Development of a Formal Verification Methodology for B Specifications using PERF formal toolkit. Application to safety requirements of railway systems.

Alexandra Halchin

THÈSE

En vue de l'obtention du

DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

Délivré par :

Institut National Polytechnique de Toulouse (Toulouse INP)

Discipline ou spécialité :

Informatique et Télécommunication

Présentée et soutenue par :

Mme ALEXANDRA HALCHIN

le vendredi 3 décembre 2021

Titre :

Development of a Formal Verification Methodology for B Specifications using PERF formal toolkit. Application to safety requirements of railway systems.

Ecole doctorale :

Mathématiques, Informatique, Télécommunications de Toulouse (MITT)

Unité de recherche :

Institut de Recherche en Informatique de Toulouse (IRIT)

Directeurs de Thèse :

M. YAMINE AIT AMEUR

M. NEERAJ KUMAR SINGH

Rapporteurs :

M. CHRISTIAN ATTIOGBE, UNIVERSITE DE NANTES

MME AMEL MAMMAR, TELECOM SUD PARIS

Membres du jury :

M. FRÉDÉRIC MALLET, INRIA SOPHIA ANTIPOLIS, Président

M. ABDERRAHMANE FELIACHI, RATP, Membre

M. DAVID BONVOISIN, RATP, Invité

M. JULIEN ORDIONI, RATP, Membre

M. NEERAJ KUMAR SINGH, TOULOUSE INP, Membre

M. SVEN LEGER, RATP, Invité

MI. YAMINE AIT AMEUR, TOULOUSE INP, Membre

Summary

The design of complex systems involves several design models supporting different analysis techniques for validation and verification purposes. These activities lead to the definition of heterogeneous modelling languages and analysis techniques. In this setting, meeting certification standards becomes a key issue in system engineering. Reducing heterogeneity due to the presence of different modelling languages can be addressed by providing an integrated framework in which involved modelling languages and techniques are formalised. In such a framework, checking global requirements fulfilment on heterogeneous models of a complex critical system becomes possible in many cases.

The work presented in this thesis addresses the problem of integrated verification of system design models in the context of transportation systems, in particular railway systems. It has been achieved in context of the B-PERFect project of RATP (Parisian Public Transport Operator and Maintainer) aiming at applying formal verification using the PERF approach on the integrated safety-critical models of embedded software related to railway domain expressed in a single unifying modelling language: High Level Languge (HLL). We also discuss integrated verification at the system level. The proposed method for verification of safety-critical software is a bottom-up approach, starting from the source code to the high-level specification.

This work addresses the particular case of the B method. It presents a certified translation of B formal models to HLL models. The proposed approach uses Isabelle/HOL as a unified logical framework to describe the formal semantics and to formalise the transformation relation between both modelling languages. The developed Isabelle/HOL models are proved in order to guarantee the correctness of our translation process. Moreover, we have also

used weak-bisimulation relation to check semantic preservation after transformations.

In this thesis, we also present the implementation of the defined transformation syntactic rules as the B2HLL tool. Moreover, we show the model animation process we set up to validate the B2HLL translator tool with respect to the formalised transformation rules we defined in Isabelle/HOL. This approach helps us to validate definitions, lemmas and theorems of our formalised specifications.

We have used the B2HLL tool to translate multiple B models, and we also show that when models are translated into this unified modelling language, HLL, it becomes possible to handle verification of properties expressed across different models.

Resume

Dans le contexte du développement des systèmes critiques industriels qui traduisent des exigences de sécurité et de sûreté de première importance, car elles impliquent des vies humaines, un processus de développement de haute qualité doit être mis en place. Ces systèmes ne cessent de se complexifier et sont contrôlés par des programmes logiciels. Afin d'éviter les aléas de l'erreur humaine lors de la conception d'un système, il faut s'assurer que le système vérifie les propriétés de sécurité nécessaires. Une solution pour aider à garantir la sécurité d'un système est d'utiliser les méthodes formelles.

La conception de systèmes complexes comprend plusieurs techniques de validation et de vérification. Ces activités conduisent à la définition de langages de modélisation hétérogènes et de techniques d'analyse. Dans ce contexte, le respect des normes de certification devient un enjeu clé de l'ingénierie des systèmes. La réduction de l'hétérogénéité due à la présence de différents langages de modélisation peut être abordée en fournissant un cadre intégré dans lequel les langages et les techniques de modélisation impliqués sont formalisés. Dans un tel cadre, la vérification du respect des exigences globales sur des modèles hétérogènes d'un système critique complexe devient possible dans de nombreux cas.

Les travaux présentés dans cette thèse abordent le problème de la vérification intégrée des modèles de conception des systèmes dans le contexte des systèmes de transport, en particulier des systèmes ferroviaires. Il a été réalisé dans le cadre du projet B-PERFect de la RATP visant à appliquer une vérification formelle en utilisant l'approche PERF sur des modèles critiques de logiciels embarqués du domaine ferroviaire exprimés dans un seul langage de modélisation : High Level Languge (HLL). Nous discutons également de la vérification intégrée au niveau du système. La méthode proposée pour la vérification des

logiciels critiques est une approche ascendante, qui va du code source à la spécification de haut niveau.

Ce travail porte sur le cas particulier de la méthode B. Il présente une traduction certifiée des modèles formels B en modèles HLL. L'approche proposée utilise Isabelle/HOL comme cadre logique unifié pour décrire la sémantique formelle et formaliser la relation de transformation entre les deux langages de modélisation. Les modèles Isabelle/HOL développés sont prouvés afin de garantir la correction de la traduction, en vue d'une validation formelle. De plus, une relation de bisimulation faible a été utilisé pour démontrer l'équivalence sémantique entre le langage source B et le langage cible HLL.

Dans cette thèse, nous présentons également l'implémentation des règles syntaxiques de transformation dans le prototype d'outil B2HLL. De plus, nous montrons le processus d'animation de modèle que nous avons mis en place pour valider l'outil de traduction B2HLL par rapport aux règles de transformation que nous avons définies, formalisées dans Isabelle/HOL. Cette approche nous aide à valider les définitions, les lemmes et les théorèmes de cette formalisation.

Nous avons utilisé l'outil B2HLL pour traduire plusieurs modèles B, et nous montrons également que lorsque les modèles sont traduits dans ce langage de modélisation unifié, HLL, il devient possible de gérer la vérification des propriétés exprimées à travers différents modèles.

Acknowledgements

My deepest gratitude goes to my supervisors Yamine Ait-Ameur, Neeraj Singh, Julien Ordioni and Abderrahmane Feliachi for their constant encouragement and guidance.

I would like to thank Yamine for pushing me further than I thought possible, for his patience, for his insightful and useful comments, valuable suggestions and direction. Neeraj's thoughts and suggestions have always been a blessing. I thank Neeraj for his availability, support, help and discussions on the world of research and industry.

I would like to thank Abderrahmane for his appropriate remarks, his perspective and for the long discussions that aided my understanding of formal methods, especially with Isabelle/ HOL, all of which contributed to the greater quality of this work. I thank Julien for his availability and support as I worked on my thesis and for the interesting discussions that aided my understanding of railway systems and safety verification techniques in an industrial setting.

I would like to give special thanks to Amel Mammar and Christian Attiogbe for reviewing this thesis. Thank you for reading this document and for your useful feedback and constructive recommendations. I would also like to thank Frederic Mallet, David Bonvoisin and Sven Leger for agreeing to be part of the defense committee.

I would like to thank Marc Pantel, Xavier Crégut and Katia Jaffresrunser for giving me the opportunity to teach and for the many interesting discussions. I would like to thank Annabelle Sansus and Sylvie Armengaud, who were always friendly and efficient and helped me in all administrative matters.

I would like to thank all the staff of ENSEEIHT and IRIT, the ACADIE team and especially the PhD students for the stimulating discussions.

I would like to thank my colleagues from RATP/QS and especially from AQL for the warm welcome, the good atmosphere and the very enriching exchanges in terms of personal and professional development. A special thank you to the TSL team.

I would like to express my deep gratitude to everyone who has supported me throughout this experience, especially my friends and family. There are far too many people I would like to thank for their encouragement, love, and for challenging me. A particular thank you to my parents who have always believed in me.

Finally, I would like to thank the person who has always been by my side, who has cheered me up in the challenging times, and who has rejoiced in my success, thank you for being there.

Contents

Part I

Background

Chapter 1

Introduction

Contents

1.1 Introduction

Nowadays, it is well known that the development of complex industrial systems involves both hardware and software, and such complex systems require high-quality development processes. In fact, such systems need to set up robust testing and verification protocols when working with critical applications such as transportation, aviation, medical, etc. Such systems are said to be critical. The critical adjective denotes that a system malfunction can have dramatic consequences for people, significant material damage, significant economic losses, or serious environmental consequences. Formal methods have been widely accepted and used in rigorous development process for verification and validation of safety critical systems.

In the development of a complex system, several stakeholders are involved in a single task or multiple tasks associated with different development processes of the system to be developed. Each of these development processes involves multiple development activities and models that are shared among all stakeholders. A consequence of the involvement of many

stakeholders in such developments is *h*eterogeneity. Indeed, each stakeholder can set up a number of modelling techniques, programming languages, design processes, validation and verification procedures, etc. Each stakeholder is responsible for delivering the components (hardware or software) he is in charge of. Then, the main issue remains in the global verification and validation, of the whole complex system. To solve this issue, one solution consists in imposing a standardised approach based on shared processes and languages.

This thesis topic is relevant to the larger context of railway safety, specifically the use of formal methods to evaluate and verify safety requirements in the context of safety critical systems. For safety critical systems, this validation and verification process is part of the design process. More specifically, we are concerned with the validation and verification of systems developed by various stakeholders using their own modelling languages and development processes. Our work addresses handling of *h*eterogeneity in large industrial systems developments. As automated systems become more complex, evaluating their structural and behavioural properties becomes increasingly difficult.

1.2 Industrial Context

The research presented in this thesis was conducted in collaboration with IRIT (Institut de Recherche en Informatique de Toulouse) and RATP (Régie Autonome des Transports Parisiens). RATP operates one of the most complex urban multi-modal public transportation networks in the world. In the Parisian region, its network includes 16 metro lines, 2 RER (inter-city trains) lines, 7 tramway lines and more than 300 bus lines; transporting an average of 10 Million passengers each day. RATP has built, throughout the years, a rich expertise not only in operating transportation networks but also in the engineering of railway transportation systems. This expertise made RATP one of the world pioneers in metro automation and one of the experts in automating existing lines.

Growing transportation capacity demand coupled with continued advances in computer technology accelerate the obsolescence of existing systems. These factors, added to the improvement and modernisation desires, have led RATP to upgrade its network by adopting integrated and upgradeable solutions, through partially or fully automated transportation systems. RATP deals with CBTC (communication based train control) and interlocking systems. The coexistence of these different systems brings additional difficulties, particularly related to the safety assessment of the railway system and depending on the automation

level of this system. One major concern of RATP is to ensure the safety of any deployed system on the network during all the project phases. In order to guarantee a better and more extensive safety analysis of the railway software systems, RATP's engineering department relies on rigorous verification methodologies based on formal methods.

PERF. RATP has been involved for several years in applying formal verification techniques to assess safety of railway systems that gave rise to a formal verification methodology named PERF (Proof Executed over a Retro Engineered Formal Model) [1], *d*esigned to be applicable to any software system independently of their development processes and languages. The approach allows to develop and analyse all the product component models translated in a single shared PERF pivot modelling language equipped with efficient formal verification procedures. This pivot language, HLL [2], is a synchronous data-flow language, close to Lustre[3], allowing to specify both system behaviour and safety properties together. *T*his translation shall be sound, i.e. semantic preserving. Once models translation is achieved, then the obtained shared models can be used for (integrated) verification and validation purposes. By taking the source code of the developed software as the verification target, it ensures a complete language-agnostic and non-interference with the supplier of the software that drastically reduces any bias. Moreover, maintaining multiple validation techniques in different domains may be expensive in particular when automated assistance is not available.

Today, at RATP, more than 50% of safety-critical software for CBTC control/command systems is developed using B method. Historically, RATP supported the development of the B method among railway manufacturers in the 1990s with the METEOR project. Techniques based on formal proof and refinement using the B method, defined by J.-R. Abrial [4] have shown their great potential for improving the quality of the software produced and the associated development processes. The B method is based on set theory and proposes incremental development from specification to code, enabling the production of correct by construction software.

PERF and B. Some of the RATP suppliers use the B method for formal design and verification of their critical software. Despite the use of formal methods, vendors cannot incorporate all the safety requirements that the software must meet into their model due to technical and time constraints and to the difficulty to express system requirements. Therefore, when this method is actually implemented in industrial projects, it is rarely

possible to demonstrate compliance with all the safety requirements for the software.

As a result of several years of experience with this method, the observation that bugs can easily be introduced when translating the informal software specification (document set) into the formal specification in B has been made. It is the starting point of B models.

While the B method ensures that the implementation is correct with respect to the software specification, it does not guarantee that the algorithms (encoded behaviour) themselves are correct with respect to the system-level requirements. Therefore, analysing the system and proving the key properties that ensure the correct and safe functioning of this system is crucial when developing or extending the algorithms of CBTC systems.

Consequently, the RATP methodology for checking the conformance and completeness of the formal specification of B software against informal documents is manual. It is based either on analysis of documentation produced by the manufacturer (who is responsible for delivering the system and software) or on critical reading of code to evaluate the software developed in B. **Currently, this software cannot be evaluated using the PERF method and the PERF workshop because there is no way to translate the B source code to the target language of the PERF workshop.**

Therefore, in order to improve evaluation methods (manual verification quickly reaches its limits when dealing with complex models), to bring the analysis of B software to the same level as the analysis of other critical software at RATP, and to unify validation techniques for critical software, a theoretical and practical study of the feasibility of developing a B to HLL translator was carried out as part of this thesis.

1.3 Objectives of the thesis

In this thesis, we focus on the *c*orrect by construction B method [4] to develop software systems by refining a high level specification and to guarantee the correctness of the given safety system requirements.

RATP collects a set of heterogeneous models, seen as black-boxes, that are validated by each stakeholder. A rigorous standard procedure for black box verification and validation for heterogeneous models collected from different stakeholders is set up. In this process, stakeholders receive a set of requirements and produce software components satisfying these requirements, modelled and verified using their own techniques. We demonstrate that a

robust approach is given by systematic modelling techniques to enable verification and validation activities. In other words, all the stakeholders still proceed with the design of system models using their own modelling and verification techniques, and each produced model is translated into the HLL modelling language in the PERF integrated verification framework. The B method [4] is one of these modelling and verification techniques. At this level, the question of preserving and aligning the semantics of the different modelling languages arises. The answer to this question can be provided by means of a formalised certification procedure to ensure that the semantics of the source model, expressed in B, is preserved by the target model expressed in HLL. For this purpose, a proof assistant can be used. We have chosen to use the Isabelle/HOL [5] proof assistant tool.

To investigate the applicability of PERF on software systems developed using the B method [4], the B-PERFect project was initiated by RATP. The concept behind the B-PERFect initiative is not to substitute formal verification process of B, but to propose an additional method to be used for an unbiased internal safety assessment. Here, the objective is to enrich PERF in the handling of B models. In addition to the classical safety assessment entailed by B, the B-PERFect framework offers the ability to check additional properties on B models integrated with the other models produced by other stakeholders. This process does not question the proof process of B. However, it may eventually reveal possible flaws in the initially stated safety requirements. The proposed method for the verification of safety-critical software is a bottom-up approach from the source code to the high-level specification.

Following the defined process, the B models are automatically translated into HLL models. Since this approach is based on a model transformation tool, semantic preservation and thus translator's certification are key and vital issues. An approach using a proof assistant is set up for this purpose.

In summary, in this thesis we focus on :

- Extending the PERF approach so that it can take into account the B language, which is widely used in the development of railway systems. To this end, the interpretation in HLL of the concepts inherent in B language, such as preconditions, postconditions, invariants, data structures, substitutions, etc., is of central importance.

- Ensuring the semantic adequacy of the interpretation of the B-models in HLL. Formalised arguments will be provided here to show the preservation of the interpretation

of these various constructions and their compositions. The results of this work are used to certify and/or qualify the developed tools.

- Develop a prototype that transforms the concepts and constructs of the B language to the HLL language, based on the transformation rules that emerge from the work done to reach the first objective.

The ultimate goals of this thesis are to demonstrate clearly the benefits of an integration verification environment, PERF, and to bridge the gap between the system specification and low level implementations.

1.4 Contributions

This work motivates and presents a translation from B language to HLL to verify safety properties of critical software independent of its construction. This work allows to extend the PERF approach and shows transformation from imperative to synchronous modelling languages.

This work is composed of the following steps:

- **Definition of transformation rules.** In order to create a translation proposal, the main elements of the two languages are identified. Rules for the translation of B concepts and constructs into HLL are defined in [6]. The defined transformation handles the *IMPLEMENTATION* level of the B language, corresponding to the proven imperative programs, with terminating loops (existing variants), and refers (with SEES and IMPORTS clauses) to other programs defined in externally defined B models.

- **Certification of the translation.** In order to preserve the formal developments, the transformation process must be correct and consistent. This phase of generating HLL models from B models is critical because a faulty translator may produce a non-conforming program, ruining the formal verification cycle and the whole approach. For this reason, a study was conducted to ensure the semantic adequacy of the HLL interpretation of B models, [7]. We have built a certified transformation of B models implementation to HLL models for verification purposes. The defined certification process first expresses the semantics of both B and HLL models together with the defined transformation in a single setting and second proves that the semantics is preserved by transformation. Isabelle/HOL is used to support this certification process.

- **Development of a B2HLL translator prototype.** This study of the transformation of the B language to the HLL language led to the development and production of a translator prototype. This prototype contains the translation rules we defined. The definition of a methodology to certify and/or qualify the tool produced was carried out by means of model animation[8, 9]. This approach is exemplified on a case study.

This thesis contributions are:

- providing a complete description of the transformation process from B implementations, corresponding to the programming level, to HLL models
- addressing the formalisation, in Isabelle/HOL, of the semantics of both B and HLL as state-transitions systems
- formalising and proving, within Isabelle/HOL, the equivalence theorem based on a bi-simulation relationship
- showing the interest of formal model animation for validation purposes.

1.5 Thesis Outline

This section gives references to later parts of the manuscript that deal with specific topics. The first part of the thesis, consisting of the second chapter, gives the general context of this work and presents the state-of-the-art on the subject. The second chapter describes the different methods and tools used to build the methodology we present. We introduce the context of safety-critical systems and give an overview of the application of formal methods for validation and verification of these systems. Furthermore, concepts of B and HLL modelling languages are presented. An overview of techniques for certifying a transformation is given.

The second part of this thesis deals with our contribution to the verification of safety-critical systems in a heterogeneous context. This part consists of four chapters. Chapter 3 describes our motivation for this project and presents our methodology for software safety verification and our proposal for supporting certification in this domain. Chapter 4 describes the transformation of a system developed using the B-method into HLL for the purpose of system verification. In Chapter 5, we propose a certification process for the translation principles mentioned in Chapter 4. The Isabelle/ HOL formalisation and proof of semantic

equivalence is presented. Chapter 6 deals with the implementation of this approach, the B2HLL tool. It proposes to apply and validate our approach on a case study. At last, the validation of the translator implementation is discussed using animation techniques.

This thesis ends with a general conclusion that highlights the main contribution of our work and provides some research perspectives.

1.6 Publications related to the thesis

- Alexandra Halchin, Abderrahmane Feliachi, Neeraj Kumar Singh, Yamine Aït Ameur,and Julien Ordioni. B-perfect - applying the PERF approach to B based system developments. In RSSRail, pages 160–172, 2017. [6]
- Alexandra Halchin, Yamine Ait Ameur, Neeraj Singh, Abderrahmane Feliachi, and Julien Ordioni. Certified Embedding of B Models in an Integrated Verification Framework (regular paper). In International Symposium on Theoretical Aspects of Software Engineering (TASE 2019), Guilin, Chine, 2019. [7]
- Alexandra Halchin, Neeraj Kumar Singh, Yamine Aït Ameur, Julien Ordioni, and Abderrahmane Feliachi. Validation of Formal Models Transformation through Animation. In KMOTS Worskshop 2019.[8]
- Alexandra Halchin, Yamine Aït Ameur, Neeraj Kumar Singh, Julien Ordioni, and Abderrahmane Feliachi. Handling B models in the PERF integrated verification framework: Formalised and certified embedding.Sci. Comput. Program., 196:102477,2020. [9]

Chapter 2

State of the Art

Contents

This chapter presents the required scientific background. This thesis focuses on using formal languages to verify complex systems that are both safety-critical and heterogeneous. To accomplish this, we address the particular case of safety-critical systems developed in a state-based language and verify them using a synchronous data flow language. First, the definition of safety critical systems is given, followed by an overview of industry standards. Second, formal methods used for safety verification are presented. Further, the background of formal methods used in our work and their industrial application is discussed. Finally, a review of the literature on scientific and technical related topics is presented.

2.1 Safety Critical Systems

A system is an organised collection of elements (or subsystems) that are highly integrated to accomplish an overall goal [10]. These elements include components (hardware, software, firmware), processes and other support elements. The system has various inputs, which go through certain processes to produce certain outputs, which together, accomplish the overall desired goal for the system. It's referred to as a safety critical system when any system failure or malfunction may have disastrous consequences.

Transport, nuclear, defence, finance and healthcare are major fields where safety critical systems are involved. The stakeholders of such systems are constantly looking for more cost-effective ways to cope with the massive increase in size and complexity, while still ensuring system safety. As a result, the safe behaviour of safety critical systems must be scrupulously validated. Efforts were made to mitigate risk, remove or reduce it so that acceptable levels of safety could be achieved.

In these fields, regulatory constraints must be respected in the development process of a system. Companies must receive approval from a relevant authority that the system they are developing is acceptably safe to operate in accordance with the applicable assurance standards such as CENELEC EN-50128 [11] for railway or DO-178C [12] for avionics. The purpose of this industry standards is to reduce the number of risks that might be introduced in the development process, imposing constraints on how the system should be developed and verified according to the criticality of system application. All of these standards cover their subject's entire life cycle, from specification to end-of-life treatment [13]. The focus of this research, however, is on the verification phase of safety-critical systems.

The subject of our study focuses on the railway context. Railway systems generally aim to provide on-time, efficient and, above all, safe train services. A reliable command and control system is required to ensure that the train can safely travel. In general, the railway topology consists of a series of interconnected elements protected by signals that transmit information to the trains. The safety of a train is ensured by the fact that its path may only be established if it does not conflict with the occupation of another train.

The development of software for safety critical railway systems should enable confidence in the system; therefore, the software must be developed so that it is free from design flaws which could cause catastrophic failure [14]. Although software does not cause loss of life directly, it can control some equipment that can cause serious injury, such as on-board train

equipment. Therefore the development process must follow the set of norms and methods described by the CENELEC [1] standard.

The CENELEC EN-50128 [11] standard specifies a wide range of methods that can be used for the development of railway control and protection systems. It defines the degree of rigour in the development process based on Safety Integrity Levels (SILs), which are used to determine the system's acceptable failure rate. There are four SIL levels defined, with SIL4 being the most reliable and SIL1 being the least. The production method and safety life cycle management are among the quantitative and qualitative considerations that go into determining a SIL. In this context, it is necessary to demonstrate that the system meets its safety requirements, that the development process complies with the standards, and that the tools that contributed to the system design or its verification have been qualified in terms of their use and contribution to global safety. Without imposing a solution, EN50128 provides some guidelines on software development methodology and accepted techniques in relation to the established SIL. Indeed, despite CENELEC directives that strongly recommend the use of formal methods that are mathematical methods used to prove that a system meets the safety requirements, verification using a testing approach can be accepted.

Software verification has traditionally been accomplished by two methods: reviewing and testing. However, testing approaches don't allow an exhaustive verification. Given the limitations of manual activities, we review in next Section some verification methods of safety critical systems based on formal methods.

According to [15, 16], the requirements and specification phases are the most error-prone phases in the development of safety critical systems. Therefore, it is critical to validate and verify safety-critical systems from the beginning of their development. While the use of formal methods and provers guarantees model consistency, determining whether the specification models the desired behaviour is more difficult. However, model completeness and quality remain critical issues that need to be addressed.

2.2 Methods for Safety Verification

The CENELEC standard highly recommends the use of formal methods to demonstrate that the system meets the safety requirements. Several formal methods and modelling techniques

[1] European Committee for Electrotechnical Standardization : https://www.cenelec.eu/

are listed in the standard, such as CSP [2], CCS [3], LOTOS [4], Temporal Logic, VDM [5], Z method [6], B method and model checking. Formal proof is also highly recommended as a verification activity.

Various formal methods can be used to validate or verify the safety of safety-critical systems. These methods are generally based on set and type theories and predicate logic. Formal methods are extensively discussed in the literature and several classifications have been proposed. A general classification of formal methods can be made based on their foundations and theoretical background. Here we recall some of these formal methods:

- Process algebras, describe the behaviour of concurrent processes based on the interaction between them using a set of algebraically defined operators. For example languages such us CSP [17], CCS [18] and LOTOS [19].

- Logic-based methods, where logic is used to describe specification of system or program behaviour. For example, temporal logic [20].

- State-based methods, give an explicit definition of systems states and transitions that transform the state. Examples of such methods are Z [21], VDM [22], B method [4] and Event-B [23]

- Higher-order logic based methods, their particularity is that they describe the system and the associated verification procedure in a uniform setting. Among these methods we can cite Isabelle/HOL [5], Coq [24], PVS [25].

These methods have been associated to several tools used to validate and verify formal models, ensuring that a design conforms to its specification. Formal verification provides methods and techniques to mathematically prove the correctness of a system, i.e. to prove that the model of the system satisfies the properties required by the user, such as Theorem Proving [26, 27] and Model Checking [28, 29].

[2] Communicating Sequential Processes
[3] Calculus of Communicating Systems
[4] Language Of Temporal Ordering Specification
[5] Vienna Development Method
[6] Zermelo

2.2.1 Theorem Proving

Theorem proving relies on higher order logic and mathematical structures to construct specifications describing system's behaviour and a proof system to prove expressed properties. This method can be applicable to projects of any complexity. However, in order to perform verification, the user must have a high level of knowledge about logic notions and about the design being verified. The user must perform the proving in a systematic way by developing the formulas, feeding them into the tool, and analysing the results. For this kind of proving process a certain degree of automation is possible thanks to available automatic provers depending on the formalism used to model the system and to describe the properties [30]. The B method uses AtelierB tool based on automatic theorem proving.

