wedge$
$inputMode = KEYBOARD \wedge$
$cardStatus = IN \wedge$
$newString = FALSE) \vee$
$(e = changeBrightness \wedge$

$$
\begin{array}{l}
\quad brightnessUpdates \leq MAX\_BRIGHTNESS\_UPDATE \wedge \\
\quad cardStatus = IN) \vee \\
(e = confirmKBDString \wedge 0 \leq attempts \wedge \\
\quad attempts < MAX\_ATTEMPTS \wedge \\
\quad inputMode = KEYBOARD \wedge \\
\quad cardStatus = IN \wedge \\
\quad confirmationStatus = FALSE \wedge \\
\quad validationStatus = FALSE \wedge \\
\quad newString = TRUE) \vee \\
(e = confirmSCRString \wedge 0 \leq attempts \wedge \\
\quad attempts < MAX\_ATTEMPTS \wedge \\
\quad inputMode = SCREEN \wedge \\
\quad cardStatus = IN \wedge \\
\quad confirmationStatus = FALSE \wedge \\
\quad validationStatus = FALSE \wedge \\
\quad newString = TRUE) \vee \\
(e = checkStringCorrect \wedge confirmationStatus = TRUE \wedge \\
\quad string = CORRECT\_PASSCODE \wedge \\
\quad cardStatus = IN \wedge \\
\quad newString = TRUE) \vee \\
(e = checkStringWrong \wedge confirmationStatus = TRUE \wedge \\
\quad string \neq CORRECT\_PASSCODE \wedge \\
\quad cardStatus = IN \wedge \\
\quad newString = TRUE) \vee \\
(e = deliverBankNotes \wedge validationStatus = TRUE \wedge \\
\quad cardStatus = IN \wedge \\
\quad deliveryStatus = FALSE)\}
\end{array}
$$

BAP: $BAP(ATM) =$

$$
\begin{array}{l}
\{e \mapsto ((string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto \\
\quad attempts \mapsto confirmationStatus \mapsto \\
\quad validationStatus \mapsto deliveryStatus \mapsto \\
\quad isStringVisible \mapsto inputMode \mapsto \\
\quad cardStatus \mapsto brightness \mapsto \\
\quad brightnessUpdates \mapsto \\
\quad newString \mapsto \\
\quad sum) \mapsto \\
\quad (stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto \\
\quad attemptsp \mapsto confirmationStatusp \mapsto \\
\quad validationStatusp \mapsto deliveryStatusp \mapsto \\
\quad isStringVisiblep \mapsto inputModep \mapsto \\
\quad cardStatusp \mapsto brightnessp \mapsto \\
\quad brightnessUpdatesp \mapsto \\
\quad newStringp \mapsto \\
\quad sump)) \mid \\
\quad (e = insertCard \wedge cardStatusp = IN \wedge \\
\quad\quad inputModep \in INPUT\_MODE \wedge
\end{array}
$$

$$
\begin{array}{l}
\quad newStringp = FALSE \wedge \\
\quad sump = NO\_MONEY \wedge validationStatusp = FALSE \wedge \\
\quad string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto \\
\qquad attempts \mapsto confirmationStatus \mapsto deliveryStatus \mapsto \\
\qquad isStringVisible \mapsto brightness \mapsto \\
\qquad brightnessUpdates = \\
\qquad stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto \\
\qquad attemptsp \mapsto confirmationStatusp \mapsto deliveryStatusp \mapsto \\
\qquad isStringVisiblep \mapsto brightnessp \mapsto \\
\qquad brightnessUpdatesp) \vee \\
(e = SCRString \wedge virtualNumpadRegisterp \in STRINGS \wedge \\
\quad stringp = virtualNumpadRegisterp \wedge \\
\quad brightnessUpdatesp = 0 \wedge \\
\quad newStringp = TRUE \wedge confirmationStatusp = FALSE \wedge \\
\qquad validationStatusp = FALSE \wedge \\
\quad keyboardRegister \mapsto \\
\qquad attempts \mapsto deliveryStatus \mapsto \\
\qquad isStringVisible \mapsto inputMode \mapsto \\
\qquad cardStatus \mapsto brightness \mapsto \\
\qquad sum = \\
\qquad keyboardRegisterp \mapsto \\
\qquad attemptsp \mapsto deliveryStatusp \mapsto \\
\qquad isStringVisiblep \mapsto inputModep \mapsto \\
\qquad cardStatusp \mapsto brightnessp \mapsto \\
\qquad sump) \vee \\
(e = KBDString \wedge keyboardRegisterp \in STRINGS \wedge \\
\quad stringp = keyboardRegisterp \wedge \\
\quad brightnessUpdatesp = 0 \wedge \\
\quad newStringp = TRUE \wedge confirmationStatusp = FALSE \wedge \\
\qquad validationStatusp = FALSE \wedge \\
\quad virtualNumpadRegister \mapsto \\
\qquad attempts \mapsto \\
\qquad deliveryStatus \mapsto \\
\qquad isStringVisible \mapsto inputMode \mapsto \\
\qquad cardStatus \mapsto brightness \mapsto \\
\qquad sum = \\
\qquad virtualNumpadRegisterp \mapsto \\
\qquad attemptsp \mapsto \\
\qquad deliveryStatusp \mapsto \\
\qquad isStringVisiblep \mapsto inputModep \mapsto \\
\qquad cardStatusp \mapsto brightnessp \mapsto \\
\qquad sump) \vee \\
(e = confirmKBDString \wedge attemptsp = attempts + 1 \wedge \\
\quad confirmationStatusp = TRUE \wedge \\
\quad string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto \\
\qquad validationStatus \mapsto deliveryStatus \mapsto
\end{array}
$$

$$
\begin{array}{l}
\quad isStringVisible \mapsto inputMode \mapsto \\
\quad cardStatus \mapsto brightness \mapsto \\
\quad brightnessUpdates \mapsto \\
\quad newString \mapsto sum = \\
\quad stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto \\
\quad validationStatusp \mapsto deliveryStatusp \mapsto \\
\quad isStringVisiblep \mapsto inputModep \mapsto \\
\quad cardStatusp \mapsto brightnessp \mapsto \\
\quad brightnessUpdatesp \mapsto \\
\quad newStringp \mapsto sump) \vee \\
(e = changeBrightness \wedge brightnessp \in BRIGHTNESS\_LEVELS \wedge \\
\quad brightnessUpdatesp = brightnessUpdates + 1 \wedge \\
\quad string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto \\
\quad\quad attempts \mapsto confirmationStatus \mapsto \\
\quad\quad validationStatus \mapsto deliveryStatus \mapsto \\
\quad\quad isStringVisible \mapsto inputMode \mapsto \\
\quad\quad cardStatus \mapsto \\
\quad\quad newString \mapsto \\
\quad\quad sum = \\
\quad\quad stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto \\
\quad\quad attemptsp \mapsto confirmationStatusp \mapsto \\
\quad\quad validationStatusp \mapsto deliveryStatusp \mapsto \\
\quad\quad isStringVisiblep \mapsto inputModep \mapsto \\
\quad\quad cardStatusp \mapsto \\
\quad\quad newStringp \mapsto \\
\quad\quad sump) \vee \\
(e = confirmSCRString \wedge attemptsp = attempts + 1 \wedge \\
\quad confirmationStatusp = TRUE \wedge \\
\quad string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto \\
\quad\quad validationStatus \mapsto deliveryStatus \mapsto \\
\quad\quad isStringVisible \mapsto inputMode \mapsto \\
\quad\quad cardStatus \mapsto brightness \mapsto \\
\quad\quad brightnessUpdates \mapsto \\
\quad\quad newString \mapsto sum = \\
\quad\quad stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto \\
\quad\quad validationStatusp \mapsto deliveryStatusp \mapsto \\
\quad\quad isStringVisiblep \mapsto inputModep \mapsto \\
\quad\quad cardStatusp \mapsto brightnessp \mapsto \\
\quad\quad brightnessUpdatesp \mapsto \\
\quad\quad newStringp \mapsto sump) \vee \\
(e = checkStringCorrect \wedge validationStatusp = TRUE \wedge \\
\quad confirmationStatusp = FALSE \wedge \\
\quad string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto \\
\quad\quad attempts \mapsto deliveryStatus \mapsto \\
\quad\quad isStringVisible \mapsto inputMode \mapsto \\
\quad\quad cardStatus \mapsto brightness \mapsto
\end{array}
$$