Isabelle/HOL. Isabelle is an automated theorem prover. This interactive higher order logic theorem prover is a generic proof assistant that can be instantiated with several logics like First-Order Logic (FOL), Zermelo-Fraenkel set theory (ZF) or Higher Order Logic (HOL). It relies on a core and small theorem prover namely Logic of Computable Functions LCF [31] developed on top of the ML language. Isabelle proofs are encoded in the structured proof language Isar [32] providing human and machine understandable representation of proofs. In addition, Isabelle is also equipped with animation functionalities to run executable specifications in various functional languages. This proof assistant proved useful in formalising mathematical proofs for formal verification of computer systems and in proving of programming languages properties.

In the style of LCF [31], the Isabelle/HOL proof assistant is a generic interactive theorem prover obtained by instantiation of Isabelle with Higher-Order Logic (HOL) [5]. Isabelle/HOL can be seen as a specification and verification environment with the capability of modelling systems and proving logic based system properties. In general, when modelling systems and system properties in Isabelle/HOL, a modelling part and a proof part are defined.

The modelling part relies on functional programming languages (ML in the case of Isabelle/HOL). Basic type declaration is *typedecl*('t_1, 't_2, ...) T_{new}, where 't_i are possible type parameters and T_{new} is a new defined type. Other type constructors are available: $t_i \times t_j$ for product and $t_i \Rightarrow t_j$ for function maps. *Terms* are formed using λ-calculus expressions. Moreover, it also offers operators like *condition* ($if\ b\ then\ e\ else\ e$), *let* ($let\ x =\ e\ in\ a$) and *case* ($case\ e\ of\ p\ \Rightarrow\ a|...$) representing basic constructs in a functional programming languages and thus offering powerful modelling capabilities.

The proof part deals with proof systems and proofs. For proofs, Isabelle combines functional programming languages such as HOL and the Isar language to manipulate generic proof procedures. Moreover, already proved theorems in Isabelle can be checked and can be used in the development of other proofs interactively (e.g. used as lemmas). In addition, proof (or inference) rules can be defined from proved theorems and used in a the development of a proof. Existing and user defined tactics can be described in Isabelle/HOL. Tactics are defined to reduce series of inference rules applications (e.g. using choice, loop) into a single proof or inference rule. In Isabelle/HOL, tactics may be written as ML statements combining proof rules or already existing tactics. Examples of tactics and proof rules are inference rules, already proved statements, induction statement, variables introduction, hypotheses removal, try of inference rules, loop on the application of inference rules, etc. In Isabelle/HOL, we can organise the already proved lemmas and theorems in modules called *theories*. There exists a large amount of theories with a collection of definitions, new data types constructs and recursive functions.

2.2.2 Model Checking.

Model checking [28, 33] is a technique for verification and debugging, widely used in different fields. This technique allows to verify properties against a model of a system based on a process of exhaustive state space exploration. The safety properties represented by a boolean formulas are verified to see if they hold in every system trace. Properties such as deadlock freedom, invariants or safety, expressed as temporal logic formula, are checked to determine if they are satisfied by the system, otherwise a counterexample falsifying the property is shown [29]. Counterexamples represent illegal paths of the system and allow bug correction, this feature can be seen as one of the main advantages of model checking.

Research on efficient algorithms and techniques allowed to optimise state space exploration and further to apply model checking in the verification of realistic industrial systems. It is the case for RATP systems. Indeed, state space explosion [34] is a limitation of this technique because the state space increases drastically with the number of system variables. Explicit state, BDD [7], SAT [8] [35, 36], SMT [9] [37], and other technologies exist to explore system traces, each resulting in a distinct category of model-checkers. Examples of such

[7] Boolean Decision Diagrams
[8] Boolean Satisfiability problem
[9] Satisfiability Modulo Theories

model checkers are NuSMV2 [38], nuXmv [39], ProB [40]. Model checking usual use cases are safety proof, system debugging and equivalence checking [41]. Since the analyses are often automated and a posteriori, they can be integrated into existing production processes.

2.2.3 Formal methods for safety-critical system in railway industry

Despite all the variety of formal methods we have presented in the above Section, not all of them have reached a level of maturity and application in an industrial setting. The railway domain is one of the domains where formal methods have been used intensively for formal specification and verification activities, with numerous success stories documented in the literature [42]. Formal methods, such as the B method, are widely used in the French railway industry, and have been successfully applied to the verification of the safety-critical components of a metro system [43]. Our research focuses on the process of applying formal methods to safety-critical systems in the railway industry, as well as how to respond to specific constraints such as independent safety assessments of heterogeneous systems.

Generally, formal methods can be applied in two distinct cases to ensure the safety of critical railway systems:

- to validate the specification of the system. The main advantage of this formalisation is that it allows for the elimination of ambiguities that natural language may introduce. The behaviour of the systems described in the specification documents, as well as their safety properties, are modelled. Formalising safety requirements is primarily a manual engineering task that requires taking the general safety principles from technical documents and formalising them as precise requirements expressed in a logical programming language. Further, the specification is considered correct if it satisfies the required safety properties in all system states. As previously stated, formal specifications can be produced and further used for verification activities. This activity could also allow to identify the safety properties that the system must respect and provide them for global program correctness proof.

- to demonstrate the software's compliance with its specifications. This activity enables formally proving that the software complies with its safety requirements and, more broadly, its specifications, in other words, that the software does what it is supposed to do. Our research work is part of this category.

Several success stories have been described in designing critical systems using a formal development approach with B method such as the control system of the driverless Meteor line 14 in Paris or the VAL shuttle for Roissy Charles de Gaulle airport [43, 44, 45]. Since then, several railway system manufacturers have generalised the use of B as their solutions for building critical systems in a correct by construction manner.

2.2.4 Formal Methods at RATP

One of the RATP's guiding principles is to provide safe and reliable transportation service. RATP has to constantly adapt its verification methods to the evolution of its systems and growing complexity. In addition, RATP projects involve several subcontractors that use different development methods and languages. The resulting heterogeneity enables RATP to master all subcontractors methods and languages and to manage a complex assessment process. In light of these circumstances, the RATP engineering department must answer the following question: *How to ensure the safety of systems facing constraints such as complexity and heterogeneity?* .

Over the years, RATP has used a variety of formal methods and techniques, successful stories of their application in large-scale projects have promoted their use and increased confidence in these techniques [46]. In order to deal with heterogeneity difficulty, a unified verification approach, offering an *"ex post facto"* proof, is applied to each supplied product regardless of the subcontractor's development language or method. At RATP formal verification is used not only to improve safety but also as a way to reduce the time and cost of safety assessment by eliminating the need for safety testing for example.

One of the first application of formal methods in an RATP project dates back to the late eighties, when the Z method revealed a number of safety critical bugs in the SACEM [10] system (RER A), which had already passed the test campaign. SACEM is an automatic train protection system that regulates the speed of all trains on the line with the goal of increasing network traffic by 25%. A formal specification and verification process based on *a posterori* proof was used [47] allowing to put in evidence several anomalies before commissioning. The proof process was performed by software high-level experts without the use of any tools. This experience not only established the SACEM as a new safety standard

[10] Système d' Aide à la Conduite, à l'Exploitation et à la Maintenance - Driving, operation and maintenance assisting system

(zero unsafe behaviors detected after 30 years of operation), but also demonstrated formal proof's potential advantages over test-based approaches [46].

This successful application of formal methods led RATP to require the use of formal methods for all its safety-critical software systems suppliers. As a consequence, the development of the first driverless metro line in Paris (Line 14) in 1998 was supported using the formal B method. The safety of the system was proven by construction which helped to get rid of some testing phases while guaranteeing a better coverage. These projects represent the roots of the development and use of the B method in the French railway industry and in general. In addition, RATP, Alstom, and SNCF launched a project to industrialise a tool to support the B method, which resulted in the Atelier B [48] tool.

A posteriori formal proof

RATP cannot require anymore the use of formal methods because, according to the regulations, this would favour some suppliers over others. Despite the fact that the CENELEC standard strongly recommends the use of formal methods for the development of safety-critical software components, it does not prohibit the use of test-based processes. However, RATP continues to strongly advise all of its suppliers to use a formal development method.

RATP performs its own internal safety assessment of safety critical systems, independent of the development and verification processes carried out by system suppliers. According to RATP, using formal methods independent of the supplier usually reveals more bugs than simply verifying the supplier's testing campaign.

Since the early 2000s, RATP has collaborated with a different suppliers, employing a variety of development methods and languages. The resulting heterogeneity requires RATP's mastery of all supplier methods and languages, introducing a skill management challenge in the assessment process. The solution was to use a unified verification approach, referred to as a "ex post facto" proof, for the different projects, allowing formal verification to be applied regardless of the supplier's development language or method.

This situation was the starting point of the PERF [11] methodology and its supporting workshop [1]. RATP has thus developed and procured proof tools for its suppliers in order to encourage them to use formal methods. The technique has been used successfully on Thales, Ansaldo, and Alstom (ex-Areva TA) products, in charge of the Computer Based

[11] Proof Executed over a Retro engineered Formal model

Interlocking Lines 1, 4, 8, 12, the wayside, and the on-board equipment of CBTC [12] Lines 3, 5, and 9 and Line 13 projects [49, 50].

This *a posteriori* proof approach proved to be effective [51, 49]. PERF is now applied in every project, whenever it is possible, meaning essentially that the source language of the software is supported by the PERF workshop.

Several projects are still using the B method or language to develop safety-critical systems. This is unquestionably good news, but it complicates the independent assessment at RATP. Even though the formal verification performed by the B proof engines is reliable, the independent validation of the safety properties at RATP can only be accomplished through cross-reading. This technique, while time-consuming, may not be very effective as it is not exhaustive and cannot guarantee that there are no defects.

The object of this research, is to provide an independent alternative for the verification of the safety properties on systems developed using the B method. The PERF approach makes this verification non-intrusive and, if necessary, supports in the verification of the code generation process. It will also help, in the context of heterogeneous systems, to apply a unified verification across all system components.

PERF: an integration verification framework

The PERF method, along with its associated workshop, enables a posteriori safety assessment of critical software. In other words, formal proof is used in the ascending part of the V-cycle, after the design and development phase, to ensure that the software meets the expected safety properties. This verification technique can also be used when formal proof was not envisaged in the early stages of the system. It offers a framework for integrated model and program verification provided in different modelling and programming languages developed by different stakeholders.

Figure 2.1 depicts a general workflow of the PERF methodology. Next we describe the PERF verification process. The following elements are PERF's inputs:

1. a model of the software behaviour, the system's source code or model that is transformed into a formal HLL [2] model. The system's software is developed by RATP's suppliers based on requirements provided by RATP.

[12] Communication Based Train Control

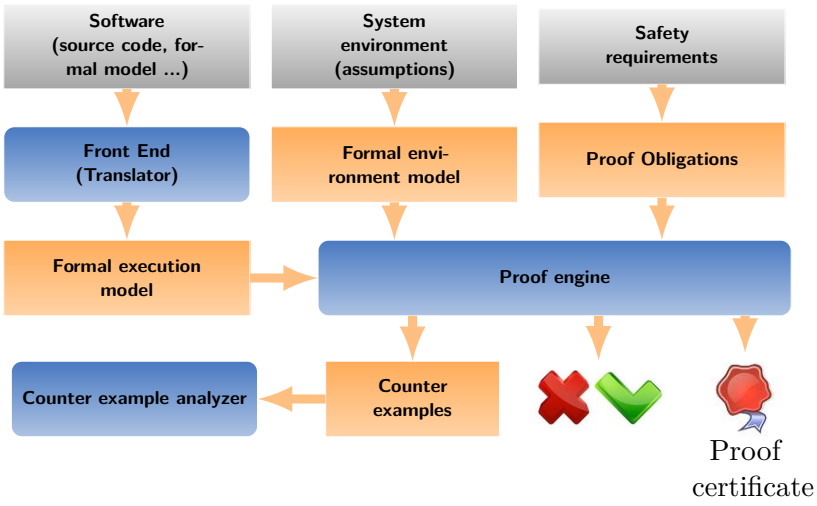

Figure 2.1: The PERF verification workflow

2. the safety properties, resulting from the formalisation of system requirements, are also expressed in HLL as HLL proof obligations (i.e. as first order logic formulas) to meet the global requirements and standard safety regime of RATP. This activity is realised by RATP's engineers.

3. an environment model to explore the behaviour of the software in it, described in HLL as constraints or assumptions. The RATP achieves this modelling step to express hypotheses about the environment in which the software evolves.

HLL [2] serves as a pivot modelling language of PERF. It is a formal verification language close to synchronous data-flow language LUSTRE [3], suitable to describe discrete-time sequential behaviours and to express temporal properties associated to this behaviours. The verification is then carried out on the model obtained as a result of the different transformations of source models on the one hand and the enrichment of the proof obligations by the safety engineers at RATP on the other hand.

The verification procedure associated to HLL is based on model checking and SAT-solving. When these verification tools run, if counterexamples are revealed by the proof engine, the corresponding scenario is analysed to understand the safety risk associated with this property violation. A complete PERF-related tool chain (translators, counterexample analysers, SAT-based proof engines) is available to perform such analysis.

A number of translators have been developed and integrated into PERF formal toolkit to support the different solutions of all RATP suppliers. Such translators provide a standardised

description of the intended source code in the PERF's pivot language HLL. The primary role of the translators is to provide a semantic-preserving formalisation of the software to be verified in HLL. PERF is actually applied in every project where translators are available to identify the possible bugs. Currently, PERF supports the automatic transformation, into HLL models, of several programming and modelling languages, such as C, Ada or Scade.

RATP includes a wide variety of signalling systems from different manufacturers, and the application of PERF on safety validation of these systems has clearly shown the value of these approaches in terms of increased quality of safety assessment and reduced cost of this activity. The high heterogeneity of suppliers' modelling languages is transparent to the RATP safety team (due to the use of a single HLL language), allowing them to focus on the HLL modeling, the properties to be verified, and the environment (as mentioned above) for evaluating critical systems. Nonetheless, depending on the modelling languages chosen by the suppliers, RATP may need to develop and enrich its methodology to take into account potential new languages.

B-PERFect. Currently, the B method is not supported by the PERF framework. Software systems developed using B are valid by *correct by construction* with respect to safety requirements. In order to harmonise the safety assessment methods at RATP, the B-PERFect project was initiated. Unlike top-down formal development techniques such as B method, there exist formal verification techniques that can be applied at the ascending phase, of the V-shaped development cycle, to asses the safety of critical software with respect to their high level specification. The PERF methodology with the HLL modelling language is one of such verification techniques.

Our research aims to integrate B models into the PERF framework and achieve additional guarantees regarding the encoding of invariant issued from the safety requirements. The idea behind the B-PERFect project is not to replace the formal verification process of B but to propose a verification alternative to be used for an internal independent safety assessment. In our approach we integrate both top-down and bottom-up verification approaches.

2.3 B Method

The B method is a formal method that supports correct by construction verification approach allowing engineers to build software with high guarantees of confidence. This is possible

because the verification is performed all along the development process through the use of refinement methodology and preservation of user-written invariants. It is based on first-order logic and set theory and handles a complete critical-software development process from specification to code [4]. The B method has proven its feasibility for large scale industrial applications, particularly in railway domain [43]. Major players in the railway domain, use the B method in the development of critical software applications.

Modelling. A B development process is layered. Each layer corresponds to an abstraction level and the refinement relationship provides a formal link between layers. Software specification is encoded in abstract components that represent the highest level of abstraction. The most concrete parts of a B model are the obtained implementations where only programming-like constructs are allowed [48].

In B, models are represented as machines. The concept of machine is analogous to the concepts of module and object in traditional programming languages. A B specification includes several *machines*. A B machine contains a set of variables (which may be expressed in terms of integers, Booleans, sets, relations or functions, among others and representing the state), invariant properties with respect to that variables (a state invariant expressed using first-order predicate logic), instance of other machines, an initialisation clause and operations acting on the defined variables (the transformations of that variables are expressed using substitutions).

Generally, B project models represent a state transition system in which the initialisation clause sets the initial values of variables and the operation clause specifies how variables are modified from one state to another. A machine describes the dynamic parts (states and transitions) of a model and the static parts (constants, invariants, properties). The invariant describes the safety properties of the model and it is specified using predicate logic.

In Listing 2.1 we give a schematic description of a B machine. Briefly these clauses mean:

- **MACHINE**, **REFINEMENT** or **IMPLEMENTATION** represent the component name definition which should be unique in a B project. MACHINE are the upper level of software development, describing the formal specification of a system. They represent the higher level of abstraction. REFINEMENT machines are intermediate steps in the refinement of abstract concepts derived from machines. IMPLEMENTATIONS are the most concrete representation of machine-level algorithms. They may be converted into a program written in a given programming language once they have been proven.

```
MACHINE mch_id1        /* Machine name */
REFINES mch_id2
IMPORTS mch_id3
SEES   mch_id4
SETS S,                /* Abstract sets */
     T = {a,b, ...}    /* Enumerated sets */
CONSTANTS C            /* Constants */
PROPERTIES R           /* Constants specification */
VARIABLES x            /* Variables */
INVARIANT Inv          /* Variables specification */
INITIALISATION Init
OPERATIONS Op1;        /* List of operations */
           Op2;
           ...
           Opn
END
```

Listing 2.1: Schematic description of a B machine

- **SETS** defines the sets that are manipulated by the specification. In implementation, sets must be explicitly introduced.
- **CONSTANTS** defines all the constants that are used in the machine.
- **PROPERTIES** are logical expressions that are satisfied by the constants described previously.
- **VARIABLES** is the clause where all the state variables of the described model are declared. Refinements can add new variables to enrich the described system and the specification model from higher levels of abstraction.
- **INVARIANT** clause describes the properties of the attributes defined in the clause $VARIABLES$, using first-order logic expressions. This clause describe various types of properties, such as typing, safety, and functional properties. The gluing invariant required by any refinement is an example of an invariant describing the link between abstract and concrete variables. The logical expressions described in this clause must hold after all state changes, the invariant must hold after initialisation, and state changes from operations must preserve the invariant.
- **INITIALISATION** clause allows to give initial values to the variables of the machine. The initial value must satisfy the invariant.

- **OPERATIONS** clause defines all the procedures to specify the desired behaviour of a model. Each operation is described by a set of actions that modify the state defined in the $VARIBALES$ clause.

Proof is required to verify the correctness of the high level software specification encoded in abstract machines, of the refinement steps and of the obtained implementation. A key concept in B formal developments is the encoding of invariants representative of high level safety requirements. A software component is said to be correct if all safety requirements are encoded by B invariant. In practice this is not really the case.

Refinement of B machines. Refinement is a relation that connects two models by expressing the enrichment of one model by another. The refinement technique is used to gradually introduce details into the specification. It is the process of transforming an abstract model into a concrete model specified in a subset of the B language: the B language that can be automatically translated into executable code [52, 53]. It starts with a high-level system-level specification and incrementally adds more concrete components of the system, such as software computations or physical variables. Typically, specifications are complex, so that global properties are distributed across different components. To arrive at an executable application, the set of data structures or non-deterministic elements of an abstract machine must be incrementally replaced by structures similar to those of programming languages, such as while loops. The implementation must be deterministic, contrary to abstract modules, parallel substitutions are not allowed, the type of the variables must be scalar, and they are written in a procedural style.

A key feature of the B method is that it explicitly distinguishes between language constructions needed for implementation and those only used for specification. Each kind of components has specific syntactic restrictions.

Composition. B language offers several structuring mechanisms that make it easier to modularise, compose components, decompose proofs and share the state. In addition to the previous features, there exist other operators to compose machines. The relations between machines are given by the following clauses: $INCLUDES$, $IMPORTS$, $SEES$, $USES$, $EXTENDS$, $PROMOTES$. Each clause defines a different level of access to other abstract machine's components.

$INCLUDES$ clause allows to put together state from several machines and gets read and write access to state variables from other machines. This clause allows to modify the state of another machine by using the included operations. The $IMPORTS$ operator connects implementations to other abstract machines by referring to the machines from which external operations are imported. It enables the creation of layered software and serves as a final composition primitive because the implementations are no longer refinable.

The $USES$ operator expands an abstract machine's data space by introducing data sharing between included machines. $SEES$ refers to the machine that defines relevant features(i.e., types) and can appear in any type of B machine. When this operator is used, data (e.g., sets, definitions, and variables) can be shared with read-only access, variables must not be modified by the seeing components.

There are two more clauses available: $PROMOTES$ and $EXTENDS$. The $PROMOTES$ operator enables the use of the definitions of operations provided by including machines without the need to redefine them in the current machine. $EXTENDS$ is a clause analogous to including machine instances and simultaneously promoting all of the operations of the included machine instances in abstract machines or refinements.

A whole description of all B clauses can be found in the B-Book [4]. Significant work [54, 55, 56, 57, 58] has been accomplished to explain B refinement and composition, these mechanisms that are applied to build structured specifications.

2.3.1 Semantics and verification process

The dynamic part of a B machine (i.e. operations and initialisation) is described using the Generalised Substitutions Language (GSL), which allows mathematical notations to be used to describe the transformation of state. Thus, the transition from a state before the execution of an operation, defined by a before predicate to a state reached after the execution of the operation defined by an after predicate, can be expressed. Each action defines a predicate, the "Before After Predicate" (BAP), that relates pre and post states. This predicate transformation is based on the weakest precondition calculus defined by Dijkstra [59]. Indeed, the transformation of a predicate P into a predicate Q by the substitution S, noting $Q = [S]P$, is equivalent to calculating the weakest precondition that ensures that S terminates and that the assertion P is true after the termination of S, one then says that the substitution S establishes the predicate P. Two predicates define generalised substitutions: trm(S), which is the required condition for substitution S to

terminate $trm(S) \Leftrightarrow [S]true$, and prd(S), which describes the relation between before and after states, $prd_v(S) \Leftrightarrow \neg[S]\neg(v' = v)$, where v, v' are respectively the values before and after the execution of the substitution S.

Proof Obligations (PO). The correctness of a B model (machine, refinement, or implementation) is established by proving POs, by showing that invariants are not violated by state transitions. POs are generated automatically based on calculus of substitutions. POs require to demonstrate that starting from a state that satisfies the properties of the machine, including constraints and constant properties in which the invariant is satisfied, this state remains satisfied even after substitutions from initialization as well as from operations have been executed. An initialisation is said to establish the invariant if the execution of the substitution $Init$ leads to a state that satisfies the invariant Inv. Similarly, the substitutions of an operation called from a state satisfying both the properties of the machine, the invariant and preconditions for its execution must lead to a state in which the invariant is true.

Proof obligations must allow to demonstrate that [60]:

- each invariant is preserved in a machine, it holds in observable states, states before and after operation calls.
- each operation simulates its corresponding abstract version. The proof activity concerning the refinement involves performing a set of static checks and proving that the refinement is a valid reformulation of the specification.
- each while loop is converging. States that each iteration of the loop decreases the variant and that the state changes preserve the global invariant.

B method has a proof system associated with it along with a set of proof rules on B constructs.

Tools. The success of B in the railway sector led to the creation and improvement of B method tools such as Atelier B [48], the main tool for B-based development. Included in this integrated development environment are editors, a proof obligation generator, automated provers, and other tools such as BART [61] an automatic refinement tool and code generators. AtelierB needed to be tested and validated to be useful for the development of safety-critical applications, and initially these tasks were carried out under the overall supervision of RATP [62].

Another well-known B-method tool is ProB [40, 63], an animator, constraint solver, and model checker that lets you instantiate models and validate if the specification produces the desired behaviour.

Development process. In the context of critical software development, the B method is used to specify only those components that perform treatments that may affect human and hardware safety. The formal development cycle of a software is divided in the following steps.

- The production of a series of documents that specify what is expected of the software, known as specification documents.
- Formalisation in B of the specification's contents, i.e. the requirements to which the software answers. This allows to obtain the abstract B model. A balance must be found between directly encoding certain safety properties and modelling them in terms of typing invariants.
- Refinement of the abstract model all the way to implementation. The required information is added to obtain a concrete model that can be translated into a compilable language (C, Ada).
- The proof activity ensures that the concrete model respects the abstract model and that both establish the safety properties encoded as invariants.
- Since the natural language documents are the entry point of the process, coherence and completeness between these documents and the formalised abstract B model must be ensured. This verification activity is usually realised in the industrial context by cross-reading and testing.

Given the complexity of B models obtained in an industrial setting, manual checks and review of traceability tables are insufficient to validate and detect any mistakes in the development process. As a result, this verification, as well as the verification of the realised refinement between system specification and component or software requirements, must be automated.

2.3.2 B and Industrial Projects

The development and use of the B method in an industrial context of railway systems was triggered by the application of formal methods in the development of the software of SACEM system deployed on RER A line in Paris for RATP [47]. Since then, several railway system manufacturers have generalised the use of B as their solutions for building critical systems in a correct by construction manner. In the context of RATP, the B method is used to develop the majority of the critical software.

Several success stories have been described in designing critical systems using a formal development approach with the B method, some detailed review are given in [64, 65, 66]. Further we present the most famous ones, such as the control system of the driverless Meteor line 14 in Paris or the VAL shuttle for the airport Roissy Charles de Gaulle [43, 44, 45].

Railway software development. *MÉTÉOR project*, the well-known Paris Métro system on Line 14, was the first to use the B method for the development of safety-critical software for railway systems. It is the first Parisian driverless Métro system and the main goal was to reduce the time interval between trains while ensuring the safety of the system. The software for this system was completely developed and formally verified using B for safety critical parts such as train running and stopping, train door control, and platform doors, which account for one-third of the overall program [65]. In addition to obtaining a correct by construction code for the system, simulation was used to validate the functional software requirements, this resulted with no errors during the integration testing phase and global validation test. The development method and validation process permitted the omission of unit testing [43, 67]. The application of formal methods combined with a validation process that took into account all the activities required to obtain the conviction that the critical functions have been safely implemented resulted in a reliable system with no errors detected since it was put into operation in 1998.

Several derived systems were developed and deployed globally based on METEOR system. The VAL system [44], which is the shuttle train at Roissy Airport, is MÉTÉOR's technical predecessor, and has been in operation since 2007. New VALs are now operational in Taipei, Rennes, and Turin as a result of the completion of this project. The Canarsie CBTC system [45], was deployed in New York on the Canarsie Line and, in comparison to Meteor, this system manages two different types of trains, equipped with CBTC systems and older ones that are not.

Hardware application. The generation of code for hardware is another application of the B method. RATP demanded the implementation of a system to control the platform screen doors that can be installed in metro stations in order to protect passengers. In this project, Clearsy used the B method for specification and programming of doors controllers [68]. This system is entirely independent of the train CBTC systems and it is used on the Paris Lines 1 and 13 as well as in São Paulo Metro.

Data Validation. The evaluation of the safety of a railway system is undeniably dependent on the correction of all configuration data and parameters used by the system. For each individual system deployment, the data parameters may be instantiated in different ways. This need to ensure that assumptions about configuration data hold led to the development of a new use case for the B method. As a result, formal methods can be used in data validation.