$brightnessUpdates \mapsto$
$newString \mapsto$
$sum =$
$stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto$
$attemptsp \mapsto deliveryStatusp \mapsto$
$isStringVisiblep \mapsto inputModep \mapsto$
$cardStatusp \mapsto brightnessp \mapsto$
$brightnessUpdatesp \mapsto$
$newStringp \mapsto sump) \vee$
$(e = checkStringWrong \wedge validationStatusp = FALSE \wedge$
$confirmationStatusp = FALSE \wedge$
$newStringp = FALSE \wedge$
$string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto$
$attempts \mapsto deliveryStatus \mapsto$
$isStringVisible \mapsto inputMode \mapsto$
$cardStatus \mapsto brightness \mapsto$
$brightnessUpdates \mapsto$
$sum =$
$stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto$
$attemptsp \mapsto deliveryStatusp \mapsto$
$isStringVisiblep \mapsto inputModep \mapsto$
$cardStatusp \mapsto brightnessp \mapsto$
$brightnessUpdatesp \mapsto$
$sump) \vee$
$(e = deliverBankNotes \wedge deliveryStatusp = TRUE \wedge$
$cardStatusp = OUT \wedge$
$sump \in AMOUNTS \setminus \{NO\_MONEY\} \wedge$
$string \mapsto virtualNumpadRegister \mapsto keyboardRegister \mapsto$
$attempts \mapsto confirmationStatus \mapsto$
$isStringVisible \mapsto inputMode \mapsto$
$brightness \mapsto$
$brightnessUpdates \mapsto$
$newString =$
$stringp \mapsto virtualNumpadRegisterp \mapsto keyboardRegisterp \mapsto$
$attemptsp \mapsto confirmationStatusp \mapsto$
$isStringVisiblep \mapsto inputModep \mapsto$
$brightnessp \mapsto$
$brightnessUpdatesp \mapsto$
$newStringp)\}$

Ordinary: $Ordinary(ATM) = Ev$
Convergent: $Convergent(ATM) = \emptyset$
Anticipated: $Anticipated(ATM) = \emptyset$
Variant: $Variant(ATM) \in State(ATM) \to \mathbb{Z}$
check_Machine_Consistency: $check\_Machine\_Consistency(ATM)$
GetNextStatesRw:
$Get\_next\_states\_of\_evts(ATM)[\{changeBrightness\}] =$

$$
\begin{array}{l}
\quad BAP(ATM) \\
\qquad [\{changeBrightness\}] \\
\qquad\quad [Grd(ATM)[\{changeBrightness\}] \cap Inv(ATM)] \\
\textbf{END}
\end{array}
$$

B.5.3 Analyses

Listing B.19: The generation of the domain specific properties on the ATM models

$$
\begin{array}{l}
\textbf{CONTEXT} \\
\quad AnnotatedModel \\
\textbf{EXTENDS} \\
\quad EventTagOntology \\
\textbf{CONSTANTS} \\
\quad annotate \\
\quad vv \\
\textbf{AXIOMS} \\
\quad \textbf{annotate}: \ annotate = \\
\qquad (\{nonPreEmptive\}\times \\
\qquad\quad \{changeBrightness, checkStringCorrect, \\
\qquad\qquad checkStringWrong, deliverBankNotes\})\cup \\
\qquad (\{internal\}\times \\
\qquad\quad \{changeBrightness, checkStringCorrect, \\
\qquad\qquad checkStringWrong, deliverBankNotes\})\cup \\
\qquad\quad (\{bounded\} \times \{changeBrightness\})\cup \\
\qquad\quad (\{inputByKeyboard\} \times \{KBDString\})\cup \\
\qquad\quad (\{inputByScreen\} \times \{SCRString\})\cup \\
\qquad\quad (\{inputByVoice\} \times \emptyset)\cup \\
\qquad\quad (\{input\} \times \{KBDString, SCRString\})\cup \\
\qquad\quad (\{textualConfirmation\} \times \{confirmKBDString\})\cup \\
\qquad\quad (\{visualConfirmation\} \times \{confirmSCRString\})\cup \\
\qquad\quad (\{auralConfirmation\} \times \emptyset)\cup \\
\qquad\quad (\{hapticConfirmation\} \times \emptyset)\cup \\
\qquad\quad (\{confirmation\} \times \{confirmSCRString, confirmKBDString\})\cup \\
\qquad\quad (\{interaction\}\times \\
\qquad\qquad \{KBDString, SCRString, confirmKBDString, confirmSCRString\})\cup \\
\qquad\quad (\{tag\}\times \\
\qquad\qquad \{insertCard, KBDString, SCRString, \\
\qquad\qquad\quad changeBrightness, confirmKBDString, \\
\qquad\qquad\quad confirmSCRString, checkStringCorrect, \\
\qquad\qquad\quad checkStringWrong, deliverBankNotes\}) \\
\quad \textbf{classInstances}: \ classInstances(eventOntology) = annotate \\
\quad \textbf{isWDOntology}: \ isWDOntology(eventOntology) \\
\quad \textbf{vv}: \ vv = \\
\qquad\quad \{KBDString \mapsto \\
\qquad\qquad\quad \{p \mapsto bright \mapsto ck \mapsto cs \mapsto v \mid
\end{array}
$$

$$
\begin{array}{l}
\quad\quad\quad\quad p \in STRINGS \times STRINGS \times STRINGS \times \mathbb{Z} \times \\
\quad\quad\quad\quad\quad BOOL \times BOOL \times BOOL \times BOOL \times \\
\quad\quad\quad\quad\quad INPUT\_MODE \times INSERTION\_STATUS \times \mathbb{Z} \wedge \\
\quad\quad\quad\quad bright \in \mathbb{Z} \wedge \\
\quad\quad\quad\quad ck \mapsto cs \in BOOL \times AMOUNTS \wedge \\
\quad\quad\quad\quad v = MAX\_BRIGHTNESS\_UPDATE - bright\} \} \cup \\
\quad\quad\quad \{SCRString \mapsto \\
\quad\quad\quad\quad \{p \mapsto bright \mapsto ck \mapsto cs \mapsto v \mid \\
\quad\quad\quad\quad\quad p \in STRINGS \times STRINGS \times STRINGS \times \mathbb{Z} \times \\
\quad\quad\quad\quad\quad\quad BOOL \times BOOL \times BOOL \times BOOL \times \\
\quad\quad\quad\quad\quad\quad INPUT\_MODE \times INSERTION\_STATUS \times \mathbb{Z} \wedge \\
\quad\quad\quad\quad\quad bright \in \mathbb{Z} \wedge \\
\quad\quad\quad\quad\quad ck \mapsto cs \in BOOL \times AMOUNTS \wedge \\
\quad\quad\quad\quad\quad v = MAX\_BRIGHTNESS\_UPDATE - bright\} \} \\
\quad \boldsymbol{vvbis}: \quad vv \in annotate[\{input\}] \rightarrow \mathbb{P}(State(ATM) \times \mathbb{Z}) \\
\quad \boldsymbol{isNecessarilyFollowedBy}: \quad isNecessarilyFollowedBy(\\
\quad\quad\quad ATM, \\
\quad\quad\quad eventOntology, \\
\quad\quad\quad \{input\}, \\
\quad\quad\quad \{bounded\}, \\
\quad\quad\quad \{confirmation\}, \\
\quad\quad\quad vv) \\
\textbf{END}
\end{array}
$$

RÉPUBLIQUE FRANÇAISE
Liberté
Égalité
Fraternité

Université de Toulouse

Titre : Génération automatique d'obligations de preuves paramétrée par des théories de domaine dans Event-B : Le cadre de travail EB4EB

Mots clés : Raffinement et preuve, Théories de domaines, Event-B, Annotation de modèles