At RATP, this is a critical activity because the data parameters are typically instantiated differently for each system deployment and must be verified. To accomplish this, RATP has developed OVADO [69, 70], a generic tool for formal validation of system and software data safety constraints. This tool is based on a B predicate evaluation engine. On many metro lines in Paris and worldwide, data validation with B was used [71, 72, 63].

System Modelling. Following the success of the B method for Paris Line 14, another version of the B language, known as Event-B[23], was developed, allowing for the formal modelling and specification of not only the software but also of the system. Although the B approach allows the formal development of software based on software requirements and prove that it is correct, the resulting system will fail if the encoded requirements are incorrect. This results in the need for formal methods to be used during the design phase of system development.

Event-B has been used in the rail industry to validate system designs such as the New York Flushing Line [73], URBALIS 400 Zone Controller [74], RailGround interlocking, ETCS Hybrid Level 3 and Octys systems [75]. These projects, developed formal models of several CBTC systems in Event-B, and key system-level safety properties were specified and proved. Avoidance of train collisions, trains passing through unlocked switches (leading to derailments) and overspeed were among the safety properties addressed [64]. This was done either for the development of a CBTC, as in the case of the New York system, or for the

safety analysis of an existing CBTC, as in Octys sytem. The Octys formalisation for RATP made it possible to describe the properties that ensure the safety of the system and to fill the gap in the reasoning that provides complete arguments for safety.

2.4 HLL Language

Synchronous languages appeared as a solution for the modelling of reactive systems known to have high security needs. These languages are based on solid mathematical foundations, but in the same time they want to be simple from syntactic point of view. The languages must have the simplest formal model possible to make formal reasoning tractable [76]. They were introduced to respond to specific domain of application needs. Furthermore, they allow to apply a formal approach from specification to implementation or even execution of a program [76]. HLL [2] is a formal declarative and synchronous data flow language.

Reactive systems are systems that require a highly regulated real-time response to their environment. To model the behaviour of reactive systems, synchronous approach simplifies the programming of real time notion, because they are based on the notion of logical time. The logical time is defined as a sequence of instants of equal length, each of them corresponding to the execution of a system reaction [77]. The idea behind this logical time is to model the calculations that happen during an instant and that are completed before the beginning of the next instant, ignoring the exact time when the calculations occur. The most famous synchronous languages, successfully applied in several industries are:

- *Synchronous languages based on equations* such as LUSTRE [3] or SIGNAL [78]. LUSTRE and SIGNAL are two similar data-flow synchronous languages. A program is a set of equations defining system variables, structured in a hierarchical way, using *nodes* in LUSTRE or *processus* in SIGNAL. In both languages, a variable of a model denotes *an infinite sequence of values called flow*. Flows represent the communications between the components of the system. The notion of clock of a flow defines the moments when this flow has a value (is present) or not (is absent) and represents a mean to control the activation of different parts of the code. The main difference between them consists in the way the user can handle the clock notion. In LUSTRE the clock is integrated in the definition of flows. In SIGNAL, the clocks are explicit and can be manipulated independently of flows definition because clock variables can be declared. HLL language is close to LUSTRE.

- *Imperative synchronous languages* such as ESTEREL [79] that suitable for describing the control flow of a system. A ESTEREL program is structured in modules using a traditional imperative syntax and consists of a set of threads communicating through signals. A signal emitted by a process is instantly transmitted to all processes which monitor this signal. The execution of threads is synchronised to a global clock and represents the execution of the imperative code that it contains. It can be immediate or it resumes from where it stopped previously, being paused. The flow of control advances in each thread until a pause or termination instruction is reached.

HLL language [2] is designed for systems formal verification and it was developed by Prover Technology [13] in collaboration with RATP [14]. This language emerged for certification purposes for RATP, proposing the necessary features to enable formal verification of interlocking systems. It is a formal declarative and synchronous data flow language close to LUSTRE [3] with a SSA [15] form. The declarative nature of the language eases the definition of formal behavioural models as well as safety properties definition using temporal logic. The verification of these properties is performed automatically, using model-checking techniques, induction and SAT solvers, using dedicated tools.

HLL is suitable for specifying the attended properties of the system, it allows to make a separation between these properties, the functional description of the system and the constraints on the environment of the system. The reasoning in HLL is done in logical time without taking into consideration the real time taken by computations. This language was created to be the target language to an a posteriori formal verification approach for several programming languages such as C, SCADE or ADA. The main industrial constraints that HLL was designed to address were: simple and clear language that was expressive enough to allow for easy translation from other languages.

2.4.1 Overview of the HLL language

HLL language has a well defined syntax and semantics presented in its language specification document [2]. HLL models are defined in a propositional logic using as basic values infinite sequences or *streams*. The set of typed streams can be composed using either temporal or data operators.

[13] https://www.prover.com/
[14] https://www.ratp.fr/
[15] Single State Assignment

An HLL model describes the relation between the outputs and the inputs of a system and behaves cyclically: at each instant, the streams are clock-dependent traces of the model execution. A system defined in HLL can be described by a set of state variables (the memory variables from one cycle to another), the initial state of these variables, and the transition relation between states, from the inputs of the system to the outputs, using stream definitions.

In HLL, the substitution principle applies, meaning that any occurrence of a variable in the model can be replaced by the expression that defines that variable. Assignment statements are equations. The notion of sequence is not available (declarative) and therefore the order of stream statements does not affect the semantics of the HLL model.

```
Namespaces:
    n_id1 {
        Types: T
        Constants: c
        Inputs: I
        Declarations: v
        Definitions: S
        Constraints: C
        Proof Obligations: PO
        Outputs: O
    }
    n_id2 {
    ...
    }
```

Listing 2.2: The structure of a HLL development

Structure of a HLL model. A HLL model is structured in several sections (see Listing 2.2). The order of the sections does not affect the semantics of the model and each section may occur several times.

The sections of a HLL model are:

- **Namespaces** offers a hierarchical organisation of HLL models and may contain any of the HLL sections. This is a mechanism to avoid naming conflicts between introduced streams or types with another part of the model. The name of a namespace should be unique. A namespace may include other namespaces.

- **Types** defines specific types by associating a type name and a type expression. This can be a simple alias for basic types such as boolean, integer or arrays. However the type name can introduce new types when it designates for instance a sort, a structure or an enumeration type. This feature allows to define specific types for the system under study based on its data structure.
- **Constants** represents a list of declared constants streams.
- **Declarations** represents streams declaration with a name and a type information associated.
- **Definitions** introduces flow definitions with their values. More precisely, in this section values are assigned to the system flows and the state transition of the system is set.
- **Inputs** defines the input flows of the model. These flows (streams) are used in expressions but they don't have a defined value. An input stream represents any sequence of values in its declared type. A new value is affected to the input variable at each time frame. Inputs are an important component of variables in HLL because they can model the real inputs of the system or they can be seen as a way of abstracting the behaviour of physical components such as sensor for example.
- **Constraints** introduces the contracts of the system, assumptions on systems environment or allows to reduce the domain definition of unbound inputs streams. These allows to eliminate the situations in which the system cannot arrive. On the other hand, over constraining the system could be problematic, since by removing states from exploration we might hide violations of safety properties. Last, if the constraints are contradictory, all the properties are valid because the system is inconsistent.
- **Proof Obligations** represents a set of properties related to streams for requirements verification purpose. They express safety properties as a boolean stream and they are analysed by the model checker to determine their validity.
- **Outputs** defines the output streams of the system.

Streams. An HLL stream is represented as a sequence of values, of equal length, one for each discrete time frame (see Table 2.1). For instance, the variable x stands for the infinite

collection of values, one for each time frame, $x_0\ x_1\ x_2\ x_3\ ...\ x_n\ ...$, where x_n represents the value of the stream at time frame n. Similarly, for constants or numerals, 1 denotes the sequence of $1,\ 1,\ 1,\ ...$. The value of a stream respects the stream type at each discrete time frame. Streams have integer or boolean values and are interpreted in the mathematical sense, without side effects.

Stream	Values
x	$x_0\ x_1\ x_2\ x_3\ ...\ x_n\ ...$
y	$y_0\ y_1\ y_2\ y_3\ ...\ y_n\ ...$
$x + y$	$x_0 + y_0,\ x_1 + y_1,\ x_2 + y_2,\ x_3 + y_3,\ ...\ x_n + y_n$
c	$t,\ f,\ t,\ f,\ ...\ t$
$if\ c\ then\ x\ else\ x + y$	$x_0,\ x_1 + y_1,\ x_2,\ x_3 + y_3,\ ...\ x_n$

Table 2.1: HLL stream expressions

Streams are typed and can have the following scalar types: booleans, integers, sorts and enumerated. Based on these basic types, HLL proposes types such as arrays, structures or functions. HLL language proposes strict typing rules. The language is strongly typed and it relies on standard type inference techniques [80]. The inference rules of the language are given in [2]. Each flow has a single well-defined type and type errors may be produced at run-time. HLL language imposes a finite size for integer types because HLL models are based on bounded arithmetics.

Operators. HLL models are defined by a set of typed flows or streams that can be composed using either temporal or data operators. The operators used in HLL are:

- **Data operators**, like arithmetic, logical, array operators or lambda expressions and if-then-else, are used to manipulate streams values. These operators are point-wise applied characterised by the property: $\forall\ n\ \in N.\ f(x, y)_n = f(x_n, y_n)$. For instance, the sum of the two streams x and y has the following result: $x_0 + y_0,\ x_1 + y_1,\ x_2 + y_2,\ x_3 + y_3,\ ...\ x_n + y_n$. In HLL, similar to LUSTRE, the if-then-else is a conditional operator and should not be mistaken for the instruction with the same name of imperative languages. This operator requires that both then and else branches to be defined and the condition decides the value of the current flow being the one of the branches.

- **Temporal operators** that describe a relation from streams to streams, more precisely, clock-dependent expressions. The temporal operators, handling the time in HLL models, are *next* and *previous* operator (see Table 2.2). The *next* operator, written in HLL as `X(x)`, allows to access the value of expression x at the time frame following the current one. More precisely, it returns a stream shifted one time frame. The *previous* operator, `pre(x)`, allows to access the value of expression x at the time step preceding the current one. At the initial time frame, for a pre expression, the initial value must be given separately because it is not defined. For example, `pre(x, init)` where *init* is the value given for time step 0.

  The semantic of temporal operators is given as follows:

  - Previous operator without initialisation: for time frame 0 is $pre(x)_0 = nil$ and for next time frames is $\forall\ n\ \in N^*.\ pre(x)_n = x_{n-1}$
  - Previous operator with initial value: for time frame 0 is $pre(x, v)_0 = v$ and for next time frames is $\forall\ n\ \in N^*.\ pre(x, v)_n = x_{n-1}$
  - Next operator: for each time frame n is $\forall\ n\ \in N.\ X(x)_n = x_{n+1}$

Stream	Values
x	$x_0\ x_1\ x_2\ x_3\ \ldots\ x_n\ \ldots$
$X(x)$	$x_1\ x_2\ x_3\ x_4\ \ldots\ x_{n+1}\ \ldots$
$pre(x)$	$nil\ x_0\ x_1\ x_2\ \ldots\ x_{n-1}\ \ldots$
$pre(x, init)$	$init\ x_0\ x_1\ x_2\ \ldots\ x_{n-1}\ \ldots$

Table 2.2: HLL temporal operators

Besides temporal and data operators, HLL expressions can be defined using quantifications over finite domains. HLL provides the well known universal (∀, ALL) and existential (∃, SOME) quantifiers and as well other similar quantifiers on integers expressions such as computing the sum or the product, SUM or $PROD$, of an expression for all the values of streams defined in the domain. For example, $SUM\ x\ :\ [0, 2]\ (x + 1)$ is equal to $(0 + 1) + (1 + 1) + (2 + 1)$.

Types. The language features tuples, structures, arrays and functions. These constructions are streams composed of sub-streams and can be defined using a large set of operators. They represent mappings from values to values. Arrays of streams can be built and are considered

as streams that may have different values at each time frame. The access to arrays elements is done in a classical manner and if the arrays index is a defined stream, the resulting value is a stream dependent on the value of the index at each time step. The index of the arrays shall be in the arrays bounds.

HLL functions are stateless, combinatorial, because their outputs at current time-frame are dependent only on the current value of inputs without any time operators. They can be defined using only scalar parameters and they support a single output. This can be seen as an ordered mapping between the values from the domain of the function (the input parameters) and the value from the range of the function (the output parameter). We can define a function such that for each time frame $n \in N, x_n = y_n \implies f(x)_n = f(y)_n$ where x and y are two streams with the value at current time frame.

Stream Definition. A stream variable is defined by assigning it a value. If the stream variable is undeclared, it becomes implicitly declared by its definition and type inference is done.

A stream variable can be defined using the following operators: `I(x)`, to set the value of stream x at the first time step and `X(x)` to set the value of x stream for all other time steps. Furthermore, flows can also be defined in the form of a memory using `pre` operator (i.e. its value at the current cycle depends on its value at the previous cycle) or a definition using *next* operator (its definition depends on other flow variables values for its initial value definition and its next value definition). A flow variable can be defined with an expression of its type (HLL conditions: if then else or switch case, ...) or using a collection (function, array, ...).

HLL proposes several ways of stream equations writing:

- $x := e$ meaning that $\forall\ n\ in\ N.x_n = e_n$, x has the value of e at every time frame.
- $I(x) := e$ represents $x_0 = e_0$, initial definition
- $X(x) := e$ represents $\forall\ n\ in\ N.x_{n+1} = e_n$, next definition
- $x := e, f$ represents syntactic sugar and denotes $I(x) := e\ \ \&\ \ X(x) := f$, memory definition

HLL vs Lustre. Compared to LUSTRE there are several differences: in HLL only one global clock is available, while in LUSTRE the base clock of a programme can be divided and

streams are associated with a clock that defines the instants at which the current value of the stream is present. HLL is similar to LUSTRE but the writing style of properties is different due to the introduction of next operator. While in LUSTRE properties are expressed on the relation between the present and past (using the *pre* operator), in HLL properties are defined based on the evolution of flows in the future (using the *next* operator). This different way of writing the properties avoids the potential problems of non initialisation of flows, needed when using the *pre* operator. On the other hand, HLL is poor in modularity, we cannot define functions of flows similar to nodes in LUSTRE. In HLL, control structures such as clocks or hierarchical automata to describe sequential behaviours are not present.

2.4.2 Verification process

The correctness of properties is proved by using model checking. In general, in HLL models, the safety related properties are modelled as observers [81]. Observers either provide a higher assurance level of software correctness when proved valid or a valuable counterexample to analyse it and to correct the design.

Verification Tools. HLL models can be verified using a complete tool chain developed by RATP around the PERF methodology, based on a proof engine that combines model checking and induction proof methods. The tool chain is based on the HLL language, which is used to describe a formal model of the system to be validated and the safety properties to be verified. An overview of the tool chain is given in Section 2.2.4. Due to regulatory constraints, the tools used at RATP for systems verification must meet additional requirements in their development process to be accepted for use in a CENELEC SIL 4 process. RATP has qualified its tools with respect to their usage and to their contribution to the global safety of their systems.

Several commercial tool sets are proposed for industrial verification and are used at RATP. Prover Technology [16] has developed the PSL model checker[51] and the S3 [82] toolset for formal verification has been developed by Systerel [17].

Each tool uses a portfolio-based solver approach that supports both bounded model checking and k-induction to prove or falsify properties. For a falsified property, the counterexample is traced and generated, and debug tools are proposed for analysis. In addition to

[16] https://www.prover.com/
[17] https://www.systerel.fr/en/

property proof and equivalence checking, these tools support static run-time error detection. The development process of these tools makes them suitable to be used as verification tools according to the standard EN -50128:2011 [11].

These tools are from the family of SAT model checkers, symbolic model checking based on satisfiability [36]. They implement their own SAT solver, but can also be used with other external SAT solvers. Some of the solvers implemented in these tools are:

- Bounded Model Ckecking (BMC) engine [83, 35], it relies on a strategy to search whether a property is satisfied or not on fixed length traces. The exploration domain is fixed by a constant. It performs a standard iterative unrolling of the transition relation and outputs either a counterexample or a guarantee that there is no violating trace up to the given trace length. If a counterexample is found, the BMC engine guarantees to find it at minimum length.

- k-Induction engine performs the proof of properties over an infinite trace by performing the inductive step of k-Induction. k-Induction is a well-known technique for the verification of transition systems [84, 85]. For a proof of k-induction, the engine checks in two parts: base case and inductive case.

In the process of formal verification, a HLL model is flattened to a LLL model, a highly restricted boolean subset of HLL. The obtained LLL model is passed to the model checker tool for analysis. The model checker performs static verifications, such as checking the sanity of the model to determine if undefined behaviours have been introduced due to undefined array accesses, circular definitions, or the use of pre operator with undefined initial values. Proof of properties is then performed according to the chosen proof strategy: bounded model checking or induction.

Analysis. It must be shown that the properties are preserved at each cycle This can be done by induction to validate the proof. If the model checker cannot prove that the properties are inductive, it breaks down and generates a counterexample. There are two main reasons for this: either the properties are not sufficient to be inductive resulting in an indeterminate result, or the properties are not respected by the system. To separate these two cases more easily, the proof phase can be preceded by a bounded model checking to debug the model, which consists of directly showing that the property is true for a certain

number of cycles. This proof is no longer based on induction and thus eliminates the second source of counterexamples.

The result of the proof may be valid, falsified, or indeterminate. A property is said to be valid if the analysis of that property leads to the conclusion that it is true in any reachable state. If, on the other hand, there exists a reachable state in which the property evaluates to false, then the property is falsified.

The verification engineer should use the analysis of the generated counterexample to either fix the system architecture or refine the propositions for the specific case of a false positive result. The indeterminate result may arise when the model checker cannot validate or falsify a property. In particular, this behaviour occurs when the property to be proved is either non-inductive or falsifiable, and the model checker requires longer traces than those analysed to achieve falsification and show the counterexample. A non-inductive property requires additional lemmas to be proved.

In the proof process, constraints on the system are assumptions for the proof because they reduce the state space. Only the states in which the constraints hold are considered, leading to a proof under the assumption that the constraints are always true.

2.4.3 HLL industrial projects

Like the B language, the HLL language and tools emerged from industry needs. The HLL language has already proven its usefulness in industry projects, presented later.

The HLL language and the formal verification methodology built around it, PERF, were first used for safety demonstration of PMI systems (computer based interlocking). This activity allowed to conclude that the formal verification could replace the classical safety testing review activity and further RATP applied it for safety assessment of different systems. This was the beginning of independent safety assessment at RATP using HLL and PERF method.

RATP has been using HLL language and the PERF workshop in its verification process for more than 10 years. It has been successfully used for verification of systems such as CBI (Computer Based Interlocking), wayside and on-board equipment of CBTC (Communication Based Train Control) [46]. The industrial use cases around HLL and PERF applied at RATP are safety proof, system debugging and equivalence checking. Checking the soundness of a specification, proving SCADE programs at a low level, proving handwritten source code, proving relay schemes, and demonstrating the equivalence between ADA and SCADE

programs or between C and SCADE programs are among the most popular activities already completed at RATP.

Interlocking Systems Verification. Safety assessment of the interlocking software was realised by proving the safety properties, the absence of safety hazards generated by design anomalies in the software, such as non-collision and absence of derailments. The proof considered a HLL modelling of the environment for train behaviour and allowed the validation of CBI systems with about 70 routes and more [1]. Table 2.3 provides an overview of the major CBI projects whose software was validated with HLL language at RATP. Different proof workshop were used for the verification activity and they support different input development methods such as Petri net [86] or relay based schema [87].

Line	System	Dev. Method	Toolkit	Year	Usage
8	PMI	PETRI Nets	Prover Certifer	2011	Safety Demonstration
12	PMI	PETRI Nets	Prover Certifer	2011-2012	Safety Demonstration
4	PMI	PETRI Nets	Prover Certifer	2013	Safety Demonstration
1	PMI	PETRI Nets	Prover Certifer	2013	Safety Demonstration
6	PHPI	Relay schema	PERF Toolkit	2018	Safety Demonstration
8	PHPI	Relay schema	PERF Toolkit	2021	Safety Demonstration

Table 2.3: Interlocking systems verification with HLL

Relay-based interlocking Systems Verification. Further, this formal process was used for safety demonstration of a new generation of relay-based interlocking systems at RATP, the PHPI system for Paris Metro lines 6 and 8. For this project, a tool was developed by RATP to translate relay-based schemes into HLL models, more details can be found in [87]. In addition to using this approach for internal purposes, RATP has also applied it to external missions such as the independent safety assessment for the New York City metro system or the expertise for the RFF project [49].

CBTC Systems Verification. Table 2.4 shows the CBTC projects that have been validated by RATP using PERF, with one of the largest systems to be validated having 6 million lines of code. The proof process allowed to show several counterexamples which were corrected in the design before commissioning.

Line	System	Dev. Method	Toolkit	Year	Usage
3	CBTC wayside equipment	SCADE 5	Prover Certifer	2010	Safety Demonstration
5,9	CBTC on-board equipment	SCADE 5	PERF Toolkit	2013	Safety assessment
13	CBTC equipments	SCADE 6	PERF Toolkit	2014	Safety assessment

Table 2.4: CBTC systems verification with HLL

Proof at system level. Because of the growing complexity of the railway systems that RATP must verify, engineers were forced to upgrade their verification methods, and a new use case for HLL, formal proof at the system level, emerged. This process entails a system validation to formally derive high-level safety requirements that must be met by subsystems and proved on software at the HLL level using model checking and abstraction techniques [50].

The application of PERF on safety validation of these systems has clearly shown the value of this approach in terms of improved quality of safety assessment.

Use of HLL. In [41], several use cases for HLL and model checking realised by Systerel are presented. First, the triplex sensor voter of aircraft systems is presented, which relies on three equivalent sensors to compute an output value. The aim of this case study is to investigate how floating-point arithmetic can be handled in HLL and the S3 model checker. They propose a solution that involves building a library for floating-point arithmetic and integrating it into the S3 tool. They also describe in detail a use case from the railway domain for safety checking of CBIs. Similarly, in [88], the authors reported an a posteriori approach to apply formal methods to already developed software by translating SCADE code into HLL models. This work is similar to the RATP verification method described above. HLL and model checking have been used by Prover Technology for a variety of use cases, including specification verification, interlocking systems safety verification and route visualisation [18]. In [89], they provide an overview of the implementation of their solution on various projects. A formal verification approach using HLL was applied at Alstom on their

[18]For more information on projects where Prover's solutions have been used, see: https://www.prover.com/references/

interlocking systems using tools developed by Systerel and RATP [90]. Another example of the use of HLL is the search for a set of tests, inputs, and oracles that can be used to satisfy a software's structural coverage criterion.

HLL, a formal verification standard. The ease of use of HLL, combined with formal verification tools capable of managing large industrial systems and producing certifiable results, may account for its popularity among railway manufacturers and operators. A community is emerging around this language with the aim of developing a formal verification standard from it.

2.5 State based semantics

In our approach, we are interested in the B method. The B method is a state-based method with explicit modelling of states by variables and manipulation of states by operations. It follows a state-based semantics formalised as a state transition system. Transition systems are useful for defining systems and their behaviour as well as for reasoning about their execution traces. A variety of properties can be checked on such systems.

Transition System. A LTS (Labelled Transition System) is a 4-uplet $\langle S, S_0, E, R\rangle$, where S a set of states, $S_0 \subseteq S$ a set of initial states, E a set of actions or labels and $R \subseteq S \times L \times S$ is the transition relation. A transition $(s, e, s') \in R$, written $s \xrightarrow{e} s'$ denotes a transition from the state s to s' after the occurrence of the action e. We assume that E includes a special action τ, $\tau \in E$ to represent an internal action. We introduce the following notation $s(\xrightarrow{\tau})^* s'$ to denote the trace $s = s_0 \xrightarrow{\tau} s_1 \xrightarrow{\tau} ... \xrightarrow{\tau} s_n = s'$. It represents the trace of zero or more internal actions from state s to s'. We write $s \stackrel{e}{\Longrightarrow} s'$ to denote the trace $s(\xrightarrow{\tau})^* s_1 \xrightarrow{e} s_2(\xrightarrow{\tau})^* s'$.

The semantics of the B method can be represented by a state-based semantics modelled by LTS. For this purpose, states can be defined as a set of pairs that map each variable to its value. A function exists to retrieve the value of any variable state, $l : VarName :\rightarrow Values$. Substitutions associated to B operations represent transitions.

Strong and Weak Bi-simulation

It is possible to compare a LTS to another LTS. Relationships based on behavioural equivalences, such as simulation or bi-simulation [18], allow LTS with the same set of labels to be compared. Milner [18] was the first to define the concept of bi-simulation for process comparison, called observational equivalence. There are two types of observational equivalences based on the type of considered action (observable or internal), strong bi-simulation that only considers observable actions and weak bi-simulation [18, 17] that considers both observable and internal actions. We discuss these relationships in more detail further.

Simulation. Let $S = \langle S, S_0, E, R\rangle$ and $S' = \langle S', S'_0, E, R'\rangle$ be two labelled transition systems. A simulation is a binary relation $\preceq$ between states, $\preceq \subseteq S \times S'$, such that:

$$\forall s1, s2 \in S, a \in E, s1' \in S'.$$
$$s1 \preceq s1' \textit{ implies that } \textit{if } s1 \xrightarrow{a} s2 \in R \textit{ then } \exists s2'.\ s1' \xrightarrow{a} s2' \in R' \textit{ and } s2 \preceq s2'$$

A state $s1'$ simulates another state $s1$ if $s1'$ may match step by step the behaviour of $s1$. Simulation is not a symmetric relation.

Bi-simulation. Let $S = \langle S, S_0, E, R\rangle$ and $S' = \langle S', S'_0, E, R'\rangle$ be two labelled transition systems. We say that a relation between states in two LTS, noted $\simeq \subseteq S \times S'$, is a bisimulation if

$$s1 \simeq s1' \textit{ implies that } \textit{if } s1 \xrightarrow{a} s2 \in R \textit{ then } \exists s2'.\ s1' \xrightarrow{a} s2' \in R' \textit{ and } s2 \simeq s2'$$
$$\textit{if } s1' \xrightarrow{a} s2' \in R' \textit{ then } \exists s2.\ s1 \xrightarrow{a} s2 \in R \textit{ and } s2 \simeq s2'$$

Bi-simulation is a symmetric relation. $s1$ is bi-similar to $s1'$ implies that $s1'$ can do everything that $s1$ can do, and vice versa, at every step of the computation.