Résumé : De nos jours, nous sommes entourés de systèmes critiques complexes tels que les microprocesseurs, les trains, les appareils intelligents, les robots, les avions, etc. Ces systèmes sont extrêmement complexes et critiques en termes de sûreté, et doivent donc être vérifiés et validés. L'utilisation de méthodes formelles à états s'est avérée efficace pour concevoir des systèmes complexes. Event-B a joué un rôle clé dans le développement de tels systèmes. Event-B est une méthode formelle de conception de systèmes à états avec une approche correcte par construction, qui met l'accent sur la preuve et le raffinement. Event-B facilite la vérification de propriétés telles que la préservation des invariants, la convergence et le raffinement en générant des obligations de preuve et en permettant de les décharger. Certaines propriétés additionnelles du système, telles que l'absence d'inter-blocage, l'atteignabilité ou encore la vivacité, doivent être explicitement encodées et vérifiées par le concepteur, ou formalisées à l'aide d'une autre méthode formelle. Une telle approche pénalise la réutilisabilité des modèles et des techniques, et peut introduire des erreurs, en particulier dans les systèmes complexes. Pour pallier cela, nous avons introduit un "framework" réflexif EB4EB, formalisé au sein de Event-B. Dans ce cadre, chacun des concepts d'Event-B est formalisé comme un objet de première classe en utilisant la logique du premier ordre (FOL) et la théorie des ensembles. EB4EB permet la manipulation et l'analyse de modèles Event-B, et permet la définition d'extensions afin de réaliser des analyses supplémentaires non intrusives sur des modèles, telles que la validation de propriétés temporelles, l'analyse de la couverture d'un invariant, ou encore l'absence de blocage. Ce framework est réalisé grâce aux théories d'Event-B, qui étendent le langage d'Event-B avec des éléments définis dans des théories, et aussi en formalisant de nouvelles obligations de preuves, qui ne sont pas présentes initialement dans Event-B. De plus, la sémantique opérationnelle d'Event-B (basée sur les traces) a été formalisée, de même qu'un cadre qui sert à garantir la correction des théorèmes définis, y compris les opérateurs et les obligations de preuve. Enfin, le cadre proposé et ses extensions ont été validés dans de multiples études de cas, notamment l'horloge de Lamport, le problème du lecteur/rédacteur, l'algorithme de Peterson, les distributeurs automatiques de billets (DAB), les véhicules autonomes, etc.

Title: Automatic generation of proof obligations parameterised by domain theories implementation in Event-B: The EB4EB Framework

Key words: Refinement and proof, Domain theories, Event-B, Model annotation

Abstract: Nowadays, we are surrounded by complex critical systems such as microprocessors, railways, home appliances, robots, aeroplanes, and so on. These systems are extremely complex and are safety-critical, and they must be verified and validated. The use of state-based formal methods has proven to be effective in designing complex systems. Event-B has played a key role in the development of such systems. Event-B is a formal system design method that is state-based and correct-by-construction, with a focus on proof and refinement. Event-B facilitates verification of properties such as invariant preservation, convergence, and refinement by generating and discharging proof obligations. Additional properties for system verification, such as deadlock-freeness, reachability, and liveness, must be explicitly defined and verified by the designer or formalised using another formal method. Such an approach reduces re-usability and may introduce errors, particularly in complex systems. To tackle these challenges, we introduced the reflexive EB4EB framework in Event-B. In this framework, each Event-B concept is formalised as a first-class object using First Order Logic (FOL) and set theory. This framework allows for the manipulation and analysis of Event-B models, with extensions for additional, non-intrusive analyses such as temporal properties, weak invariants, deadlock freeness, and so on. This is accomplished through Event-B Theories, which extend the Event-B language with the theory's defined elements, and also by formalising and articulating new proof obligations that are not present in traditional Event-B. Furthermore, Event-B's operational semantics (based on traces) have been formalised, along with a framework for guaranteeing the soundness of the defined theorems, including operators and proof obligations. Finally, the proposed framework and its extensions have been validated across multiple case studies, including Lamport's clock case study, read/write processes, the Peterson algorithm, Automated Teller Machine (ATM), autonomous vehicles, and so on.