Weak Bi-simulation. Let $S = \langle S, S_0, E, R\rangle$ and $S' = \langle S', S'_0, E, R'\rangle$ be two labelled transition systems. We say that a relation $\cong$ between states in two LTS, noted $\cong \subseteq S \times S'$,

is a weak bi-simulation if

$$s1 \cong s1' \; implies \; that \; if \; s1 \xrightarrow{a} s2 \in R \; then \; \exists s2'.\; s1' \stackrel{a}{\Longrightarrow} s2' \in R' \; and \; s2 \cong s2'$$
$$if \; s1' \xrightarrow{a} s2' \in R' \; then \; \exists s2.\; s1 \stackrel{a}{\Longrightarrow} s2 \in R \; and \; s2 \cong s2'$$

$s1$ is weakly bi-similar to $s1'$ means that at every step of the computation, if we ignore the internal actions, $s1'$ can do all that $s1$ can do, and vice versa.

We could, for example, demonstrate that a source LTS is transformed into a bi-similar target LTS. The semantic relationship between these two models is bi-simulation, which means that both systems can simulate each other from an observational point of view. To prove such a relation, the labelled transition systems corresponding to the semantics of the source and target models must be encoded.

2.6 Previous work on semantic formalisation

Embedding the abstract syntax, semantics, or type system of a language in a formal setting, can be used to reason about models written in that language as well as about the language itself.

State based languages. The semantics of a state-based language is usually defined by a set of transition rules that record changes of the state of the program. The B language has been the subject of several formalisations using theorem provers with different goals, addressing some aspects of the language or the full B specification. A semantic embedding of B abstract substitutions and machines in Isabelle/HOL is given in [91] of B with the purpose of producing a formally verified proof obligation generator. For example, the proof obligations for the composition relation INCLUDES have been formalised and checked. Note that a shallow embedding of B models is used in this work. An axiomatic semantics based on predicates transformations is defined. Similarly, in [92] a formalisation approach of the B semantics in terms of state transition systems of B substitutions and refinement theory using COQ and PVS is carried out. A deep embedding of B language is given in [93] using Coq theorem prover to study the theory of B.

In [94], a formalisation of B semantics in Isabelle/HOL is presented as a labelled transition system. The focus is on the component composition relation in B, the formalisation of notions such as specification, refinement and composing components. Based on internal and

observable states, a behavioural equivalence - simulation relation, is demonstrated between the LTS of B abstract machines and the LTS of refinement machines. This work serves as a foundation for a formal verification of the automatic refinement rules implemented in the BART tool [61]. B generalised substitutions formalisation implemented in Isabelle/HOL is also realised in [95].

Single State Assignment. In addition to state-based language semantics, synchronous language semantics using the single state assignment (SSA) form has been studied in detail in the context of modern compilers. HLL is a synchronous language with SSA form. In the SSA form, each variable in a program occurs only once on the left-hand side of an assignment. A program is usually converted to SSA form by replacing the assignments of a program variable with assignments to an intermediate version of that variable corresponding to each assignment.

Many conversion algorithms from an imperative language to SSA form rely on graph theory for verification. Indeed, control flow graphs (CFG) [96] have been used to represent the semantics of SSA form. CFG of a program is a way to represent the control structure of imperative programs, where edges stand for possible transitions between nodes or blocks containing the instructions to be executed. When a variable can be assigned in both branches of a program block, such as in if statements or in the body of a loop, the ϕ operator is used to select the new variable value based on the program control flow. The ϕ nodes represent the introduction of new assignments to merge information from different control flow branches to restore the flow of values from renamed variables.

SSA was first used in imperative compilers as a data structure to reduce the complexity of data flow analysis algorithms. It can also be used as a target for functional languages [97]. The SSA form can be seen as a first step towards abstracting the semantics of imperative languages into a language of declarations; it is a way of unifying multiple program paradigms.

The semantics of an SSA program is specified as a function that maps identifiers to the semantics of their values in [98]. Usually, the semantics of the SSA representation is presented together with the conversion algorithm. These algorithms focus on optimisation techniques and practical aspects to facilitate the implementation of a compiler [99, 100]. The control flow graph of B substitutions from a B *inplementation* is presented in [101].

2.7 Formal Verification of Model Transformations

Verification tools or compilers frequently deal with language transformations, and their correctness is critical. The correctness of the tools used to verify safety-critical systems is just as crucial as the correctness of the system itself. These tools typically use models or program transformations. Verifying the correctness of complex programs, such as translators, is challenging. When used in a verification process, the translator is also a program that can introduce errors during compilation by generating incorrect target code from a correct source program. As a result, system verification could be performed on an incorrect model. Checking a translator involves ensuring that the translator does not generate incorrect target code from a correct source program. Formal verification and certification of translators in [102] for first order logic compilers. A biography on compilation of higher-order functional languages is given in [103].

A transformation means that, given a source model, an automatic process produces a target model that conforms to a transformation description [104]. As a result, the primary requirement for a transformation to be considered to behave correctly is to produce a target model conform to the source model from a semantic point of view. To ensure this correctness many approaches exist, one of them is to relate the meaning between source and target models. This implies to realise an embedding of language properties to define a program's semantics. The explicit representation of a model's semantics can be realised for example using Labelled Transition Systems (LTS) [105], event structures [23] or Petri nets [86].

Several notions are necessary for proving the correctness of a translator/compiler such as:

- Formal definition of source and target language semantics and properties.
- Formal specification of the translator, it could be a definition of the algorithm or the implementation itself. The translator is specified by the transformation relation between source and target language elements.
- Making manual or computer-assisted proofs in the proving environment. The verification can be performed using a proof assistant, such as Isabelle/HOL[5], Coq[24] or automated theorem provers.

Many compiler verification strategies have emerged to ease the difficulty of verification processes. To review these techniques, we follow the classification proposed in [106].

Certified compilers. A certified compiler is one that has been proven correct by computer assisted proof. This implies that the compiler conforms to its specifications and generates models that act in accordance with the source model. A certified compiler, in particular, requires two verification processes. The first step is to demonstrate that the translation specification is correct for all provided source models, i.e. the behaviour of target model matches with exact behaviour of the source model. The second stage is to demonstrate that the translator's algorithms are correctly implemented in accordance with their specifications. The specification of the translator and its implementation are formally specified. Certified compilers are acquired using proof assistants to express the compilers correctness theorem and prove it using the logic of the assistant.

Many contributions investigated compiler certification for various language paradigms using different provers. The CompCert compiler [107] is a formally certified compiler that generates assembly code from the C language using the Coq proof assistant [24]. The goal of CompCert is to produce optimised running programs that are free of miscompilation issues while preserving semantics. The generated target code is extracted directly from the theorem prover. The compiler has been decomposed into several elementary transformations whose correctness has been proven in Coq to facilitate verification. It translates in several sequential steps through a number of intermediate languages, as most optimising compilers do. Each intermediate language is associated with a semantics, and each compilation step is followed by a theorem stating that it introduces no new behaviours. A similar method is provided in [108] for compiling Java-like programming languages with Isabelle/ HOL.

Other approaches, such as the Verifix [109] and Vellvm[110] projects, aim to build mathematically correct compilers. They develop techniques for mechanised formal semantics of languages and SSA form, compiler decomposition, and semantic equivalence of source and target programs. An approach for verified compositional compilers for multi-language software is presented in [111]. The formal verification of the compiler is provided using LUSTRE in [112, 113].

Certifying compilers. Certifying compilers are about establishing and proving the link between two programs - source and target - rather than focusing on the correctness of the algorithm or the implementation of the translation. Proof-carrying code and translation validation are some of the techniques that have been proposed to automatically generate proofs for each run of the compiler.

Proof carrying code [114] is a method that allows to guarantee that requirements such as type safety or absence of stack overflows are satisfied by a target program by generating a proof, called a certificate, alongside the code program that this code is correct with respect to its specification. When the code is used, the certificate is examined separately by an external verifier.

Translation validation method was first introduced in [115] to detect errors in compilers to avoid generating incorrect code when compiling synchronous languages. Translation validation requires proving that a particular execution of a compiler exhibits the desired behaviours, rather than showing that any compiler execution is correct. This approach is an a posteriori approach applied to target models already obtained. After compilation, the source and target models are passed to an independent tool called *validator*, which verifies the defined correctness properties at syntactic or semantic level, so that the result of execution is the same for the model and the code semantics. Depending on the result returned by the validator, the compilation is continued or aborted. However, the validation module must itself be certified. This approach has been used in the verification of various systems [116, 117] and various validator techniques can be used, such as proving algorithms using proof assistants, abstract interpretation, or model verification. In [118] formal verification of the validator is realised by symbolic execution to show semantic equivalence, with proof formalised in Coq proof assistant.

Some of the advantages of translation validation include the fact that, unlike certified compilers, it is an approach independent of the code generator, making it less susceptible to modification when the code changes. Furthermore, it is easier to verify that the output has the expected properties than verifying the entire code generator. What appears to be a benefit can also be viewed as a disadvantage because this validation must be performed for each translation.

A transformation validation technique capturing the clock semantics in the models is shown in [119]. There, a refinement relation between the source specification and the generated code is proved using SMT solvers. Program refinement is defined on inputs and outputs, stating that the behaviours of the target code include all the behaviours of the source code. A similar approach proving that binary code refines its C source using an SMT-based proof process is presented in [120]. Synchronous versus sequential code validation based on the proof strategy is presented in [121, 122]. A refinement relation between synchronous transition systems is shown and it is applied to the non-optimizing

compilation of SIGNAL to C.

Many contributions studied the correctness of compilers using formal validation and certification of translators or translation validation approach for various language paradigms using different provers. Both methods have their own advantages and disadvantages. Depending on the verification objective and the resources available, one or the other method may be chosen. We believe that combining the two methods can provide great results, such as providing a formal proof of the correctness of algorithms implemented in a compiler for all inputs and the use of translation validation to ensure that algorithms have been implemented correctly at tool level.

The compiler is generally regarded as a black box and the semantic equivalence is defined through the analysis of proofs based on semantic relationships between source and target programs. [123] uses Isabelle/HOL to systematically confirm the transformation of Java program to Java byte code. In [124], the authors present a Isabelle/HOL formalisation of a subset of Java and its corresponding abstract machine to verify type related safety properties on the programming language. In [125], an automated generation of correct translation of the program is defined. Source and target code semantic equivalence is demonstrated using a simulation-based proof.

The previous approaches rely on the formalisation of a relation guaranteeing semantic preservation. Refinement, simulation, bi-simulation certified compilation, correct by construction transformation, etc. are the relationships reviewed above. In our work, we adopted a methodology close to the above approaches which consists in checking semantic preservation and/or semantic equivalence using bi-simulation relationship defining an observational equivalence. This relation compares states of the two models at each execution steps.

2.7.1 Translators for SSA models

In order to prepare the state based translation to dataflow, the particular case of translation of SSA based semantics is considered. Below we review the techniques. Several works use translation validation technique combined with mechanised proofs to prove a compiler correctness when an SSA form is used. In [126], authors have presented a formal approach for translating source of imperative programming languages, such as C and C++, into synchronous language Signal [127] using an intermediate SSA language. Model-checking is used in this work to check the required properties. They attempt to show that the transformation in SSA form does not produce state explosion. Even if usually the transformation

from imperative to SSA form passes from multiple assignments for a variable to single assignment of multiple temporary variables, this intermediate variables do not impact the state space. CompCertSSA [128] is a translation validator for SSA form construction algorithm that converts imperative variables to SSA variables. CertSSA translations maintain a close relationship between source and target variable names, which simplifies simulation relationship checking. Pop et al. [129] present non-standard denotational specification of the SSA form, including translation from imperative languages to SSA, and vice versa. A similar approach to SSA formalisation is provided in [130]. A transformation based on SSA register assignment from a functional to an imperative language is shown in [131]. The transformation is validated via a bi-simulation relation proven in COQ.

2.7.2 B Translators

The use of the B method in the context of French railway industry is recognised and accepted as a strong safety proof. In safety-critical systems modelling, several techniques have been proposed to bridge the gap between natural language system specification and its implementation. In this setting, there are many modelling examples that are combined with B method for code generation.

Iliasov [132] describe the transformation from a Domain Specific Language for railways into Event-B formal modelling language. Their goal is to verify the safety of the railway systems using a multi-level modelling approach while ensuring an efficient exploitation of a railway network. [133] describes how railway systems can be designed, starting from requirements formalisation to system specification and safety requirements verification as hierarchical Coloured Petri nets, to final implementation by presenting the transformation from Coloured Petri net model to B language. The interaction between CSP-like system behaviour descriptions and abstract B machines is described in [134, 135] for reactive systems development. They study how these designs can be converted to B for analysis and refinement to code. Several works have been achieved on requirements formalisation combining graphical notations with formal one such as: the translation of UML diagrams into B specification [136, 137]. In [138], B System specification is derived from the SysML/KAOS domain modelling language to fill the gap between system textual description and the formal specification for better readability of models and traceability.

Often, after the modelling and verification of B models, their transformation into executable code using different code generators is intended. This transformation is realised from

a concrete representation of B models, the implementation level that must be deterministic. From this level the translation into an imperative language is straightforward but many features of B language are missing. One such code generator is presented in [53], it allows generation of provably correct Ada code. [139] shows an optimised transformation from B specifications to executable C code to meet hardware constraints. The general architecture of the transformation process is presented together with optimisation techniques. However, they do not address the formal certification of the transformation. Other code generators are integrated in AtelierB [48] such us C4B that generates C code from implementation level subset of B. Moreover AtelierB also integrates Ada code generator.

In [52], B implementations are used to generate Java code that can be embedded in smart cards. The architecture of the experimental open source platform, jBTools, supporting the code generation tool from B to Java is presented in [140]. Another tool for B to Java code generation, BSmart [141], has been developed in the context of smart cards applications and provides automated support to handle communication and codification aspects particular to Java Card platform.

Bonichon et al. [142] have developed a tool, *b*2llvm, for generating LLVM executable code from B models. Singh et al. [143] proposed EB2ALL, a notable tool supported code generator, that generates source code in many programming languages such us Java, C, C++ and C# from verified Event-B specifications. Furst et al. [144] proposed a code generator to produce C code from Event-B models. In similar vein, [145] described a method for generating VHDL source code from B specification.

In [146], authors proposed a set of translation rules for generating HLL models from Event-B models. In fact, the main objective of this work is to use an intermediate HLL representation to produce C code from the Event-B specification. They use equivalence proof to check the correctness of the code generation process. To our knowledge, the proposed translation approach from Event-B to HLL is not automated yet.

Another approach is to translate B models from higher abstraction levels. For example, a set of translation rules is presented in [137] to generate Java/SQL code from B models for designing and analysing database systems. A tool B2Jml [147] is developed to generate JML (Java Modelling Language) specifications from B specifications for combining the use of formal methods with software development techniques. A similar approach is described in [148] for Event-B to Java code generation along with the generation of additional JML contracts for code verification.

The B2Program tool presented in [149] targets code generation from all abstraction levels of B. A template based approach that features various output languages, including Java, is shown as well as extension possibilities for other languages.

The previous approaches rely on generating code, namely from B models to various programming languages. To our knowledge, there is no work that addresses the certified transformation from B language to HLL dataflow language. Moreover, to the best of our knowledge, there is no work that addresses the verification and certification for heterogeneous systems that meets our stated objectives. Our work aims to provide an integrated verification framework for modelling and verifying heterogeneous systems under given requirements formalised as logic properties in a non-intrusive manner.

2.8 Conclusion

In this chapter, we have provided the scientific background for this thesis, which deals with the verification of complex systems. First, we defined safety critical systems and provided a general overview of formal methods that can aid in the verification of these systems, as our work consists in proposing an independent formal verification of safety critical systems for railways. Then, we presented two formal languages, B and HLL, as well as their verification processes and applications in the railway domain.

In the implementation of railway applications, the achievement of the highest possible guarantees is often a key factor. Our verification technique is based on the language transformation from a state-based to a dataflow language. We presented the concepts required to certify this transformation, taking into account the application domain of this research and its strong regulatory constraints. Finally, a literature review was conducted. Such a transformation will serve as a link between critical tasks, bridging the gap between requirements review and their implementation on real systems.

Part II

Contributions

Chapter 3

B-PERFect

Contents

RATP launched the B-PERFect project to study the applicability of PERF on software systems developed using the B method. The goal of the B-PERFect project is not to replace B's formal verification process, but rather to provide a verification alternative for internal independent safety assessment. This chapter describes the motivation and the main goals of the B-PERFect project. We then describe our approach to software safety verification and give an overview of the overall certification process of the transformation. The subset of the B language that will serve as our starting point for this research is the introduced. Finally, we present a toy case study that exemplifies our approach.

3.1 Introduction

In the railway domain, safety assurance is difficult to achieve due to the existing gap between the high-level system requirements expressed in natural language and the low-level software implementation. Indeed, requirements are usually expressed at a higher level, while software implementations manipulate concrete concepts not even mentioned in these requirements,

resulting a gap between abstract-level requirements and concrete implementations.

Furthermore, gluing system-level safety requirements to the software components responsible for ensuring those safety requirements is a difficult task. In general, this glue is made explicit through a series of models that refine abstract-level specifications to produce concrete software components. The verification and validation of system safety requirements can be challenging in particular when the complexity of such systems results from features such as: high reliability and safety, multiple technologies used, large systems, and complexity of functionalities.

Several approaches have been proposed to bridge the gap between natural language system specification and its implementation when modelling complex safety critical systems. Such approaches rely on a top-down verification of the system using various techniques in the field of requirements engineering and formal methods. These approaches are applicable in the design and development phases of a project [150]. In this setting, there are many modelling examples that are combined with B method for code generation.

These papers [133, 132, 134, 135, 136, 137, 138] present various approaches that attempt to bridge the level at which one can realise verification between the specification and design of a system and code generation. Although it is really important to address system verification in the early stages of system design, if the system is not yet implemented, there is little guarantee that the system implementation will actually satisfy the properties specified in the design phase.

The main motivation for our work is to provide an independent alternative to establish that system implementations preserve the safety properties that have been defined at the design level.

In the railway domain, before putting into service a safety critical system, detailed verifications and validations are necessary. In order to maintain a high-level safety assurance of its systems, RATP designed a formal verification methodology named PERF [1] to assess the safety level of railway systems and to deal with the heterogeneity features of its configurations. The aim of this approach is to realise an independent assessment that helps to double check the safety of the developed software in addition to the verification performed by the software supplier. PERF was designed to be applicable to any software system independently of their development process and languages. By taking the source code of the developed software as the target of the verification, it ensures a complete independence and non-interference with the software supplier which drastically reduces any possible bias.

It also allows for applying formal verification techniques to the safety assessment activity, which is not always achieved by the software supplier in its safety verification.

Our research contributes to broadening the range of systems verified using the method PERF. It focuses on the integration of systems developed with the B-method [4] in PERF. In this research work, we answer the question of how safety evaluation can be realised in the context of heterogeneous systems.

This chapter presents the proposed verification approach for systems modelled with the B-method and how they are integrated into the PERF toolkit.

3.2 B-PERFect Goals

Our concern is to validate and verify systems developed by different stakeholders using their own modelling languages and development processes, based on the requirements of the main contractor (here RATP). RATP collects a set of heterogeneous models seen as black boxes that are validated by each stakeholder. A rigorous standard black-box verification and validation process is established for heterogeneous models collected from different stakeholders. In this process, stakeholders receive a set of requirements and produce software components that satisfy those requirements, modelled and verified using their own, sometimes proprietary, verification techniques. In parallel with the verification activities performed by the stakeholders, the system integrator performs independent safety assessment activities using the PERF integrated verification framework. The B method [4] is one of the modelling and verification techniques that stakeholders consider.

The B method supports the development of software components by refinement of high level specifications. At each refinement step, safety properties are expressed using invariants, and more and more concrete design decisions are introduced. The last refinement step reaches the level of software code. Each refinement step is associated with a set of proof obligations to be proved in order to ensure invariants preservation. It has been applied in several projects and has proven to be an effective approach for developing a complex system using refinements to move from high level system requirements to low level implementations [68]. However, independent assessment of safety-critical systems developed using the B method in terms of informal requirements may be time consuming, and in some cases, intrusive. Indeed, the abstraction choices made to model the system in B could lead to error masking between different systems requirements and desirable properties.

Detecting inconsistencies in invariants automatically is a difficult task. In fact, the B models are verified using the associated proof system. Proving invariants may thus fail because the proof is incomplete or the invariants are inconsistent. In general, model animation or model checkers may be used to find potential counter examples in the B models, making this approach intrusive, i.e. it needs to dive into and manipulate B models.

Although the formal verification performed by the B proof engines can be trusted, the validation of the safety properties can only be accomplished through tedious and inefficient code or specification review activities. Another method for assessing the consistency of the B models is to introduce additional lemmas or insert contradictory invariants, and thus by modifying the original models. Note that in our work, this approach is not applicable because such changes to the given model may jeopardise the safety assessment process.

Currently, some suppliers at RATP use the B method and thus formal proof as part of the design of their critical software. The practical implementation of this method in industry projects rarely proves that all software requirements are met. Even though the B method allows to bypass the unit and integration tests, it does not replace validation tests. However, the evaluation of these tests can be particularly costly for safety assessment teams. Inspections are carried out to ensure that the B models of a system are consistent with the requirements [44]. It is well known that textual descriptions of requirements can be ambiguous and open to different interpretations, especially when dealing with complex systems. During inspections, traceability is established between B-models and their specification, and manual checks are performed to verify the implementation choices and to determine whether the system model implements its specification. This activity has shown its limitations, especially in the current context where systems are evolving and becoming more complex. The evaluation of validation tests combined with B-model inspections can be considered relatively inefficient compared to the benefits offered by the use of formal proofs, thanks to the formal modelling of safety requirements and the completeness of specification analysis through proofs.

Goals. In order to propose an independent and automatic safety assessment approach of safety critical railway systems developed with B method, the B-PERFect project was initiated. We establish the following goals:

- Ensure that the requirements of the system specification are well met by using a method or a validation technique that is **independent of the manufacturer**.

- From the perspective of system integrator, consider validating **the requirements on the overall system** integrating the different subsystems delivered by stakeholders and possibly designed with different modelling techniques.

- Propose *a non intrusive* verification methodology that does not require any changes to system's code.

- Increase the applicability of PERF methodology.

- Offer the capability to prove high level system properties on concrete software using an integrated verification environment.

- Using compositional verification, ensure that **subcomponents of a system respect global safety invariants** by decomposing the verification of a software system into different subparts.

The B-PERFect project supports the verification of systems developed using the B-method with respect to the safety properties expressed in HLL and addresses the above goals. The B models are transformed into HLL models according to the PERF approach, where the required safety requirements are considered to check the correctness of the system behaviour. The use of such an approach allows to introduce and prove additional system properties and to verify properties that are at a different level of abstraction. Furthermore, B models can be integrated with other models developed in a different language, allowing for integrated verification of safety properties in a single modelling language. This could be especially useful in heterogeneous systems.

The idea behind this is not to prove again the already proven properties on the collected B models, but to guarantee the correctness of new safety properties which could not be expressed on the isolated B model due to lack of its environment, i.e. when information on the environment where the programs run is missing. Indeed, configuration data representing the environment in which the system evolves can be formalised in HLL outside the B models. As a result, the translated models are enriched with constraints or assumptions describing this environment. This allows to constrain the verification to realistic conditions and a better understanding of the environment model. Furthermore, this approach enables to check the conformity of the implementation choices and the encoding of the safety properties in B models.

This process questions the B models without manipulating nor instrumenting them. The resulting approach is qualified as non-intrusive and facilitates formal verification and validation of the integrated system components by external system designers.

3.3 Our framework

As mentioned previously, our aim is to deploy the PERF approach for B models. In the context of the integrated development of safety-critical software, *a posteriori* and *non-intrusive* verification by expressing high-level properties in an independent language such as HLL is achieved. To support this task for B software, we use B2HLL tool, which is a prototyping approach that transforms B into HLL. A crucial part in our approach is to show that the translation in HLL preserves the semantics of B. The following processus is set up for this purpose. Our approach is depicted in Fig. 3.1.

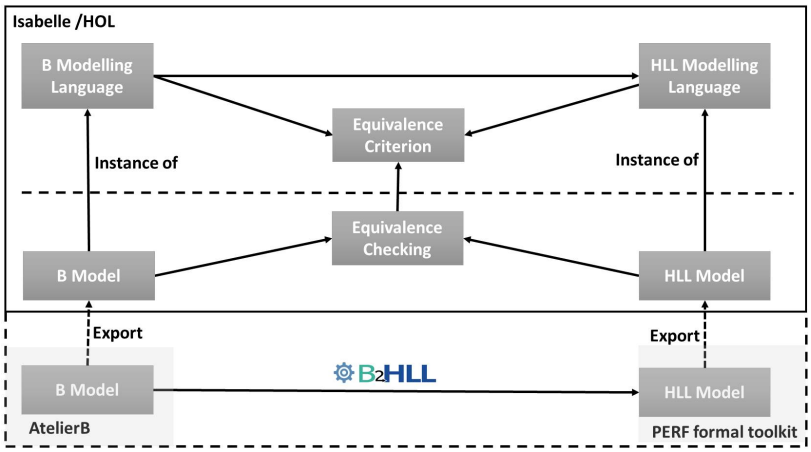

Figure 3.1: A formal framework of certified translator

1. Our approach is based on a **deep embedding** using the Isabelle/ HOL framework as a unified formal modelling framework. First, both the B and HLL modelling language semantics are modelled in Isabelle/ HOL. In the upper part of Fig. 3.1, the formal semantics description of the source and target modelling languages are given in a denotational style. B and HLL models are formalised by state transition systems in order to unify their semantics. Since new state variables and new transitions may appear in the HLL state transition system, it is possible to express properties on these

intermediate states to observe these new variables and transitions. We then give the denotational specification of the transformation functions, which correspond to the central concepts of translating B-models into HLL, as described in Chapter 4 and implemented as rules in the B2HLL tool. This procedure is described in Chapter 5.

2. An equivalence relation enabling to semantically compare B programs with HLL models is formally defined (upper part on Fig. 3.1). Within our project, we consider two programs as semantically equivalent if they have the same output traces. Formally we require both B programs and HLL models to be in a bi-simulation relation. We define an *equivalence criterion* and prove that variables and data flows have corresponding values during each execution step (transition) of B and HLL models. An equivalence theorem is stated and proved (by a structural induction) once for all. In particular, due to the occurrence of new state variables and transitions in HLL state transitions systems, a weak bi-simulation relationship is used as a comparison relationship (Chapter 5).

3. B and HLL models are checked to be equivalent. Both B and HLL specific models are defined as instances of the formal semantic models (Instance of relation on Fig. 3.1). Then, the equivalence theorem has to be checked for these two instances by discharging the associated proof obligations successfully. We use in Chapter 6 the animation approach, including proof steps to validate the formalised models. The associated proofs related to equivalence checking certify that functional representation in modelling languages satisfies the translator specification of the original model.

4. Finally, an export tool (lower part of Fig. 3.1) produces Isabelle/HOL models for the specific input B models and HLL models produced by B2HLL tool.

The developed framework enables to:

- Define a set of sound transformation rules from B language to HLL.
- Guarantee the semantic preservation of the transformation of B models to HLL that can be used for tool certification purpose.
- Develop a tool that implements the defined transformation from B to HLL.

3.4 Considered B language

Our target is the translation of models from the last level of refinement namely the *implementation* level. This is the final entry model that code generators can use, but it still requires a significant amount of manual work to translate all B concepts. To get closer to the target of the code generator, an automatic refinement can be applied.

Figure 3.2 presents the subset of the B language to be transformed to HLL. The definition of the syntax is adapted from [151]. Note that at *implementation* level, only a subset of B language can be used, the substitutions are concrete and the data must have implementable types.

In the transformation function, we have defined the following syntactic categories: component, top-level clauses definition, expressions and substitutions. For the translation process, we consider that a B *implementation* $< Component >$ has a simplified structure, in which $< StructureDef >$ introduces B composition clauses and defines the B components associated with it, $< DataDef >$ describes constants and properties, $< StateDef >$ defines the state variables (state of the system), their types and invariants, $< Init >$ introduces the initial values of the state variables, and $< Ops >$ defines the set of operations that may modify state variables (transitions).

$< Subst >$ describes the forms of substitutions allowed in operations of an *implementation* component. Conditional statements like *if* and *case* may have different forms, e.g., with or without *else* branching. The statement block `BEGIN <Subst> END` is semantically equivalent to the substitution *<Subst>*. The identity substitution `skip` has no effect on the internal state of the B machine.

3.5 Toy Example

In this section, we present an example of a B model and its corresponding HLL code. This example will be used in the Chapter 4 to demonstrate how the transformation rules from B to HLL are applied.

3.5.1 B Development

The below implementation describes a simplified B machine that reads input values from an external machine and computes the minimum of two variables. This example contains two B

$< Component >$::=IMPLEMENTATION $Name$
$< StructureDef >< DataDef >< StateDef >$
$< Init >< Ops >$
END
$< StructureDef >$::= IMPORTS $Name^+$ | SEES $Name^+$
$< DataDef >$::= SETS $< Set >^*$ | CONSTANTS $Name^+$
| PROPERTIES $< Pred >$ | VALUES $< Values >$
$< StateDef >$::= CONCRETE_VARIABLES $Name^+$ | INVARIANT $< Pred >$
$< Init >$::= INITIALISATION $< Subst >$
$< Ops >$::= OPERATIONS $< Operation >^*$
$< Operation >$::= $[Name^+]$ $\leftarrow$ $Name$ $[(Name^+)]$ $=< Subst >$
$< Subst >$::= $Name :=< Exp >$
| $< Subst >; < Subst >$
| $< IfSubst >$
| WHILE $< BExp >$ DO $< Subst >$ INVARIANT $< BExp >$
VARIANT $< Exp >$ END
| $[Name^+]$ $\leftarrow$ $Name$ $[(Name^+)]$
| CASE <Exp> OF EITHER <Exp> THEN <Subst> OR <Exp> THEN <Subst>
ELSE <Subst> END
| VAR Name IN <Subst> END)
| BEGIN <Subst> END)
| (skip)
$< IfSubst >$::= IF <BExp> THEN <Subst> ELSE <Subst> END
| IF <BExp> THEN <Subst> END
| IF <BExp> THEN <Subst>
ELSIF <BExp> THEN <Subst> ELSE <Subst> END
$< Set >$::= $Name$ | $Name = \{Name^+\}$
$< Exp >$::= $< AExp >$ | $< BExp >$ | $Name$
$< Pred >$::= B $Predicates$
$< BExp >$::= $Boolean$ $expression$
$< AExp >$::= $Arithmetical$ $expression$
$< Values >$::= B $Valuing$

Figure 3.2: Considered B language subset. The superscript operators $^+$ and * denote respectively a comma-separated and semicolon list of elements of the annotated element.

machines: `Main_i` defines the main program and `Utils_i` defines auxiliary operations. The `Main_i` implementation represents an entry point of the execution. `Main` is an operation to select an order of the execution using defined operations in the imported machine. In the example, firstly, the operation `computeSum` is called, it changes the state of the machine `Utils_i` as a side effect. The variable `xx` is initialised using the output of the operation `readVar`. This operation returns the value of a variable modified when `computeSum` is called. Finally, the minimum of two variables is computed using the operation `minimum`.

```
IMPLEMENTATION Main_i REFINES Main IMPORTS Utils
CONCRETE_VARIABLES xx, yy, rr
INVARIANT xx ∈ NAT ∧ yy ∈ NAT ∧ rr ∈ NAT
INITIALISATION xx := 0 ;
yy := 0 ;
rr := 0
OPERATIONS
  Main =
   computeSum; xx <-- readVar; rr <-- minimum (xx , yy)
  END
END
```

Listing 3.1: Main Implementation

```
IMPLEMENTATION Utils_i REFINES Utils
CONCRETE_VARIABLES sum
INVARIANT sum ∈ NAT
INITIALISATION sum := 0
OPERATIONS
 rr <-- minimum (aa, bb) =
 IF aa ≥ bb THEN rr := bb ELSE rr := aa END;
 computeSum =
  VAR ii IN ii := 0;
  WHILE ii < 2 DO
    ii := ii + 1; sum := sum + ii;
  INVARIANT ii ∈ NAT ∧ ii ≤ 2 VARIANT 2 − ii
  END END;
 rr <-- readVar =
  rr := sum
END
```

Listing 3.2: Utils Implementation

3.5.2 HLL Development

This section describes the HLL model that would result from translating the B example of Section 3.5 on listings 3.1 and 3.2. The produced HLL model contains two namespaces, one corresponding to the translation of the `Main_i` machine and another for the translation of the imported machine `Utils_i`. For each B operation, a corresponding HLL namespace section is created, such as `"Main"` which contains the translation of the B operation Main.

```
1  Namespaces: Main_i{ // B: Main_i implementation
2  Declarations:
3  int xx; int yy; int rr; int xx_0; int yy_0;int rr_0;
4  Definitions:  xx_0 := 0; yy_0 := 0; rr_0 := 0;
5  xx := Main::xx_1; // B: xx <-- readVar;
6  yy := Main::yy_0;
7  rr := Main::rr_1; // B: rr <-- minimum(xx,yy)
8  Namespaces: Main{ // B: Main operation
9  Declarations: int xx_0; int yy_0; int rr_0;
10 Definitions:
11 xx_0 := Main_i::xx_0; // Maps the initial values of variables
12 yy_0 := Main_i::yy_0;
13 rr_0 := Main_i::rr_0;
14 xx_1 := Utils_i_0::readVar_0::rr; // Operation call
15 mm_1 := Utils_i_0::minimum_0::rr; // Operation call
16 }}
```

Listing 3.3: HLL Translation of Main Machine

HLL is SSA based (Single State Assignment), therefore a stream can only be assigned once in a model. As stated in [100], when converting from a programming language to SSA form, assignments of a program variable are replaced by assignments to new versions of the variable. Thus, each B assignment is translated into an HLL assignment with a new version of the modified variable. The value of the original variable is replaced by the value of the last known version of that variable (similar to dataflow streams).

Line 3 defines the variables used for the translation of the machine `Main_i` with their corresponding type. Line 4 and lines 8-16 represent the computation performed in `INITIALISATION` and `OPERATIONS` blocks of the B machine, respectively. In line 14, the output of the operation `readVar` is assigned to the local variable `"xx<1>"`. Note that state variables are necessary to memorise the final values of variables after the execution of the operation `Main` (lines 5-7). As the operation call `computeSum`, does not modify the state of variables in the machine `Main_i`, its translation is not present in the `Main` namespace. Lines 21-35 represent the translation of the first call of `computeSum`.

```
1  Namespaces: Utils_i_0{ // B:Utils_i implementation
2  Declarations: int sum_0;  int sum_1;
3  Definitions:  sum_0 := 0;
4  sum_1 := computeSum_0::sum;
5  Namespaces: computeSum_0{ // First call of B: computeSum operation
6  Declarations:
7  int sum_0; int ii_0; int ii_1; int ii_2; int sum;
8  Definitions:
9  sum_0 := Utils_i_0::sum_0; ii_0 := 0;
10 // While Loop - iter 0
11 ii_1 := ii_0 + 1;
12 sum_1:= sum_0 + ii_1;
13 ii_2 := if ii_0 < 2 then ii_1 else ii_0;
14 sum_2 := if ii_0 < 2 then sum_1 else sum_0;
15 //... Repeat the loop code with new index
16 sum := sum_4;
17 }
18 readVar_0{ // First call of B: readVar operation
19 Declarations: int rr;
20 Definitions: rr:= Utils_i_0::sum_1;
21 }
22 minimum_0{ // First call of B: minimum operation
23 Declarations:
24 int aa_0;int bb_0;int rr;int rr_0;int rr_1;intrr_2;
25 Definitions:
26 aa_0 := Main_i::Main::xx_1; //Mapping of input parameters
27 bb_0 := Main_i::Main::yy_0;
28 rr_0 := bb_0; // IF block substitution
29 rr_1 := aa_0; // ELSE block substitution
30 rr_2 := if aa_0 ≥ bb_0 then rr_0 else rr_1;//IF block
31 rr := rr_2;
32 }}
```

Listing 3.4: HLL Translation of Utils Machine

3.6 Conclusion

In this chapter we have presented the motivation and main goals of our work, the B-PERFect project. We investigate the applicability of PERF, an industrial toolset that allows formal verification of systems independent of their development process, on software developed in B. More precisely, we have outlined a certified process for integrating the B method into the PERF verification framework. This is extended in Chapters 4, 5, 6. Chapter 4, describes the transformation principles from B to HLL. Chapter 5 presents the certification approach

and Chapter 6 describes the implementation of the transformation rules in the B2HLL tool and the application of the tool to a case study. The B language subset under consideration is presented. Finally, we describe a case study used to illustrate the transformation rules presented in Chapter 4.

Chapter 4

Transformation of B implementation to HLL Code

Contents

In this Chapter we describe the general transformation strategy allowing to obtain an equivalent HLL code from concrete formal models based on B language. The detailed transformation rules for static and dynamic B clauses are presented. Also, a set of transformation rules is given for syntactic constructs of B language that appear in B operations. Finally, the transformation of B projects is described.

4.1 Introduction

In the previous chapters, we discussed the importance of applying formal methods in the context of safety critical systems. We briefly described the overall approach proposed to

verify B models using the PERF methodology. Our goal is to translate B models at last level of refinement, namely the *i*mplementation level. In this chapter, we will to describe the transformation approach from B implementations level to HLL models.

To keep track of the original rational in the design of a railway system can be challenging because of the several refinement steps and implementation choices between: the system specification (produced by system experts) and the software implementation (performed by software engineers), more precisely the connection between physical actions and logical objects. Safety specifications can have different levels of refinement and not all requirements are encoded as invariants in the B model or explicitly considered in implementation.

Although the B method has proven its reliability, the assessment of high-level design is difficult to obtain and needs guidance from human expertise. Let us assume that a B model with the complete refinement process has been validated and proven to be correct. This raises the question of whether the B modelling is correct in terms of the high level natural language specifications. To be more precise, the goal of our work is to transform B models into a synchronous formal language, HLL, for additional verification purposes of B models. This approach does not question the B proof.

The work presented in this chapter inspired the rest of the thesis and represents the basis for the implementation of this transformation as the B2HLL tool published in [6].

This chapter will present how to obtain an HLL model from a B model. We define the general transformation scheme for a B project: implementations with their including clauses and the dependent machines obtained from *IMPORTS* and *SEES* clauses. The translation rules for the core language of B including expressions and instructions are described.

4.2 Transformation Principles. From B to HLL

The goal of this work is to obtain HLL modules which are behaviourally equivalent to B implementations. Due to the semantic mismatch, the transformation of B models to HLL models is not straightforward. On the B side, imperative style is used while data flow paradigm with single static assignment form (SSA) is used on the HLL side. This work concerns the transformation from different paradigms widely studied:

- imperative style to a declarative style with single state assignment form
- sequential style to synchronous dataflow style

When defining the translation rules, we focus on several specific technical problems:

1. variables to data streams transformation
2. sequential to SSA form transformation
3. variables tracing
4. semantic preserving transformation

Particular attention must be paid to a number of concepts, such as evolution and updates of variables, loop behaviours or B operations with side-effects, when defining the translation principles.

B models are finite state machines where the state represents a tuple of variables and their values, the transitions may update the state by modifying those values. A B model is considered correct if its state is valid with respect to the defined invariant. More precisely this means that the initialisation takes the component to a valid state and that no invalid state is reached out by applying the transitions from operations. This behaviour shall be preserved in HLL.

The link between B variables and HLL streams has been considered as crucial for semantics preservation of the transformation. In B, variables may evolve during the execution of operations whereas in HLL, they correspond to data streams and they can be evaluated during a cycle. On the HLL side, a specific data flow is defined to record the changes. B state variables become HLL data streams and a correspondence between a B variable and its corresponding stream in HLL is set up.

HLL is a single state assignment language, which means that a variable cannot have multiple values during a cycle. Since B variables can be affected multiple times in an operation, we must flatten the value changes of variables. Applying the proposed approach the following properties must be preserved: (i) all value changes of a variable shall be traced and (ii) generated code shall preserve the semantics of B language.

In B, a sequence represents an action which leads to the next action in a predetermined order. The instructions in B and HLL have the same sequencing delimitation, represented by semicolon but the meaning is not the same. In HLL, the order of instructions is insignificant, each variable can only be assigned once in a temporal cycle and all the variables are evaluated at the same time. Unlike B, HLL does not support loop structures. Therefore, a B loop

should be flattened in the HLL model. In order to facilitate loop flattening, we extract from the B loop variant its maximum number of iterations.

We detail the defined rules of transformation from B to HLL. In the following, we assume that the source B models are correctly type-checked, visibility checks have been performed before translation, and all the proof obligations generated by AtelierB are proved. Source B language constructs represent the entry point of the translation. The translator is based on a set of translation rules to map between source and target languages constructs.

Notations

In this section, we define the notation used to write the transformation rules from B to HLL and the auxiliary defined functions to assist the transformation. First, we introduce the translation environment that stores the correspondences (mapping) between B variables and HLL flows. Then, we set out the definition of the defined transformation functions.

Mapping: Transformation Environment

Most transformation functions we defined are parameterised by a translation environment *Mapping* reflecting the link between B state variables and HLL data flows. In the following definitions, let us consider $proj_1$ and $proj_2$ the first and the second projection of the Cartesian Product.

For each state modification of a B variable x, a fresh HLL data flow is generated and a mapping is created in the tuple environment. Let $Mapping : Var_B \rightarrow (Label_{HLL} \times Label_{HLL})$ be the function that maps B variables to pairs of HLL (read and write streams). We write $\mathcal{M}$(x), where $\mathcal{M} \in Mapping$, to denote the pair (x_1, x_2) of associated HLL streams. For example, the B variable x would be represented as $x \mapsto (x_1, x_2) \in \mathcal{M}$, where x_1 is the associated read stream, used in expressions or predicates translation and x_2 is the write stream used during statements translation. This environment is constructed at the initialisation of the translation of a B machine and updated by the translation functions.

We introduce the mapping composition operator ($\otimes$) to compose two environments. Given two translation environments $\mathcal{M}_1$ and $\mathcal{M}_2$, it maps all B variables to the associated HLL tuple composed of the reading flow from $\mathcal{M}_1$ and the writing flow from $\mathcal{M}_2$. Formally, $\otimes : Mapping \times Mapping \rightarrow Mapping$ composition mappings is defined as if $v \in Var_B$, $r_{HLL} \in Label_{HLL}$ and $w_{HLL} \in Label_{HLL}$ are respectively a B variable and the corresponding

read and write HLL streams and if $v \mapsto (r_{HLL1}, w_{HLL1}) \in \mathcal{M}_1$ and $v \mapsto (r_{HLL2}, w_{HLL2}) \in \mathcal{M}_2$ then $v \mapsto (r_{HLL1}, w_{HLL2}) \in \mathcal{M}_1 \otimes \mathcal{M}_2$.

$$Mapping : Var_B \rightarrow (Label_{HLL} \times Label_{HLL})$$
$$\_ \otimes \_ : Mapping \times Mapping \rightarrow Mapping$$
$$\mathcal{M}(v) = (r_{HLL}, w_{HLL})$$
$$\mathcal{M}_1 \ \otimes \ \mathcal{M}_2 = \mathcal{M}3 \text{ such that } \forall\ v \cdot v \in dom(\mathcal{M}_1) \wedge v \in dom(\mathcal{M}_2)$$
$$\mathcal{M}3[v \mapsto (proj_1(\mathcal{M}_1(v)), proj_2(\mathcal{M}_2(v)))] \text{ where } \mathcal{M}, \mathcal{M}1, \mathcal{M}2, \mathcal{M}3 \in Mapping$$

Figure 4.1: Translation Environment Domain and Operations

Transformation Functions

The source B language represents the transformation entry point. The translator is based on a set of transformation rules mapping source language constructs to target ones. Therefore the translator applies a transformation rule when matching an input construct to produce the corresponding one. We define a general transformation function $T_s(.)$ with a B model and a transformation environment $Mapping$ as input parameters that computes the corresponding HLL model and the updated environment $Mapping$. Specific transformation functions are defined on the syntax construct.

Let $model\_B$ and $model\_HLL$ be the syntactic constructs of B and HLL modeling languages, respectively. Then, for each syntactic B construct $Synt \in model\_B$, we define a transformation function T_{Synt}, that associates a B construct ($Synt$) to its corresponding HLL construct as $T_{Synt} : model\_B \times Mapping \rightarrow model\_HLL \times Mapping$. This function takes into account the previously defined translation environment for variables and streams. The following notation $T_{Synt}\ (S_B)_{\mathcal{M}_B} \doteq (S_{HLL}, M_{HLL})$ is used in the rest of the Chapter to describe applications of the transformation function T_{Synt} with initial and resulting environments $\mathcal{M}_B$ and $\mathcal{M}_{HLL}$.

$$T_{Synt} : model\_B \times Mapping \rightarrow model\_HLL \times Mapping$$
$$T_{Synt}\ (S_B)_{\mathcal{M}_B} \doteq (S_{HLL}, M_{HLL}) \text{ , where } S_B \in model\_B \text{ , } S_{HLL} \in model\_HLL \text{ ,}$$
$$M_B \in Mapping \text{ and } M_{HLL} \in Mapping$$

This transformation is decomposed into a set of transformation functions corresponding to each B construct. All these transformation functions produce an update of the environment. They may be composed using the composition operator ($\circ$).

We define the following transformation functions.

- $T_{component}$ - to translate B components into HLL Namespaces
- $T_{structDef}$ - to translate B composition primitives into HLL Namespaces
- $T_{dataDef}$ - to translate B data clauses to HLL clauses
- $T_{stateDef}$ - to translate B state clauses to HLL streams
- T_{init} - to translate B $INITIALISATION$ clause into HLL stream initialisation
- T_{ops} - to translate B $OPERATIONS$ clause into HLL computations
- T_{cst} - to translate B $CONSTANTS$ clause into HLL *Constants*
- T_{prop} - to translate B $PROPERTIES$ clause into HLL *Constraints* statements
- T_{set} - to translate B $SETS$ clause into HLL types definition
- T_{var} - to translate B $VARIABLES$ clause into HLL stream declaration
- T_{inv} - to translate B $INVARIANTS$ clause into HLL *ProofObligations* encoding safety properties
- T_V - to translate B variable identifiers into HLL dataflow identifiers
- T_E - to translate B expressions into corresponding HLL constructs
- T_I - to translate B substitutions to HLL instructions
- T_P - to translate B predicates from $PROPERTIES$ and $INVARIANTS$ clauses to HLL predicates
- T_S - to translate B sets identifiers to HLL identifiers

This translation requires a couple of helper functions defined below. Some of them are borrowed from [147].

- *GetType* - for type inference when translating a B variable or constant to HLL. For example, $GetType(x) = t$ gives B type t of variable x.

- *GetValue* - to extract the valuation of a B constant or set. $GetValue(x) = N$, extracts the value of the input parameter x.
- *CreateFresh* - to generate a fresh HLL stream label corresponding to a B variable. $CreateFresh(x, \mathcal{M}) = x_{HLL}$ produces a x_{HLL} stream label for the variable x in environment $\mathcal{M}$.
- *GetIterations* - to compute, from the B VARIANT, the number of iterations of a B loop substitution. $GetIterations(whileInstr) = N$, extracts from the $VARIANT$ of the loop substitution $whileInstr$ its number of iterations N.
- *FindModified* - to return a set of modified variables by B substitution. Given a block of B substitutions bl and the translation environment $\mathcal{M}$, $FindModified(bl, \mathcal{M}) = VarsMod$, returns the set of variables $VarsMod$ updated in bl.
- *NameHLL* - $NameHLL(Idf_B, \mathcal{M}) = Idf_{HLL}$ returns the HLL identifier Idf_{HLL} corresponding to a B machine identifier or a B operation Idf_B identifier in the environment $\mathcal{M}$. The returned identifier is composed of the input B name and the instance associated with the component/operation in the translation environment.

4.3 Transformation of B Component

An *IMPLEMENTATION* B component corresponds to a state transition system. In B components, the state of a system is represented as variables, while in HLL it is specified with the declarations of flows. The execution of transitions is expressed in B by dynamic clauses, the same mechanism is represented in HLL by stream values that change over the time frames. In HLL, the transition relation can be represented by formulas in propositional logic.

We recall the considered structure of an $IMPLEMENTATION$ B component in the translation process.

$$< Component >::=< StructureDef >< DataDef >< StateDef >< Init >< Ops >$$

where <StructureDef> introduces B composition clauses, <DataDef> describes constants and properties, <StateDef> defines the state variables (state of the system), their types and invariants, <Init> introduces the initial state variables values and <OPS> defines

the set of operations that may modify state variables (transitions). All the clauses of a B implementation are translated using the transformation functions defined in Sections 4.4, 4.5, 4.6 and 4.7 .

The starting point of the transformation is the implementation component. $T_{component}$ transformation function is applied on each machine inner structure ($DataDef$, $StateDef$, $Init$ and Ops) in an initial environment $\mathcal{M}$ ($\mathcal{M} = \varnothing$, when starting the transformation of the main machine) as follows.

$$T_{component}(Component_B)_{\mathcal{M}} \doteq (Model_{HLL}, \mathcal{M}') \text{ where}$$
$$T_{component} \doteq T_{ops} \circ T_{init} \circ T_{stateDef} \circ T_{dataDef}$$

By applying the $T_{component}$ transformation function, a HLL model is obtained, $model_{HLL}$, and the resulting translation environment $\mathcal{M}$' is constructed automatically based on the initial one $\mathcal{M}$. $Model_{HLL}$ is obtained by concatenating the resulting HLL code from specific transformation functions: $T_{dataDef}$, $T_{stateDef}$, T_{init}, T_{ops}.

We propose to model B components as HLL *Namespaces*, as shown in Table 4.1, since both have a notion of scoping variables and structuring capabilities that lead to data encapsulation. Each B component encapsulates its own internal state with the possibility to share variables with other components. This is not the case with HLL. Therefore, the internal state of a B-machine is preserved by a variable versioning mechanism when the dynamic parts of the B-model are translated.

The state of a B component is obtained by composing the transition relations of the individual statements inside the different operations body, including the operations accessed from other B components. This behaviour is achieved by successive substitutions. Let Var_B be a set of B typed variables defined in the component $Component_B$ and σ_B denote the set of all states over Var_B. Let us define Var_{HLL} and σ_{HLL}, the set of system variables and the system state for $Model_{HLL}$. Var_{HLL} contains the introduced variables due to the versioning mechanism. We expect that B and HLL states to be related by a refinement relation.

In the transformation process, the declarative part of the B component initialises the translation environment $\mathcal{M}$ with the state and the context of this component $Component_B$. The range of the translation environment $\mathcal{M}$ is a tuple of HLL variables Var_{HLL} over the domains of B variables Var_B. In fact, the mapping given by $\mathcal{M}$ allows to connect the value

of every B variable $v \in Var_B$ in B state with HLL state represented by Var_{HLL}. $\mathcal{M}$' is the translation environment $\mathcal{M}$ updated accordingly after the transformation of $Component_B$.

B Construct	HLL Construct
$T_{component}(IMPLEMENTATION\ Name\ Component_B\ END)_{\mathcal{M}} \doteq$	**Let** $T_{component}(Component_B)_{\mathcal{M}} \doteq (ModelHLL, \mathcal{M}')$ and $NameHLL(Name, \mathcal{M}) = Name_{HLL}$ **in** (Namespaces: $Name_{HLL}$ {ModelHLL} , $\mathcal{M}'$)

Table 4.1: Rule: Component Transformation

Example. In Listing 4.1 and Listing 4.2, we can observe that the IMPLEMENTATION *Utils_i* is translated as equivalent to the Namespaces section in HLL with the same name adding the postfix of the current version of the Namespace in the transformation environment.

```
IMPLEMENTATION Utils_i
...//Body
END
```

Listing 4.1: Component Transformation: B Code

```
Namespaces: Utils_i_0{
...//Body
}
```

Listing 4.2: Component Transformation: HLL Code

4.4 Transformation of Static Clauses

In B language, data can be abstract such as sets, relations, cartesian products and it is used mostly in specifications and first level of refinements. Concrete data (enumerated types, booleans, bounded integer types, arrays on finite index interval) are those used in implementations because they can be easily transformed to programming languages types.

<*DataDef*>, representing the data of a B implementation, is composed of the SETS, CONSTANTS, PROPERTIES and VALUES. The transformation of <*DataDef*> is represented as follows:

$$\begin{aligned} T_{dataDef}(< DataDef >)_{\mathcal{M}} = \ & T_{cst}\ (\texttt{CONSTANTS}\ Name)_{\mathcal{M}} \\ & \mid T_{prop}(\texttt{PROPERTIES}\ BExp)_{\mathcal{M}} \\ & \mid T_{set}(\texttt{SETS}\ Set)_{\mathcal{M}} \end{aligned}$$

<*StateDef*>, representing the state space of a B component, consists of VARIABLES and

INVARIANTS clauses. The transformation of <*StateDef*> is represented as follows:

$$T_{stateDef}(< StateDef >)_{\mathcal{M}} = \; T_{var}\ (\texttt{CONCRETE\_VARIABLES}\ Name)_{\mathcal{M}};$$
$$\mid T_{inv}(\texttt{INVARIANTS}\ BExp)_{\mathcal{M}};$$

Table 4.2 presents each transformation function used for <*DataDef*> and <*StateDef*> B clauses and their sub-constructs.

B Construct	HLL Construct
$T_{cst}(CONSTANTS\ c)_{\mathcal{M}}\ \doteq$	**Let** $CreateFresh(c, \mathcal{M}) = c_{HLL}$, GetType(c) = type and GetValue (c) = value **in** (Constants: type $c_{HLL}\ :=\ value;$, $\mathcal{M}[c \mapsto c_{HLL}]$)
$T_{prop}(PROPERTIES\ R)_{\mathcal{M}} \doteq$	**Let** $T_P(R)_{\mathcal{M}}\ \doteq\ (R_{HLL}, \mathcal{M})$ **in** (Constraints: $R_{HLL};$, $\mathcal{M}$)
$T_{set}(SETS\ A)_{\mathcal{M}}\ \doteq$	**Let** $T_S(A)_{\mathcal{M}}\ \doteq\ (A_{HLL}, \mathcal{M})$ and GetType(A) = type **in** (Types: type $A_{HLL};$, $\mathcal{M}$)
$T_{var}(CONCRETE\_VARIABLES\ x)_{\mathcal{M}}\ \doteq$	**Let** $CreateFresh(x, \mathcal{M})\ =\ x_{HLL}$ and GetType(x) = type **in** (Declarations: type $x_{HLL};$, $\mathcal{M}[x \mapsto x_{HLL}]$)
$T_{inv}(INVARIANTS\ I)_{\mathcal{M}}\ \doteq$	**Let** $T_P(I)_{\mathcal{M}}\ \doteq\ (I_{HLL}, \mathcal{M})$ **in** (Proof Obligations: $I_{HLL};$, $\mathcal{M}$)

Table 4.2: Rule: Static Clauses Transformation

In the transformation process, the declarative part of a B component initialises the translation environment $\mathcal{M}$ for this component. B CONSTANTS are translated into HLL Constants. When translating a B constant, the type is inferred from the *PROPERTIES* clause using the *GetType* function. Each concrete constant of the B implementation must be valued in the *VALUES* clause. The valuation of a constant is extracted using the *GetValue* function. B PROPERTIES are translated as HLL Constraints. The predicate R that compose this clause is translated using the T_P translation function. The SETS clause defines a list of deferred or enumerated sets. At implementation level, the deferred sets must be valued to a finite, non-empty set. The T_{Set} function transforms a B set to HLL type definition. The concrete information related to a B set is gathered from the VALUES clause. We can observe that the VALUES clause doesn't have a standalone translation function. This is due to the fact that the valuations of constants or sets are used in the transformation of CONSTANTS and SETS clauses.

The VARIABLES clause is modelled in HLL by the stream declaration section of the

HLL namespace associated with the B machine. Variables type is inferred from the given invariant in the INVARIANT clause. The translation environment $\mathcal{M}$ is initialised with the set of B variable x and their corresponding HLL streams x_{HLL}.

B invariants are introduced as a constraint predicate over the state space of a component. Typing invariants are modelled in HLL as *Constraints* predicates and the safety ones become HLL *ProofObligations*. In HLL, such a predicate is expressed using *Boolean* as the type of the predicate variable, I_{HLL}, as defined in the HLL *ProofObligations* section. The difference between B *invariants* and HLL *constraints* is that for the former, verifications are carried out to check that no invalid state will ever be reached, as long as the operations are used as specified. On HLL side, *Constraints* clause, has the role to reduce the state space exploration only on valid states given by the predicates expressed on streams in this clause. This is useful to reduce the exploration space of properties to prove in the *ProofObligations* clause if the set of *constraints* is valid and consistent.

Example. In Listing 4.4 we illustrate the translation of the declaration of B variables from Listing 4.3. For example, when we declare a B variable xx of type NAT, it becomes the first instance of the stream *xx_0* in the HLL. The variable is typed according to its B type NAT. Note that at the implementation level B integers take values in the interval MININT..MAXINT. This behaviour is taken into account during type translation in HLL. All generated HLL versions of the variable xx are declared in the same way.

```
SETS t_set
VALUES t_set = 0 .. cst_set
CONCRETE_VARIABLES xx
INVARIANT xx ∈ NAT
```

Listing 4.3: Static Clauses Transformation: B Code

```
Constants :
 int MAXINT := 2147483647;
Types:
 int [0, cst_set_0] t_set;
 int [0, MAXINT] NAT;
Declarations:
 NAT xx_0 ;
```

Listing 4.4: Static Clauses Transformation: HLL Code

4.5 Transformation of Dynamic Clauses

In B language, the dynamic parts of the components are modelled by substitutions, which allow the modification of the data space of a model. Substitutions are used in *INITIALISATION* and *OPERATIONS* clauses of a B machine. The proposed transformation of B

substitutions is based on the understanding of the semantic differences between HLL and B. The general form of an operation is: $out \leftarrow op\_name(in) \widehat{=} S$, where *in* and *out* can be variables or lists of variables representing the parameters of the operation *op_name*, and they are optional.

The translation pattern for dynamic clauses of B machines is given in Table 4.3:

- The INITIALISATION clause is translated in HLL as a *Definitions* section and sets the initial value for streams.
- Each B operation from the clause OPERATIONS is transformed into the HLL *Namespaces* section. The name of the HLL namespace is specified by the $NameHLL$ function and represents the identifier of the operation with an appended suffix. The translation of the input and output parameters is implemented by declaring the variables within the operation namespace. The translation of the operation precondition specifies the type of the input parameters. The generalised B-substitutions from the operation body are translated using the function T_I. More details are given in Section 4.6.3.

B Construct	HLL Construct
$T_{init}(INITIALISATION\ v := E)_{\mathcal{M}} \doteq$	**Let** $T_V(v)_{\mathcal{M}} \doteq (v_{HLL}, \mathcal{M})$ and $T_E(E)_{\mathcal{M}} \doteq (E_{HLL}, \mathcal{M})$ **in** (Definitions: I(v_{HLL}) := E_{HLL}; , $\mathcal{M}$)
$T_{ops}(OPERATIONS\ x_{out} \leftarrow opName(y_{in}) = S)_{\mathcal{M}} \doteq$	**Let** $T_I(S)_{\mathcal{M}} \doteq (S_{HLL}, \mathcal{M}')$ and $Name_{HLL}(opName, \mathcal{M}) = opName_{HLL}$ **in** (Namespaces: $opName_{HLL}$ { S_{HLL} } , $\mathcal{M}$')

Table 4.3: Rule: Dynamic Clauses Transformation

Example. Each initialisation and operation in Listing 4.5 is translated as shown in Listing 4.6. Inside the namespace associated to the translation of the corresponding machine, we define a new *Namespaces* section *minimum_0* that will contain the translation of an operation.

```
INITIALISATION
sum := 0
OPERATIONS
  rr <-- minimum (aa, bb) = ...
```

Listing 4.5: Dynamic Clauses Transformation: B Code

```
// M: sum -> (0,0)
Definitions:
sum_0 := 0;
Namespaces: minimum_0{...}
```

Listing 4.6: Dynamic Clauses Transformation: HLL Code

4.6 Transformation of constructs from B operations

This section is dedicated to define a set of rules that allows us to transform the operations of B models into HLL models. First, we present the transformation of *expressions* used in substitutions and then we describe the transformation rules for *substitutions*. The constructs presented in this chapter are the basic constructs that we identified for the purpose of certifying transformation from B to HLL. The syntactic categories associated with the B syntax, which will be used in B operations, are: identifiers ($Name$), expressions ($AExp$), conditions ($BExp$) and substitutions ($Subst$).

4.6.1 Transformation of Variables

HLL is a single state assignment language, meaning that a variable cannot have multiple values in a single cycle. The translation environment $\mathcal{M}$ stores the corresponding HLL stream identifier for each B variable. Note that in a B operation, since variables may be updated several times, it is necessary to flatten and trace variable-value changes by introducing intermediate variables.

To handle the variable translation from different B clauses to HLL, we associate for each B variable $varId$ a unique pair (r_{HLL}, w_{HLL}), in the translation environment. Any use of $varId$ without changing the value of $varId$ is replaced by the use of its current reading value r_{HLL}. The writing version of w_{HLL} is used when clauses modifying the internal state of the machine are translated. The function T_V accepts as input the identifier of a B variable and returns its corresponding HLL stream identifier in the $\mathcal{M}$ environment by merging the B variable name and its associated HLL label, the reading value w_{HLL} as shown in Table 4.4. The corresponding identifier for the B variable $varId$ in the $\mathcal{M}$ environment is the B identifier followed by the HLL label: $varId\_r_{HLL}$. Constants are translated using the same rule as variables except that their HLL label remains unchanged.

B Construct	HLL Construct
$T_V(varId)_{\mathcal{M}} \doteq$	**Let** $proj_1(\mathcal{M}(varId)) = r_{HLL}$ **in** ($varId\_r_{HLL}$, $\mathcal{M}$)

Table 4.4: Rule: Variable identifier to Stream identifier Transformation

Example. In Listings 4.7 and 4.8, we exemplify the translation of B variable identifier. A B variable xx is translated as the HLL label corresponding to its version *xx_1*.

```
xx  //Variable
cc  //Constant
```

Listing 4.7: Variable Transformation: B Code

```
xx_1 // M: xx -> (1,1) cc -> (1,1)
cc_1
```

Listing 4.8: Variable transformation: HLL Code

4.6.2 Transformation of Expressions

The language used in B expressions relies on predicate logic and set theory. At the implementation level, B expressions are classical arithmetic and Boolean expressions that occur in programming languages. The transformation rules for expression from *IMPLEMENTATION* substitution are described below.

An arithmetic expression of B is a mathematical formula containing constants, variables, and operators. The supported arithmetic operators are: $+$, -, $\times$, $\div$. In B, integer variables are bounded and must respect the predefined constant interval: $MININT..MAXINT$. Boolean expressions are evaluated in B as $true$ or $false$ and used in variable assignment,in *if substitution condition* or in *while loop condition*. Boolean operators are $\wedge$, $\vee$, $\neg$. The expressivity of B in the implementations is restricted to avoid constructions that might overflow at run-time (for example: *IF (xx + 1 <3)*).

In the proof process, specific lemmas are generated in B to ensure that the model is well-defined. Thus, it is checked that the variables in arithmetic expressions are of the same type and that, in the case of division, the denominator is different from zero. Behm et al. [152] had formalised the well-definedness of B-models. Since the target of our transformation is already proven B models, we are not interested in transforming the generated well-definiteness proof obligations.

On the other hand, in HLL, arithmetic operations such as division by zero are considered undefined behaviour and are checked at the solver level. By proving this proof obligations, the HLL model can be statically validated against overflows, the use of uninitialised variables, and out of bound array accesses.

The transformation of B expressions is specialised for identifiers ($Name$), arithmetic expressions (<*AExp*>) and Boolean expressions (<*BExp*>). The T_E function takes a B expression as input and returns the corresponding HLL expression. Since these expressions are available in HLL, their translation is straightforward and defined as follows.

$$
\begin{aligned}
T_E(< Exp >)_{\mathcal{M}} = \; & T_V\ (Name)_{\mathcal{M}}; \\
& \mid T_{Aexp}(\text{<AExp>})_{\mathcal{M}}; \\
& \mid T_{Bexp}\ (\text{<BExp>})_{\mathcal{M}};
\end{aligned}
$$

The translation rules for B expressions can be found in Table 4.5. We introduce the function O that associates B operators (arithmetic or boolean) to the corresponding HLL operators. Translating B operators is straightforward because HLL provides the same operators as B [146]. By applying the T_E function to each expression and the O function to the operator, binary expressions are translated. The translation environment is not modified by this transformation.

B Construct	HLL Construct
$T_E(expr1\ op_B\ expr2)_{\mathcal{M}} \doteq$	**Let** $T_E(expr1)_{\mathcal{M}} \doteq (expr1_{HLL}, \mathcal{M})$ and $O(op_B) = op_{HLL}$ and $T_E(expr2)_{\mathcal{M}} \doteq (expr2_{HLL}, \mathcal{M})$ **in** ($expr1_{HLL}\ op_{HLL}\ expr2_{HLL}$, $\mathcal{M}$)
$T_E(op_B\ expr1)_{\mathcal{M}} \doteq$	**Let** $O(op_B) = op_{HLL}$ and $T_E(expr1)_{\mathcal{M}} \doteq (expr1_{HLL}, \mathcal{M})$ **in** ($op_{HLL}\ \ expr1_{HLL}$, $\mathcal{M}$)

Table 4.5: Rule: Expressions Transformation

Example. Listing 4.10 show the translation of some arithmetic and Boolean operators, present in Listing 4.9, borrowed from the case study detailed in Chapter 3.

```
ii + 1
sum - ii
ii != sum
```

Listing 4.9: Expression Transformation: B Code

```
// M: ii -> (1,1) , sum -> (1,1)
ii_1 + 1
sum_1 - ii_1
ii_1 != sum_1
```

Listing 4.10: Expression Transformation: HLL Code

4.6.3 Transformation of Substitutions

In B language, the dynamic parts of the components are modelled by substitutions, which allow to modify the data space of a model. Substitutions are used to describe *INITIALISATION and OPERATIONS* clauses of B machine. We show the transformation of substitutions that can be used in an *IMPLEMENTATION* component. These substitutions are similar to

the statements of the classical procedural languages, and their proposed transformation is similar to the ones proposed in the literature for programming language statements to SSA form transformation.

The control flow graph of B substitutions from an *IMPLEMENTATION* is shown in [101]. In contrast to this, we do not construct the control flow graph of B substitutions for their SSA form. We use the meaning of the CFG and we rely on a different theoretical foundation to express the transformation and to prove its correctness. The proposed transformation of B substitutions is based on the understanding of different semantics of HLL and B. For each type of substitution, we apply specific transformation rule described below.

The B language is based on set-theoretic notations and it includes notations for expressing *transitions over states* of a model: generalised substitutions. The T_I function takes the different syntactic constructs to write B *operations* body as input and returns the corresponding HLL code. The transformation of B substitutions is defined below.

$$
\begin{aligned}
T_I(< Subst >)_{\mathcal{M}} = \; & T_{asg}\ (\texttt{x := E})_{\mathcal{M}}; \\
& \mid T_{seq}(\texttt{<Subst>;<Subst>})_{\mathcal{M}}; \\
& \mid T_{if}\ (\texttt{<IfSubst>})_{\mathcal{M}}; \\
& \mid\ T_{while}\ (\texttt{WHILE P DO S INVARIANT I VARIANT V END})_{\mathcal{M}}; \\
& \mid T_{op}\ (\texttt{nameOut} \leftarrow \texttt{opname(nameIn)})_{\mathcal{M}}; \\
& \mid T_{case}\ (\texttt{CASE E OF EITHER E1 THEN S1 OR E2 THEN S2} \\
& \texttt{ELSE S3 END})_{\mathcal{M}}; \\
& \mid (\texttt{skip})_{\mathcal{M}};
\end{aligned}
$$

The T_I function takes as input a subset of B substitutions that is sufficient to describe the transformation from B to HLL. The remaining substitutions from an implementation represent syntactic sugar and can be translated using the defined transformation rules.

For example, the local variable declaration that has the general form *VAR nameList IN S END*, where *nameList* represents a variable or a list of variables and S represents a substitution. This substitution is transformed in HLL as a variable declaration block followed by the transformation of the substitution S given by T_I function.

Assignment Substitution

Assignment translation from B to HLL is similar to the translation of imperative programs to SSA style. By definition, in SSA form, it is required to represent a program by elementary operations such that there is exactly one assignment for each variable [100]. Following [153], the SSA style is obtained by indexing uniquely each assignment and replacing all occurrences of variables to match their assignment's new name.

The transformation from B to HLL assignments uses the T_{asg} function and it is shown in Table 4.6. Every assignment of a B variable x generates a unique and fresh corresponding HLL identifier x_HLL using the $CreateFresh$ function. This new identifier is defined with an incremented index value, if it already exists in the translation environment or adds to it a new binding stream for the B variable if not. All the B variables are translated in HLL stream variables with same types. As stated before, in order to trace the value changes of a variable it is required to store in the translation environment the correspondence between a B variable and its HLL *current version*. The state variables are unfolded in order to observe all the intermediate variables occurring in state changes. Therefore, new state variables and new transitions may appear on the HLL state transitions system. After the assignment, the translation environment ($\mathcal{M}$) is updated, x_{HLL} is the current HLL label of x.

B Construct	HLL Construct
$T_{asg}(x := E)_{\mathcal{M}} \doteq$	**Let** $CreateFresh(x, \mathcal{M}) = x_{HLL}$ and $T_E(E)_{\mathcal{M}} \doteq (E_{HLL}, \mathcal{M})$ **in** ($x\_x_{HLL} := E_{HLL}$; , $\mathcal{M}[x \mapsto x_{HLL}]$)

Table 4.6: Rule: Assign Transformation

Example. In Listing 4.12, we exemplify the translation of B assignments presented in Listing 4.11. For example, assignment of *ii*, generates a new HLL label *ii_1*, with *1* as a unique index for this variable. The subsequent use of *ii* are replaced by its current instance.

```
ii := 0;
rr := bool (ii + 1 > sum);
```

Listing 4.11: Assign Transformation: B Code

```
// M: ii -> (0,0) , rr -> (0,0)
ii_1 := 0;
// M: ii -> (1,1) , rr -> (0,0)
rr_1 := (ii_1 + 1 > sum_1);
// M: ii -> (1,1) , rr -> (1,1)
```

Listing 4.12: Assign Transformation: HLL Code

Sequence Substitution

Sequence defines transformation rules for a sequence of instructions which can be decomposed inductively as translation of the first instruction followed by the translation of the remaining set of instructions. The instructions in B and HLL have the same sequencing delimitation, represented by semicolon. The transformation from B to HLL uses the T_{seq} function as shown in Table 4.7. It uses an input environment and produces an output one. At each transformation step, the output environment of the previous transformation ($\mathcal{M}_1$) is the input environment of the next one (continuation passing style). This process is repeated until a final translation environment is generated ($\mathcal{M}_2$).

B Construct	HLL Construct
$T_{seq}(S1; S2)_{\mathcal{M}} \doteq$	**Let** $T_I(S1)_{\mathcal{M}} \doteq (S1_{HLL}, \mathcal{M}1)$ and $T_I(S2)_{\mathcal{M}1} \doteq (S2_{HLL}, \mathcal{M}2)$ **in** ($S1_{HLL}; S2_{HLL}$; , $\mathcal{M}2$)

Table 4.7: Rule: Sequence Transformation

Example. Listing 4.14 represents the transformation of the B sequence of Listing 4.13 into HLL and the updated translation environment $\mathcal{M}$ after each assignment. As stated before, we store a reading and writing version of each B variable, *ii (0, 0)*, in the translation environment and update it accordingly.

```
ii := ii + 1;
sum := sum + ii;
```

Listing 4.13: Sequence Transformation: B Code

```
// M: ii -> (0,0), sum -> (0,0)
ii_1 := ii_0 + 1;
// M1: ii -> (1,1), sum -> (0,0)
sum_1 := sum_1 + ii_0;
// M2: ii -> (1,1), sum -> (1,1)
```

Listing 4.14: Sequence Transformation: HLL Code

Conditional Substitution

The translation of IF statements from B to HLL can be compared to the one of imperative programs with joined nodes to the SSA form. Each assignment to a variable must be unique. Therefore, different branch values of a variable must be merged. In the literature, the conversion of control structures such as *IF* statement is realised by adding a different type of assignment, the so called $\phi - functions$ [100, 153]. This function contains the list of values of a variable that can be reached at the end of the IF via different branches. This comparison

should not be taken too literally because we do not propose a graph-based approach. We generate on the fly the *merging nodes* using a functional store approach [154] by accessing several independent variable state memories (translation environment $\mathcal{M}$).

In HLL, *IF conditional* is an expression while it is a substitution in B. *Conditional statement* has several forms in B. In Table 4.8, we present the translation rules for various forms of the *conditional substitution* given by T_{if} function. The translation is achieved in three steps.

B Construct	HLL Construct
T_{if}(IF C THEN S1 ELSE S2 END)$_{\mathcal{M}} \doteq$	**Let** $T_I(S1)_{\mathcal{M}} \doteq (s1_{HLL}, \mathcal{M}1)$ and $T_I(S2)_{\mathcal{M}\otimes\mathcal{M}1} \doteq (s2_{HLL}, \mathcal{M}2)$ and $FindModified(S1) = V1,\ FindModified(S2) = V2,\ V = V1 \cup V2$ and $(s3_{HLL}, \mathcal{M}3) = \{\forall v \in V.\ v_{HLL} = CreateFresh(v, \mathcal{M}2) \wedge$ $v_{HLL} := if\ T_E(C)_{\mathcal{M}}\ then\ proj_1(\mathcal{M}1(v))\ else\ proj_1(\mathcal{M}2(v));\}$ **in** ($s1_{HLL}; s2_{HLL}; s3_{HLL};$, $\mathcal{M}3$)
T_{if}(IF C THEN S1 END)$_{\mathcal{M}} \doteq$	T_{if}(IF C THEN S1 ELSE SKIP END)$_{\mathcal{M}}$
T_{if}(IF C1 THEN S1 ELSIF C2 THEN S2 ... ELSIF Cn THEN Sn ELSE S END)$_{\mathcal{M}} \doteq$	**Let** S_{elsif} = IF C2 THEN S2 ... ELSIF Cn THEN Sn ELSE S END **in** T_{if}(IF C1 THEN S1 ELSE S_{elsif} END)$_{\mathcal{M}}$

Table 4.8: Rule: Conditional Transformation

- First, the $S1$ and $S2$ blocks of instructions of each *IF* branch are translated to $s1_{HLL}$ and $s2_{HLL}$ block of instructions. The transformation of $S2$ block is realised using the composed transformation environment $\mathcal{M} \otimes \mathcal{M}1$, where $\mathcal{M}$ is the initial transformation environment and $\mathcal{M}1$ the environment obtained after $S1$ transformation.
- Second, using the *FindModified* function, the set of modified variables (V) within the IF substitution blocks is computed.
- Last, the final assignments merging the *IF* branch values for all modified variables are produced. These assignments represent HLL *IF* expressions taking into account the condition evaluated initially and the results for both *if* and *else* branches substitutions translation. The variables evaluation is realised in different environments because we should preserve the initial state of the condition. The final block of assignments ($s3_{HLL}$) is generated for modified variables only.

The final result of the translation is the block of HLL instructions composed of previous ones ($s1_{HLL}, s2_{HLL}, s3_{HLL}$) and the translation environment $\mathcal{M}3$.

In B, the *else* branch is optional and its absence is semantically equivalent to identity substitution. In HLL, the use of *IF* without *else* branch is not possible. So, during the

translation of *IF* substitution without *else* branch, we need to add a value for the *else* branch. This is equivalent with copying the current state of variables in the initial translation environment $\mathcal{M}$ as shown in Table 4.8, the second rule. B offers a shortcut for nesting *IF substitutions* in the *else* branch: the *ELSIF* branch. The translation of this *IF* form is equivalent with translating an *IF* where the substitution of its *else* branch is the *ELSIF substitution*.

Example. Listings 4.15 and 4.16 exemplify the transformation of IF substitutions into HLL. Firstly, the body of *IF* substitution is translated: the assignment that set the value of *rr* for both branches. The *rr_1* and *rr_2* HLL variables are defined. Secondly, a new variable, *rr_3*, is defined to represent translation of final *rr* value and it is expressed in HLL *IF* expression. The condition of *IF* is also translated with respect to the version of variables in the translation environment. The second *IF* is translated by copying, for the *else* branch, the value of *rr* in the corresponding translation environment: *rr_3*. The nested form of *IF* is translated following the same principles as for the simple form of *IF*.

```
IF aa ≥ bb THEN rr := bb
             ELSE rr := aa
END
//———IF without ELSE ———
IF aa ≥ bb THEN rr := bb
END
//———Nested IF ————————
IF aa = bb     THEN rr := aa
ELSIF aa > bb THEN rr := aa + 1
ELSE rr := aa − 1
END
```

Listing 4.15: IF Transformation: B code

```
// M: rr -> (0,0), aa -> (0,0), bb -> (0,0)
rr_1 := bb_0; // IF
rr_2 := aa_0; // ELSE
rr_3 := if aa_0 ≥ bb_0 then rr_1 else rr_2;
//———IF without ELSE ———
rr_4 := bb_0; // IF
rr_5 := if aa_0 ≥ bb_0 then rr_4 else rr_3;
//———Nested IF ————————
rr_6 := aa_0;    // First IF
rr_10 := if aa_0 = bb_0 then rr_6 else rr_9;

rr_7 := aa_0 + 1; // Second IF
rr_8 := aa_0 − 1; // ELSE
rr_9 := if aa_0 > bb_0 then rr_7 else rr_8;
```

Listing 4.16: IF Transformation: HLL code

Case Substitution

The Case substitution defines a choice in a block of substitutions depending on the value of an expression. If none of its branches is selected, the *else* branch is executed. The absence of the *else* branch is equivalent to identity substitution. We propose to rewrite the *case* statement to nested *IF* statements in the obvious way. This is specified in Table 4.9 and the T_{if} transformation function is applied further to obtain the equivalent HLL code. Note that

B case alternatives are more complex in B then in HLL, therefore they are translated as *IF* statement.

B Construct	HLL Construct
T_{case}(CASE E OF EITHER E1 THEN S1 OR E2 THEN S2 ELSE S3 END)$_{\mathcal{M}}$ $\doteq$	**Let** S_{case} = (IF E = E1 THEN S1 ELSIF E = E2 THEN S2 ELSE S3 END) **in** $T_{if}(S_{case})_{\mathcal{M}}$

Table 4.9: Rule: Case Transformation

Loop Substitution

The general form of a loop construct in B is WHILE C DO S INVARIANT I VARIANT V END, where S is a substitution, C is a boolean expression, I is a loop invariant, and V is a variant to guarantee loop termination. In B, a *while loop* must end after a finite number of iterations. Unlike the B language, HLL does not support loop structures. Therefore, a B *loop* shall be flattened in the HLL model. We propose to translate a *while loop* as the IF substitution transformation shown in Table 4.10. In order to do so, the B VARIANT

B Construct	HLL Construct
T_{while}(WHILE P DO S INVARIANT I VARIANT V END)$_{\mathcal{M}}$ $\doteq$	**Let** $GetIterations(V) = N$, $S_{IF} = S_0; \ldots ; S_N$ where S_i = IF P THEN S END, $0 \leqslant i \leqslant N$ $T_{if}(S_{IF})_{\mathcal{M}} \doteq (while_{HLL}, \mathcal{M}_{HLL})$ and $T_E(P)_{\mathcal{M}_{HLL}} \doteq (exit_{HLL}, \mathcal{M}_{HLL})$ **in** (Definitions: $while_{HLL}$; Proof Obligations: $\neg\ exit_{HLL}$; , $\mathcal{M}_{HLL}$)

Table 4.10: Rule: While Transformation

clause is exploited, using the *GetIterations* function, to get the number of iterations needed to exit the loop. Let N be the number of loop iterations and S_{IF} the sequence of $S_0; \ldots S_N$ B substitutions where S_i represents the execution of single loop iteration. We use the transformation function T_{if} N times, if N is known as a constant, or define a recursive HLL IF statement producing the HLL code $while_{HLL}$ and the final translation environment $\mathcal{M}_{HLL}$. Thus when i is equal to the total number of iterations, the translation of S_i produces the value of variables after the loop has run to completion. The loop body S and the condition P are translated using their syntactic translation functions. The $exit_{HLL}$ is the translation of the *loop* condition when loop is terminated. For each loop we add a *proof obligation* ensuring that the loop condition is false at the end of the loop. Variant and loop invariant are translated to HLL *Proof Obligations* clause. As the purpose of this

translation is safety properties verification, the loop invariant translation is helpful to check intermediate states consistencies.

Example. Listings 4.17 and 4.18 show the translation of the while loop. Note that the translated HLL code only shows the first iteration of the while loop.

```
VAR ii IN
 ii := 0;
 WHILE ii < 2 DO
     ii := ii + 1;
     sum := sum + ii
 INVARIANT ii ∈ NAT ∧ ii ≤ 2
 VARIANT 2 − ii
 END
END
```

Listing 4.17: While Transformation: B Code

```
// M: ii -> (0,0), sum -> (0,0), bb -> (0,0)
Definitions:
ii_0 := 0;
// While Loop − iter 0
ii_1 := ii_0 + 1;
sum_1:= sum_0 + ii_1;
ii_2 := if ii_0 < 2 then ii_1 else ii_0;
sum_2 := if ii_0 < 2 then sum_1 else sum_0;
...
//Repeat the loop for next iterations
Proof Obligations:
¬ (ii_4 < 2);
```

Listing 4.18: While Transformation: HLL Code

The variables *ii* and *sum* are translated as *ii_2, sum_2* passing by intermediary values. The variant is a decreasing function which guarantees loop termination. As the maximum number of iterations of the loop is *2-ii*, so the HLL translation process also repeats according to it. The fact that variables are defined in function of while condition and their previous values guarantees the correctness of the translation by value propagation even if all the iterations are not executed.

Operation Call Substitution

In this section, we present the translation of an operation call. Parameter passing is one of the crucial points for the preservation of semantics when translating programs [139]. Contrary to B, HLL does not support functions with non scalar types as it is used in common programming languages.

As we know, a B model is composed of an internal state, several essential properties (the *INVARIANT* clause), and the specification of system evolution (the *OPERATIONS* clause). The general form of an operation call is: $x_{out} \leftarrow op\_name(y_{in})$, calls the operation *op_name* with the effective parameters y_{in} and assigns its result to x_{out}. Several forms of declaring an operation are available, such as an operation without parameters. An operation is triggered if its precondition is satisfied by the caller and should preserve the invariant.

We assume that the called operation (respectively caller operation) belongs to some component $Mch1$ (respectively $Mch2$). An operation call has side-effects implicitly affecting the state of the called machine $Mch1$. As each B machine has it's own data space, therefore the translation of parameter passing can be challenging because it modifies the state of both $Mch1$ and $Mch2$ machines. If the translation process does not follow the precise order of changes in variables, the generated HLL model can be erroneous. When an operation call occurs, we always translate the implementation of this operation [139] (operation body from *IMPLEMENTATION* component).

An operation represents a reusable sequence of statements. Operation calls are inlined in HLL by instantiating the operation parameters and variables in the translation environment, after substituting the formal parameters by the actual ones. The inlining of the operation body is applied by combining the formal text of both operations from a syntactical point of view. To avoid naming conflicts of variables we encapsulate each B operation in a new HLL *namespace* section. This *namespace* has the same name as the original operation appended to an index, counting the different calls of the latter.

In order to preserve the B semantics when transforming to HLL, the translation of B operation call is shown in Table 4.11 and follows these steps:

- The operation body substitution S is transformed into a corresponding block of equations in HLL, S_{HLL}, by applying the transformation function $T_I(S)_{\mathcal{M}}$.
- Extra assignments, $inout_{HLL}$, are introduced to propagate the values of input and output variables (mapping effective parameters to formal ones represented by *in* and *out* variables). The types of the input parameters are synthesised from the precondition of the operation and for the output parameters from the substitution S.
- The preservation of the invariant I (of the called machine $Mch1$) by an operation is specified by an HLL *invariant* predicate over the resulting translation environment $\mathcal{M}_{HLL}$. In HLL, such a predicate is expressed using the *Boolean* as a type of the predicate variable obtained by applying the invariant transformation function T_{Inv}.

The resulting translation environment $\mathcal{M}_{HLL}$ is updated with the result of the operation call and the state changes after the execution of the substitution S.

Example. The call of the operation *minimum* in *Main_i* implementation triggers the translation of this operation into *minimum_0* namespace.

B Construct	HLL Construct
$T_{op}(x_{out} \leftarrow \text{opname}(y_{in})\ \text{S}\ )_{\mathcal{M}} \doteq$	**Let** $T_I(S)_{\mathcal{M}} = (S_{HLL}, \mathcal{M}')$ and $\forall v \in \{x_{out}, y_{in}\}.v_{HLL} = CreateFresh(v, \mathcal{M})$ and $(inout_{HLL}, \mathcal{M}_{HLL}) = in_{HLL} := T_V(y_{in})_{\mathcal{M}'}; x_{out\_HLL} := T_V(out)_{\mathcal{M}'}$; **in** (S_{HLL}; $inout_{HLL}$; $T_{Inv}(I)$, $\mathcal{M}_{HLL}$)

Table 4.11: Rule: Operation Call Transformation

```
IMPLEMENTATION Main_i
IMPORTS Utils
...
OPERATIONS
Main =
mm <-- minimum (xx, yy);
.....
_____________________________
IMPLEMENTATION Utils_i
...
OPERATIONS
.....
rr <-- minimum (aa, bb) =
...
END;
...
```

Listing 4.19: Operation Call Transformation: B Code

```
Namespaces: Main_i{
...
 Namespaces: Main{
  Definitions:
  //Operation call
  mm_1 := Utils_i::minimum_0::rr;
 ....
} }
_____________________________________
Namespaces: Utils_i{
... // First call of "minimum" operation
Namespaces: minimum_0{
...
Definitions:
//Mapping of input parameters
aa_0 := Main_i::Main::xx_1;
bb_0 := Main_i::Main::yy_0;
//Operation body translation
...
//Output
rr := rr_2;
}}
}
```

Listing 4.20: Operation Call Transformation: HLL Code

For better readability, the formal input and output parameters of an operation keep the same name in HLL as in B, without appending their HLL version to the identifier. The variables *aa_0, bb_0* have the role of formal input parameters of the namespace and are mapped to effective parameters *xx_1, yy_0* as a result of calling operation *minimum* in *Main* machine with xx and yy parameters. The output of operation $m$inimum is given as a new assignment as shown in Listing 4.20 for variable *mm_1* and mapped to its corresponding value. After inlining, the list of statements of operation *Main* corresponds to the concatenation of the bodies of all operations called in this main operation.

4.7 Transformation of B Projects

The B language allows us to decompose the development of large applications into sub-components. The structure of a B project consists of a tree of B modules, as shown in Figure 4.2. At the lowest level of a B project, we find: *implementations* and *base machines*. Implementations are the final refinement of a B module. They must be deterministic with executable substitutions and define concrete variables with implementable types. An *implementation* imports modules that define the operations invoked by the implementation. Base machines are the machines that interface with the non-B parts of a project and whose implementations are not further developed in B. The transformation rules for the *implementation* component are explained in more detail below.

In a B project architecture, the starting point is the root machine of the project, the B-main module (*Root* machine in Figure 4.2). This module contains an entry operation that allows all operations of the B project (imported modules) to be invoked indirectly.

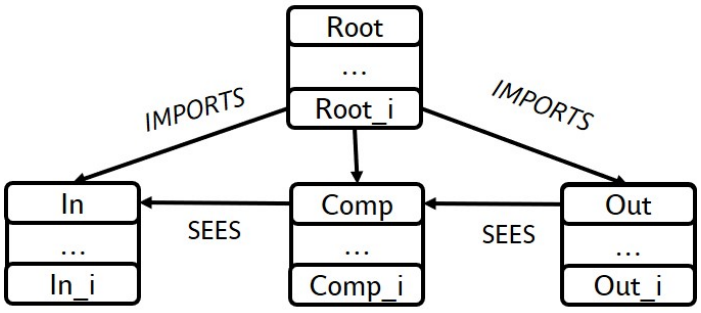

Figure 4.2: B Model Architecture

The transformation process for B components is described in Section 4.3. Section 4.7.1 presents the transformation of some composition primitives. Finally, the transformation of the B-main module is described in detail in Section 4.7.2.

4.7.1 Composition Primitives

In this section, we are mainly concerned with the translation of modules obtained by refinement and the sharing aspects of B composition clauses such as *IMPORTS* and *SEES*. These primitives are described in detail [58]. Note that the proof obligations related to refinement must be discharged prior to translation. Because of the decomposition of properties through refinement, the translation of implementations requires information from

abstract specifications (variables typing, constants definition or signature of an operation). Therefore, we employ an enrichment mechanism [58, 155, 156] to gather information from abstract data clauses such as *SETS, CONSTANTS* and *PROPERTIES* and translate the more concrete parts of the implementation to HLL.

In the case of shared states, the translation must preserve the initial reasoning about variable values. Dependent machines obtained from *IMPORTS* and *SEES* clauses must themselves be translated into HLL *Namespaces*. The transformation function $T_{component}$ applied on $< Component >$ builds a single state by merging the states of dependent B components and delivers one global HLL model. The resulting HLL model contains the translation of an entire B module (a component with its importation chain) by composing the HLL models of each B component.

Let $Component_B$ be a B component. If $Component_B$ is standalone, the translation function is applied on this component and then the domain of the translation environment $\mathcal{M}$ is the set of variables of the component Var_B. Otherwise, $Component_B$ has $SEES$ or $IMPORTS$ primitives on machines $Mch_1, ..., Mch_n$. Before performing the transformation of the initialisation clause from $Component_B$, we perform the transformation of the components Mch_i for the following clauses: $< StructureDef >$, $< DataDef >$, $< StateDef >$ and $< Init >$. T_{struct} transformation function is applied transitively on all imported or seen components Mch_i. The domain of the translation environment is composed of variables from $Component_B$, $Mch_1, ..., Mch_n$, it is updated with the state space of these components. The transformation of an imported or seen component is given in Table 4.12.

B Construct	HLL Construct
$T_{struct}(IMPORTS\ Mch_1)_{\mathcal{M}}\ \doteq$	**Let** $NameHLL(Mch_1, \mathcal{M}) = Mch_{HLL}$ and $T_{component}(< StructureDef >< DataDef >< StateDef >< Init >)_{\mathcal{M}} \doteq (modelHLL, \mathcal{M}')$ **in** (Namespaces: Mch_{HLL} { modelHLL }, $\mathcal{M}$')
$T_{struct}(SEES\ Mch_1)_{\mathcal{M}}\ \doteq$	Component Mch_1 does not exist in the current transformation environment, $\mathcal{M}(Mch_1) = \varnothing$ then: **Let** $NameHLL(Mch_1, \mathcal{M}) = Mch_{HLL}$ and $T_{component}(< StructureDef >< DataDef >< StateDef >< Init >)_{\mathcal{M}} \doteq (modelHLL, \mathcal{M}')$ **in** (Namespaces: Mch_{HLL} { modelHLL }, $\mathcal{M}$')

Table 4.12: Rule: Structuring Transformation

In B project, the clause IMPORTS allows creating new instances of the imported machines. Generally, a machine can be imported at most once in the project. However, it is

possible to instantiate more then once a machine by naming the new instances. Thus the general outline is to associate each B instance of an imported machine with a new instance of the corresponding HLL *namespace*. At HLL model generation, a naming convention is used to avoid name clash, allowing to create a unique *Namespace* for each instance of a B component. In this way we handle the case when a machine is imported several times but one of the instances is not named.

The SEES clause allows one component to access sets, variables, constants and operations found in another component. The use of this clause shall follow the following rule: variables must not be modified by the seeing components. A machine that is accessed by a *SEES* relation must be imported into the project at least once [155]. We translate B models proven to be correct by Atelier B, thus we do not repeat the verification concerning the visibility rules at the HLL level. When translating the SEES clause, a particular attention shall be given to read the data from the appropriate HLL instance of this machine. We are looking at the state of the machine that was created in the project hierarchy using *IMPORTS*. If a transformation for this component does not exist in the translation environment, we initialise it.

Example. The clause IMPORTS generates the translation of the corresponding implementation of *Utils* component. In HLL, a new namespace *Utils_i<0>* is created and represents the translation of the first instance of this machine.

```
IMPORTS Utils
```

Listing 4.21: Structuring Transformation: B Code

```
Namespaces: Utils_i_0{
...
}
```

Listing 4.22: Structuring Transformation: HLL Code

4.7.2 Main Machine

The root machine of the project (B-main module), which contains an entry operation, serves as the starting point in B projects' architecture. The role of this operation is to sequence the order of execution of operations in the project, and it is assumed to be called cyclically by an external process [44].

The translation described in this section uses the most intuitive principle to generate synchronous dataflow code from an imperative program. The basic idea of the transformation specification given below is to translate the body of the B-main operation into HLL

computations that are executed during an instant [157]. Starting from the root module of a B project, all implementations of the transitively imported modules are translated into HLL. The HLL model is obtained by recomposing the called operations of each component, taking into account the order defined by the root component, and specifying which variables become input and output streams. B-project inlining is applied by combining all formal models into a single formal text. In this way, we obtain a global state space that merges all the singular states of the B implementations.

The translation rule for a B project, starting with the main module, is provided in table 4.13 and uses defined translation rules for each B clause. The HLL model is obtained from the B-main machine, which is the root of the importation tree and indirectly invokes all operations of all components by sequentially assembling the operations of each B component in the defined execution order. The transformation of this cyclic system is described as the translation of the substitution of $INITIALISATION$, followed by the transformation of the substitution of the $OPERATIONS$ clause,which is applied repeatedly. The computations from one iteration are converted to HLL values in the current time frame, and these represent the initial value for computations in the next cycle.

Cycle precision is required when transforming B to HLL. Each iteration of the global loop (defined in the root machine) maps the system to a new state. When we observe the behaviour of the loop, we find that the final state of each iteration represents the initial state of the subsequent iteration. In HLL, we observe the evolution of the flows from one point in time to another. Given these two observations, we consider in the transformation that the evolution of flows is equivalent to state changes in B from one iteration to the next one.

We assume that the state of B variables at the end of an iteration represents the value of the HLL outputs in the current cycle and the initial state of the next execution. The cyclic behaviour given by the execution of the B-main operation is modelled in HLL by adding a transition relation that shows the evolution of the flows during the time frames corresponding to the succession of cycles. To represent this behaviour, for all B state variables we introduce additional HLL streams definition such as $v_{HLL} := v_{init}, v_{next}$. It means that v takes its initial value from stream v_{init} and for next instants from stream v_{next}. Stream v_{init} is the initial value of the B variable v given by B $INITIALISATION$ clause. Stream v_{next} represents the state of the B variable at the end of the B main operation. The resulting stream v_{HLL} is used further in system computations specified by B $OPERATIONS$ clause.

At the beginning, the transformation environment is empty, the state and the context of

the main component with its importation tree are added. $\mathcal{M}$' is the resulting translation environment updated consequently. The resulting HLL model represents the transformation of an entire B project.

B Construct	HLL Construct
$T_{component}(IMPLEMENTATION\ Name$ $< StructureDef >< DataDef >< StateDef >$ $< Init >< Ops > END)_{\varnothing}\ \doteq$	**Let** $NameHLL(Name, \varnothing) = Name_{HLL}$ and $T_{dataDef}(< DataDef >)_{\varnothing} \doteq (dataHLL, \mathcal{M})$, $T_{stateDef}(< StateDef >)_{\mathcal{M}} \doteq (stateHLL, \mathcal{M}_1)$, $T_{struct}(< StructureDef >)_{\mathcal{M}_1} \doteq (structHLL, \mathcal{M}_2)$, $T_{init}(< Init >)_{\mathcal{M}_2} \doteq (initHLL, \mathcal{M}_3)$, $\forall v \in dom(\mathcal{M}_3).\{\ v_{HLL} := v_{init}, v_{next};\ where\ v_{init} := \mathcal{M}3(v)$ and $v_{next} := \mathcal{M}4(v);\} \doteq (cycleHLL, \mathcal{M}3[v \rightarrow v_{HLL}])$, $T_{ops}(< Ops >)_{\mathcal{M}_3} \doteq (opsHLL, \mathcal{M}_4)$ **in** (Namespaces: $Name_{HLL}$ $\{dataHLL; stateHLL;$ $initHLL; cycleHLL; opsHLL;\}$ $structHLL$, $\mathcal{M}_4$)

Table 4.13: Rule: Main Machine Transformation

Example. In the following example, we can observe that the IMPLEMENTATION *Main_i_0* is translated to Namespaces clause with the same name in HLL. The clause IMPORTS generates a new namespace *Utils_i_0*, which is the first result of the translation of this machine.

```
IMPLEMENTATION Main_i
...
IMPORTS Utils
```

Listing 4.23: Main Machine Transformation: B Code

```
Namespaces: Main_i_0{
...
}
Namespaces: Utils_i_0{
...
}
```

Listing 4.24: Main Machine Transformation: HLL Code

4.8 Conclusion

This chapter introduces the key concepts for translating from B implementation to HLL dataflow language. The semantic differences between the two languages under study are pointed out and a general translation scheme is proposed. We have described a translation process and a set of translation rules for each construct that requires special attention. We have shown that B implementations can be translated into HLL models through a series of syntactic transformations.

Chapter 5

Certified Model Transformation of B to HLL

Contents

This chapter presents a certified translation from B formal language to HLL. The proposed approach uses HOL as a unified logical framework to describe the formal semantics of B and HLL and to formalise the translation relation of both languages. The developed Isabelle/HOL models are proved in order to guarantee the correctness of our translation process. We present the weak-bisimulation relation to check the correctness of translation steps and the proof process.

5.1 Introduction

The critical industrial application context requires an assessment of the quality of the defined transformation. Since this approach relies on a translator tool, a key feature is semantic preservation and thus the certification of the translator. In this chapter, we address the problem of validating the translator by proving semantic equivalence between the source code and the target code. The certification of the defined transformation process consists in formally ensuring semantic preservation after translation, i.e. we demonstrate that the transformation of the B model into the HLL model preserves the original semantics of B models. More precisely, we present the correctness of the general transformation rules described in Chapter 4 by showing a behavioural equivalence between the B models and their automatically generated HLL models.

5.2 Principles of the Certification Process

Our approach is depicted in Fig. 5.1 and uses equivalence relationship based on a weak bi-simulation relationship to relate B states and HLL flows. It is based on a **deep embedding** of the semantics of both modelling languages into the Isabelle/ HOL framework as a unified formal modelling framework that allows for meta-level logic reasoning.

First, both B and HLL modelling language semantics are modelled in Isabelle/HOL. Note that our semantics formalism takes into account only a subset of constructs from both languages required for the developed models. The certification process addresses the core language constructs from B language. Then, an equivalence relation between these models is formalised. It is based on a bi-simulation relation (upper part of Fig. 5.1). An equivalence theorem is stated and proved (by a structural induction) once for all. This is similar to the ideas presented in [129, 125]. The proof is a structural induction on the constructs of the B modelling language and on the transformation rules. Isabelle/HOL data-types and functions formalise the concepts of both B and HLL. We have formalised the operational semantics associated to the abstract syntax of each syntactic category of both languages as well as the semantic equivalence between them. Below we describe the different steps of this formalised certification process.

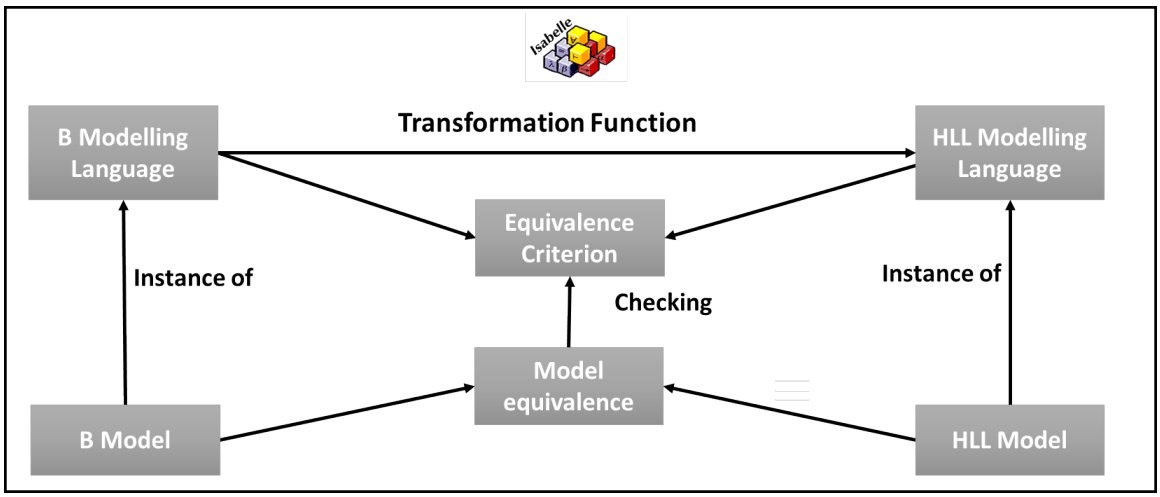

Figure 5.1: A formal certification approach

5.2.1 Basic Isabelle/HOL definitions for the transformation

Both B and HLL semantics are formalised in a big-step semantics style. Syntactic constructs are defined as datatypes. State changes are recorded using interpretation functions associated with each syntactic construct in the current state. Different types of variables for both modelling languages are implemented using `datatype` definition. Isabelle/HOL data-types modelling features and constructs of B and HLL (states, flows, expressions, modelling statements) are defined. Variables names, variable values and an environment function associating variables to their values are introduced in Listing 5.1

```
datatype Tval = Bool | Int
type_synonym varname = "name × Tval"
type_synonym env = "varname ⇒ val"
type_synonym mapping = "b.vname ↦ (hll.vname × hll.vname)"
```

Listing 5.1: Environment function for variables

A state, accessed using an environment function *env*, is defined as a total function that maps variable names to variable values. Primitive types, like integers and booleans are defined as $Tval$. For example, the variable names are encoded as a pair $varname = name \times Tval$: variable name and type. For modelling the variable's values we need to make a distinction between the values of B language and the flow values of HLL. This leads to a different definition of *val* datatype. All the transformation rules are defined as functions mapping B modelling constructs to the corresponding HLL constructs. The relation between B variables

and HLL streams is given by the *mapping* type. It is used below in the transformation functions.

5.3 B Semantics

We define a deep embedding where B models are manipulated as first class objects. We rely on the work of [158]. The semantics of B models is described using a semantic function. This function is defined on the structure of the B models. Each syntactic B construct is interpreted by this function.

5.3.1 B constructs in Isabelle/HOL

The syntax of the B language that we defined is based on Isabelle/HOL syntax. Instead of a specification language based on set theory as it is proposed by Abrial [4], the syntax we defined in Isabelle/HOL notation is based on higher order logic and type theory.

```
datatype aexp =
  Value int
| AVar vname
| Plus aexp aexp
| Times aexp aexp
| Minus aexp aexp
| Uminus aexp
```

Listing 5.2: Arithmetic expressions

```
datatype bexp =
  Value bool      | Neq aexp aexp
| Bvar vname      | Not bexp
| And bexp bexp   | Or bexp bexp
| Leq aexp aexp   | Eq aexp aexp
| Equiv bexp bexp | Lt aexp
    aexp
| Gt aexp aexp    | Greq aexp
    aexp
```

Listing 5.3: Boolean expressions

```
datatype instruction =
  Bl "instruction list"
| SKIP
| Assign vname exp
| If bexp
  instruction instruction
```

Listing 5.4: Statements

Specific data-types for arithmetic expressions *aexp*, boolean expressions *bexp* and B statements *instruction* (a block of instructions in sequence, skip, assignment, and conditional) are defined in Listings 5.2, 5.3, and 5.4 respectively to model B abstract syntax.

5.3.2 B Semantics in Isabelle

The semantics of B constructs is defined using primitive recursive functions encoded in Isabelle/HOL. Listings 5.5 and 5.6 - describe the formalisation of this semantic function.

Expressions

B expressions are interpreted by the function *meaning_exp* ∈ *exp* → *env* → *val*. An expression is evaluated (see Listing 5.5) in the environment *env*. It expresses that a B expression is interpreted in a given state and denotes a value in *val*. Two intermediate meaning functions are introduced for arithmetic (*meaning_a*) and boolean (*meaning_b*) expressions defined on their type constructors.

```
fun meaning_exp :: "exp ⇒ env ⇒ val" where
  "meaning_exp (Bexp ex) σ = B o meaning_b ex σ"
| "meaning_exp (Aexp ex) σ = I o meaning_a ex σ"

fun meaning_a :: "aexp ⇒ env ⇒ int" where
  "meaning_a (Value i) _  = i"
| "meaning_a (AVar vname) σ = (case σ vname of (I v) ⇒ v)"
| "meaning_a (Plus aexp1 aexp2) σ = (meaning_a aexp1 σ + meaning_a aexp2 σ)"
| "meaning_a (Times aexp1 aexp2) σ = (meaning_a aexp1 σ * meaning_a aexp2 σ)"
| "meaning_a (Minus aexp1 aexp2) σ = (meaning_a aexp1 σ − meaning_a aexp2 σ)"
| "meaning_a (Uminus aexp σ = − meaning_a aexp σ

fun meaning_b:: "bexp ⇒ env ⇒ bool" where
  "meaning_b (Value b) _                    = b"
| "meaning_b (Bvar vname) σ          = (case σ vname of (B v) ⇒ v)"
| "meaning_b (Not bexp1) σ           = (¬ meaning_b bexp1 σ)"
| "meaning_b (And bexp1 bexp2) σ     = (meaning_b bexp1 σ ∧ meaning_b bexp2 σ)"
| "meaning_b (Or bexp1 bexp2) σ      = (meaning_b bexp1 σ ∨ meaning_b bexp2 σ)"
| "meaning_b (Equiv bexp1 bexp2) σ   = (meaning_b bexp1 σ ⇔ meaning_b bexp2 σ)"
| "meaning_b (Leq aexp1 aexp2) σ     = (meaning_a aexp1 σ ≤ meaning_a aexp2 σ)"
| "meaning_b (Eq aexp1 aexp2) σ      = (meaning_a aexp1 σ = meaning_a aexp2 σ)"
| "meaning_b (Grthen aexp1 aexp2) σ = (meaning_a aexp1 σ ≥ meaning_a aexp2 σ)"
| "meaning_b (Neq aexp1 aexp2) σ   = (meaning_a aexp1 σ ≠ meaning_a aexp2 σ)"
```

Listing 5.5: Semantics of B expressions

Literal values are directly interpreted by their corresponding Isabelle/HOL values. The semantics of binary expressions (+, -, *, =, <, >) is defined from the interpretation of their operands with respect to the literal values given by the meaning of expressions in both sides of the symbols.

Instructions

State changes are formalised by the state transition function *meaning_instruction* ∈ *instruction* → *env* → *env*. It produces the next state after execution of a given B statement.

It updates the environment *env* with the effect of the interpreted instruction. Listing 5.6 provides the definition of the semantic function *meaning_instruction* for assignment, sequence, conditional and skip instructions. An identity function denotes an unmodified state. It is also used to denote an empty block of instructions useful for conditionals (using Skip).

```
fun meaning_instruction :: "instruction ⇒ env ⇒ env" where
  "meaning_instruction (SKIP)  σ = σ"
| "meaning_instruction (Bl list) σ =
    (case list of [] ⇒ σ
                | e#l ⇒ meaning_instruction (Bl l) (meaning_instruction e σ))"
| "meaning_instruction (Assign (vn, Tval.Bool) (Bexp exp)) σ =
    σ ((vn, Tval.Bool) := B (meaning_b exp σ))"
| "meaning_instruction (Assign (vn, Tval.Int) (Aexp exp)) σ =
    σ ((vn, Tval.Int) := I (meaning_a exp σ)"
| "meaning_instruction (If c b1 b2) σ =
(if meaning_b c σ then meaning_instruction b1 σ else meaning_instruction b2 σ)"
```

Listing 5.6: Semantics of B statements

In Listing 5.6, sequential composition of instructions is inductively defined. The interpretation of $e\#l$ is given by the interpretation of instruction e in the given state followed by the interpretation of the rest of the list l. An assignment produces the state where a variable is updated. A conditional statement is denoted by condition evaluation in the current state followed by the interpretation of either *if* or *else* branch depending on the interpretation of the condition.

The case of loops

As mentioned in Section 4.6.3 the transformation tool translates a loop as the recursive function $b\_while\_to\_if$ with conditional (see Listing 5.7). In this Listing, we observe that the function is called a *nb* number of times corresponding to the original B *VARIANT* value, used in B to ensure termination of the loops. Therefore, it can be unfolded as a sequence of *if then else* statements in a block *Bl*. In other words, each loop can be seen as a sequence of unfolded *if then else* statements. The built-in fixpoint operator available in Isabelle/HOL is used to define the $b\_while\_to\_if$ recursive function.

```
fun b_while_to_if::"nat ⇒ bexp ⇒ instruction ⇒ instruction" where
"b_while_to_if 0 _ _ = SKIP" |
"b_while_to_if (Suc nb) c i = Bl [If c i SKIP, (b_while_to_if nb c i)]"
```

Listing 5.7: A recursive function encoding while loops

To translate the `while` loop of B language, we use the formalisation of `if` statement iteratively. We provide a theorem to show an equivalence between the defined function *b_while_to_if* and the predefined while semantics from Isabelle/HOL *while combinator* theory [5]. The theorem guarantees the correctness of the translation by satisfying the condition, which states that the number of unfoldings should be greater than the maximum value of the given variant. We provide a lemma to show the equivalence between the meaning of while unfold as we defined it and the while defined with lfp in *While_Combinator* theory [5].

At this stage, all basic B constructs defined at the *operation* level in B are embedded in an Isabelle/HOL definition. All other constructs can be rewritten using these basic constructs. For example, the operation call statement, for example is not explicitly encoded in Isabelle/HOL because it can be viewed as a sequence of statements.

5.4 HLL Semantics

HLL is a declarative and synchronous language in SSA (Single State Assignment) form. Several formal models of synchronous languages with single state assignment [159, 160] have been proposed. We rely on these SSA based semantics to define HLL semantics. A HLL model can be defined as a collection of order independent flow (stream) assignments.

In our work, each stream is defined as a sequence of values. In Listing 5.8, each sequence is defined as a function mapping a natural number to a polymorphic datatype, $'a$ [112]. As we know that in the HLL language, each variable has a unique value and it has a unique definition while in the B language variables can be modified iteratively. HLL variable names are defined as $(name \times Tval) \times nat$. Each variable is uniquely identified. Uniqueness indexing of variables ensures that once translated, no B model identifier is assigned twice in the obtained HLL model (i.e. single assignment property).

```
datatype Tval = Bool | Int
type_synonym 'a stream = "nat ⇒ 'a"
datatype val = B "bool stream" | I "int stream"
type_synonym varname = "name × Tval"
type_synonym vname = "varname ×  nat"
```

Listing 5.8: Data type for HLL flows (streams)

As for B, where we defined the values and state variables as datatypes, we proceed with HLL. Before describing the semantics of the dataflow language HLL in Isabelle/HOL, we formalise the notion of flow. HLL flows (streams) are defined as functions mapping naturals on a polymorphic data-type in Listing 5.8. HLL variables are defined as $vname = (name \times Tval) \times nat$. Each variable is associated to a unique identifier defined by a natural number. *R*emind that the mapping function defined in Listing 5.1 maps B variables and HLL variables.

5.4.1 HLL constructs in Isabelle/HOL

Similarly to B, specific data-types for arithmetic expressions *aexp*, boolean expressions *bexp* and statements *instruction* are defined.

Since, the HLL conditional is an expression, a particular attention is paid to the flows resulting from conditional expressions. It is processed as an expression construct in Listing 5.11. HLL instructions are assignment blocks. The Isabelle/HOL definitions of these constructs are given in Listings 5.9, 5.10, and 5.11.

```
datatype aexp =

  Value "int stream"
| AVar vname
| Plus aexp aexp
| Times aexp aexp
| Minus aexp aexp
| Uminus aexp
```

Listing 5.9: Arithmetic Expression

```
datatype bexp =
  Value "bool stream"
| Neq aexp aexp
| Bvar vname     | Not bexp
| And bexp bexp | Or bexp bexp
| Leq aexp aexp | Eq aexp aexp
| Equiv bexp bexp |Lt aexp aexp
| Gt aexp aexp | Greq aexp aexp
```

Listing 5.10: Boolean Expression

```
datatype exp =
  Bexp bexp
| Aexp aexp
| If bexp exp exp
datatype instruction =
  Bl "instruction list"
| Assign vname exp
| Assign' vname exp exp
```

Listing 5.11: Expression and Statements

Stream composition in Isabelle/HOL

In HLL, we reason on a sequence of input using combinatorial logic. The output is the same, even if the input sequence is permuted. Like dataflow languages, the stream assignment is

order independent in HLL. An HLL model runs in an environment where the variables are defined as a bounded stream of input data. The semantics of the HLL language imposes that the updating of the flows is performed in a synchronous manner, i.e. the flows are modified simultaneously and there is no side effect. To handle this synchronous updating of flows the *stream_comp* (see Listing 5.12) has been introduced. It composes different stream values. This function is used by the semantic function interpreting the HLL statements (see Listing 5.16). The combinatorial judgements are used for analysing the values at a particular instant, for example, the stream composition of $v1$ and $v2$ returns a new stream, in which the first instant is the value of $v1$ and the next instant is the value of $v2$.

```
fun
stream_comp::"val ⇒ val ⇒ val" where
  "stream_comp (B v1) (B v2) = B(λi. if i=0 then v1 0 else v2 (i−1))"
| "stream_comp (I v1) (I v2) = I(λi. if i=0 then v1 0 else v2 (i−1))"
```

Listing 5.12: Flow composition

5.4.2 HLL Semantics in Isabelle/HOL

Like for B, the HLL semantics is given by semantic functions defined structurally on the corresponding syntactic constructs. The semantic rules for evaluating expressions are defined by the interpretation function $meaning\_exp \in exp \to env \to val$. As for B, it is defined for arithmetic expressions with *meaning_a* and boolean expressions with *meaning_b* (see Listing 5.13, 5.14, 5.15). The semantics of the *if* expression in a state σ produces stream values resulting from the recursive evaluation of branch expression depending on the given condition.

```
fun meaning_exp :: "exp ⇒ env ⇒ val" where
  "meaning_exp (Bexp ex) σ = B (meaning_b ex σ)"
| "meaning_exp (Aexp ex) σ = I (meaning_a ex σ)"
| "meaning_exp (If c b1 b2) σ = (let (val1, val2) =
   ((meaning_exp b1 σ), (meaning_exp b2 σ)) in (case (val1, val2) of
     ((I b1), (I b2)) ⇒ I (λi.(if meaning_b c σ i then b1 i else b2 i))
   | ((B b1), (B b2)) ⇒ B (λi.(if meaning_b c σ i then b1 i else b2 i))))"
```

Listing 5.13: Semantics of HLL Expressions

```
fun meaning_a :: "aexp ⇒ env ⇒ int stream" where
 "meaning_a (Value i) _                    = i"
|"meaning_a (AVar vname) σ  = (λi. (case σ vname of (I v) ⇒ v i))"
|"meaning_a (Plus aexp1 aexp2) σ = (λi.meaning_a aexp1 σ i + meaning_a aexp2 σ i)"
```

```
|"meaning_a (Times aexp1 aexp2) σ=(λi. meaning_a aexp1 σ i * meaning_a aexp2 σ i)"
|"meaning_a (Minus aexp1 aexp2) σ=(λi. meaning_a aexp1 σ i − meaning_a aexp2 σ i)"
|"meaning_a (Uminus aexp) σ=(λi. − meaning_a aexp σ i)"
```

Listing 5.14: Semantics of HLL Arithmetical Expressions

```
fun meaning_b :: "bexp ⇒ env ⇒ bool stream" where
  "meaning_b (Value b) _ = b"
| "meaning_b (Bvar vname) σ = (case σ vname of (B v) ⇒ v)"
| "meaning_b (Not bexp1) σ = (λi.¬ meaning_b bexp1 σ i)"
| "meaning_b (And bexp1 bexp2) σ = (λi.meaning_b bexp1 σ i ∧ meaning_b bexp2 σ i)"
| "meaning_b (Or bexp1 bexp2) σ = (λi.meaning_b bexp1 σ i ∨ meaning_b bexp2 σ i)"
| "meaning_b (Equiv bexp1 bexp2) σ =(λi.meaning_b bexp1 σ i ⇔meaning_b bexp2 σ i)"
| "meaning_b (Leq aexp1 aexp2) σ = (λi.meaning_a aexp1 σ i ≤ meaning_a aexp2 σ i)"
| "meaning_b (Greq aexp1 aexp2) σ =(λi.meaning_a aexp1 σ i ≥ meaning_a aexp2 σ i)"
| "meaning_b (Eq aexp1 aexp2) σ = (λi.meaning_a aexp1 σ i = meaning_a aexp2 σ i)"
| "meaning_b (Gt aexp1 aexp2) σ = (λi.meaning_a aexp1 σ i > meaning_a aexp2 σi)"
| "meaning_b (Lt aexp1 aexp2) σ = (λi.meaning_a aexp1 σ i < meaning_a aexp2 σ i)"
| "meaning_b (Neq aexp1 aexp2) σ = (λi.meaning_a aexp1 σ i ≠ meaning_a aexp2 σ i)"
```

Listing 5.15: Semantics of HLL Boolean Expressions

For statement evaluation, we introduce the interpretation function $meaning\_instruction[\![\_]\!] \in instruction \rightarrow env \rightarrow env$ defined in Listing 5.16.

```
fun meaning_instruction :: "instruction ⇒ env ⇒ env" where
  "meaning_instruction (Bl list) σ =
      (case list of [] ⇒ σ
                  | e#l ⇒ meaning_instruction (Bl l) (meaning_instruction e σ))"
| "meaning_instruction (Assign vn exp) σ = σ (vn := meaning_exp exp σ)"
| "meaning_instruction (Assign' vn exp1 exp2) σ = (let v1 = meaning_exp exp1 σ in
   let v2 = meaning_exp exp2 σ in σ (vn := stream_comp v1 v2))"
```

Listing 5.16: Semantics of HLL statements

The defined function $meaning\_instruction \in instruction \rightarrow env \rightarrow env$ updates the environment of flows according to the semantics of the HLL statement (See Listing 5.16). Two constructors are defined for assignment. The first one updates the state of a variable with the stream value obtained from the expression evaluation, and the second one updates the state variables with a composed stream (*stream_comp* function) resulting from the expression evaluation of $v1$ and $v2$.

5.5 Correctness of the Transformation

Once the B and HLL semantics are encoded in Isabelle/HOL, the specification of the B2HLL translation shall be defined in Isabelle/HOL. Following that, semantic preservation is defined by defining an equivalence relationship.

5.5.1 The Transformation Function

The transformation function from B to HLL has been defined on the syntactic constructs identified for both B and HLL in Listings 5.20, 5.19, 5.17, 5.18.

- First, we address the mapping of B state variables to HLL flows (streams) which require a specific process. Each B variable identifier is associated to a unique type compatible pair of read and write HLL identifiers using the $Mapping = Bvname \nrightarrow (Hllvname \times Hllvname)$ function of Listing 5.1. The defined $Mapping$ for variables is exploited to retrieve the HLL stream corresponding to each B variable. The first identifier is used for expression evaluation and the second one for variables mapping updates. Most of the time, both elements can have same values but they can be different during the *if* condition transformation.

- B Expressions are transformed using $T\_exp \in Bexp \rightarrow Mapping \rightarrow HLLexp$. The transformation is straightforward defined with the help of a set of functions. $T\_exp[\![\_]\!] \in b.exp \rightarrow mapping \rightarrow hll.exp$ is a translation function to transform the B expressions into HLL expressions. T_aexp and T_bexp are defined to realise the syntactic transformation of arithmetic and boolean expressions, respectively. In general, the value stream of HLL is associated to a unique B value. The defined identifiers of B become HLL identifiers. Note that these HLL identifiers are read only variables. The notion of *type-safety* is taken into account during the transformation and the typing notion is embedded directly when the HLL variables are generated to preserve the typing properties.

```
fun T_aexp :: "b.aexp ⇒ mapping ⇒ hll.aexp" where
  "T_aexp ((b.aexp.Value i)) _ = (hll.aexp.Value(λi _. i))"
| "T_aexp ((b.AVar vname )) m  = (hll.AVar (fst (the (m vname))))"
| "T_aexp ((b.Plus aexp1 aexp2)) m = hll.Plus (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_aexp ((b.Times aexp1 aexp2)) m = hll.Times (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_aexp ((b.Minus aexp1 aexp2)) m = hll.Minus (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_aexp ((b.Uminus aexp)) m = hll.Uminus (T_aexp aexp m)"
```

Listing 5.17: Transformation of B Arithmetical Expressions

```
fun T_bexp ::"b.bexp ⇒ mapping ⇒ hll.bexp" where
  "T_bexp ((b.Value b)) _             = (hll.Value(λi _. b))"
| "T_bexp ((b.Bvar vname)) m          = (hll.Bvar (fst (the (m vname))))"
| "T_bexp ((b.Not bexp)) m            = hll.Not (T_bexp  bexp m)"
| "T_bexp ((b.And bexp1 bexp2)) m     = hll.And (T_bexp bexp1 m) (T_bexp bexp2 m)"
| "T_bexp ((b.Or bexp1 bexp2)) m      = hll.Or (T_bexp bexp1 m) (T_bexp bexp2 m)"
| "T_bexp ((b.Equiv bexp1 bexp2)) m = hll.Equiv (T_bexp bexp1 m) (T_bexp bexp2 m)"
```

```
| "T_bexp ((b.Leq aexp1 aexp2)) m    = hll.Leq (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_bexp ((b.Eq aexp1 aexp2)) m     = hll.Eq (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_bexp ((b.Grthan aexp1 aexp2)) m = hll.Gt (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_bexp ((b.Neq aexp1 aexp2)) m    = hll.Neq (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_bexp ((b.Greq aexp1 aexp2)) m   = hll.Greq (T_aexp aexp1 m) (T_aexp aexp2 m)"
| "T_bexp ((b.Lessthan aexp1 aexp2)) m = hll.Lt (T_aexp aexp1 m) (T_aexp aexp2 m)"
```

Listing 5.18: Transformation of B Boolean Expressions

```
fun T_exp ::"b.exp ⇒ mapping ⇒ hll.exp" where
  "T_exp (b.Bexp exp) =  hll.Bexp o (T_bexp exp)"
| "T_exp (b.Aexp exp) =  hll.Aexp o (T_aexp exp)"
```

Listing 5.19: Transformation of B Expressions

Regarding B statements, the transformation function of Listing 5.20 $Transformation \in B.instruction \rightarrow Mapping \rightarrow (HLL.instruction \times Mapping)$ produces HLL code from B instructions and updates the mapping accordingly.

- For the assignment statement, a fresh HLL identifier is created (using *CreateFreshHLLVariable* function). This new identifier is defined with an incremented index value, if it already exists in the mapping or adds to the mapping *Mapping* a new binding stream for the B variable if not.

- The transformation of sequence is straightforward and applies to a block (a list in the model) of instructions *a#list* in a initial mapping m. The transformation uses an inductive definition based on the structure of the list. The updated mapping is a parameter of this function.

- Processing the conditional *if then else* statement is more complex. The translation of `if` constructs is challenging because in B language this construction is a statement and in HLL it is an expression. The translation is performed in different steps, the condition and the statements of each branch are transformed. In the `if` expression of B, any variable can be modified in both branches, which may raise a conflict in the bindings and it can cause information loss. To resolve this conflict, we introduce a mapping composition that creates a new mapping used to transform the *else* branch. This mapping has as its read version of variables the one that originally exists in m, and as its write version the variables from $m1$. Since B variables values may be modified in each branch, we use the associated read and write streams associated to each variable. The write stream is the only modified stream in case of assignment and

the read stream is used for expression evaluation. At the end, a list of assignments for the modified variables is produced for each branch of the conditional.

- Since B loop statements have been turned to recursive *if* statements they fall in the previously defined processing of conditional statements.

```
fun Transformation ::"b.instruction ⇒ mapping ⇒ (hll.instruction × mapping)" where
  "Transformation (b.Bl []) m            = (hll.Bl [], m)"
| "Transformation (b.Bl (a#list)) m      = (comp (Transformation a m) (Transformation (b.
    Bl list)))"
| "Transformation  b.SKIP m              = (hll.Bl [], m)"
| "Transformation (b.Assign vname exp) m =
    (let v = (createFreshHLLVariable vname m) in
         (hll.Assign v (T_exp exp m), m(vname↦ (v, v))))"
| "Transformation (b.If bexp instruction1 instruction2) m =
  (let
       (* c' ⟹ Condition transformation *)
        c' = (T_exp( b.Bexp (bexp)) m) ;
       (* c1 ⟹ IF block transformation and m1 ⟹ Resulting mapping *)
       (c1,m1) = Transformation instruction1 m;
       (* c2 ⟹ Else block transformation and m2 ⟹ Resulting mapping *)
       (c2,m2) = Transformation instruction2 (m ⊗ m1);
       (* vars ⟹ Modified vars in one of IF branches *)
       vars = {v. v : (dom m) ∧ ((m v ≠ m1 v) ∨ (m v ≠ m2 v))} ;
       inst = λi v. (case (snd v) of Tval.Bool ⇒ (hll.Bexp o hll.Bvar)
                                   | Tval.Int  ⇒  hll.Aexp o hll.AVar);
        (* st ⟹ Final state after IF *)
       st = Finite_Set.fold (T_if_step_st) m2 vars;
        (* List of assigns for modified vars *)
       assigns = Finite_Set.fold (T_if_step_i m1 m2 inst c' st) {} vars
   in (Bl ([c1, c2]@(set_to_list assigns)), st))"
```

Listing 5.20: B to HLL Transformation Function in Isabelle/HOL

5.5.2 The equivalence relationship

At this stage, it is possible to define an equivalence relationship on states and flows. This relation, namely ≅ is defined on state variables using an observational relation [161] between states of a B model and corresponding HLL flows obtained after transformation. Listing 5.21 shows this relation in the case of integer and boolean types.

```
definition meaning_equiv:: "b.env ⇒ mapping ⇒ hll.env ⇒ bool" ("_ ≅ _ _") where
"b ≅m h ≡ ∀ v ∈ (dom m). case v of
            (vname, Tval.Bool) ⇒ ((b v) ≙bool ((h o (fst o (the o m))) v))
          | (vname, Tval.Int)  ⇒ ((b v) ≙int ((h o (fst o (the o m))) v))"
```

Listing 5.21: State equivalence Relation (bi-simulation)

The equivalence relation `meaning_equiv` between the B model and HLL model is given through variables mapping. The variables mapping allows to have same values for all variables of B model and the generated data-flow stream of HLL. We prove that a variable and a data-flow stream have the same value by ensuring that each stream value of the HLL model is equal to the value of the corresponding variable of the B model. This definition defines the basic property to check semantic preservation. It connects states to flows. A bi-simulation relationship is defined to relate B models to HLL models.

5.5.3 Asserting correctness of transformation

In this section, we describe the main equivalence theorem and we show a strategy to prove that the transformation process maintains the semantic equivalence between the B and HLL representations by a structural induction.

```
theorem Equivalence:
1. fixes codeB::"b.instruction" and σB::"b.env"
2. and codeHLL::"hll.instruction" and σHLL::"hll.env"
3. and n m::mapping
4. assumes * : "(codeHLL, m) = Transformation codeB n"
5. and # : "σB ≅n σHLL"
6. and $ : "finite (dom n)"
7. and @ : "well_defined codeB n"
8. and ♣: "well_defined_mapping n"
9. and ~ : "well_defined_state σHLL"
 shows
10. "(b.meaning_instruction codeB σB) ≅ m
     (hll.meaning_instruction codeHLL σHLL)"
```

Listing 5.22: Main Equivalence Theorem–Establishing bi-Simulation

All the ingredients to write the equivalence theorem are available. Listing 5.22 describes the global equivalence theorem defining the semantic preservation property. Informally, this theorem states the bi-simulation relation between two state transitions systems. Let

- $CodeB$ and $codeHLL$ (lines 1 and 2) be a B code and its corresponding transformed HLL code (Line 4), and
- σ_B and σ_{HLL} be two states for B and HLL respectively (lines 1 and 2)
- such that σ_B and σ_{HLL} are equivalent by the $\cong$ (of section 5.5.2) relation (line 5)

then,

- the $\cong$ equivalence relation holds on the semantic interpretation of the *codeB* and *codeHLL* in the states σ_B and σ_{HLL} (Line 10).

Note that this theorem uses additional assumptions as follows.

- The initial mapping domain must be finite (Line 6).
- The B model must be well defined in the initial mapping i.e. type checking and all the variables are considered in the mapping (Line 7).
- To ensure the variables traceability in the mapping, the HLL variables should be associated to the B variables that have similar names (Line 8).
- To ensure the type safety of the produced HLL model, all the variables must be well-defined and the initial values should conform the required type (Line 9).

A B code is considered *well_defined* with respect to a mapping if the mapping contains all of the variables used in the B code. When doing the conversion from B code to HLL, we verify that we assign a value in conformance with the variable type. In B implementations, the variables are declared and typed in *CONCRETE_VARIABLES* and respectively *INVARIANT* clause. This is considered to be embedded somewhere else, and when a transformation is realised all of the program variables are already in the mapping. A mapping is *well_defined* if B identifiers are associated to the same identifiers in HLL. In this way the type correctness is preserved. A HLL state is *well_defined* if all state variables are associated to values that conform with their defined types. This is related to type safety of produced HLL code. The type safety theorem says that if the evaluation of HLL code obtained from a transformation starts in a well defined state, the resulting state is also well defined.

These assumptions are ensured thanks to defined lemmas. Indeed, different lemmas were proved to ensure that the transformation is achieved in a well defined mapping between states and flows.

Proving semantic preservation

The proof of the equivalence theorem of Listing 5.22 is performed using the Isabelle/HOL theorem prover. The construction of proofs is mechanical. The powerful tactics available in this prover allowed us to complete the whole proof of this theorem. Most of the proofs are

interactive (semi-automatic), they are completed through user interaction with the theorem prover of Isabelle/HOL.

The classical implementation of Isabelle tactics consists of rewriting operations on data structure for representing proof goals. To facilitate proofs about typing, definition and theorems related to formalisation for both modelling languages, B and HLL, we use the built-in theorem prover of Isabelle/HOL, which can discharge any kinds of conjecture by collecting all the relevant theorems. During the checking process, the tactic application proves the specified properties that must be met. In case of failure of the proof, it means that the proof script is insufficient or there is an error in the model that does not satisfy the semantic condition.

A structural induction with case based reasoning (for each syntactic construct) has been set up. These cases have been decomposed into several lemmas which have been used for the proof of the main equivalence theorem. The identity and sequence construct proofs are quite straightforward. Establishing $well\_defined\_mapping$, $well\_defined\_code$, $well\_defined$ mostly requires to use the defined lemmas to hold for transformations. The proof for assignment is a proof by case on the variables types. Here we must show that the value of an expression exp in the state σ_B is equivalent to the HLL value σ_{HLL} obtained from the transformation of this expression.

However, some complex transformation rules may require more elaborated proofs. For example, the semantic preservation proof for if conditional statement required more than 300 lines of proof script and uses 25 intermediate lemmas to complete the proof.

The proof was particularly complex because of the optimised translation of the branching composition statements, which produces assignments exclusively for the modified variables. To simplify the proof, we introduced a slightly different but equivalent description of the `if` translation, which we call $Transformation\_if'$. This translation makes assignments for all variables, modified or not, which is not optimised but drastically simplifies the proof. It is proved that the new translation is semantically equivalent to the optimised translation. It is also proved that the correctness of the translation relation is preserved for `if` constructs.

The correctness proof for while constructs is given by the following theorem: for a B-code obtained by unfolding the while construct nb times, if there exists a k less than nb for which the while condition becomes false, then the translation of the while construct is correct. The proof of this theorem is almost straightforward by applying the *Equivalence* theorem.

In summary, the proof of the correctness of the transformation from B to HLL represents

more than 5000 lines of proof scripts for discharging the proof obligations related to the transformation (i.e. mapping, existence of the variable in the mapping, type checking etc.) associated to the equivalence proofs (i.e. definition of semantic equivalence, variables updates, variables traceability etc.). In this work, our overall effort is 5-6 months (as a novice user) for completing the Isabelle/HOL formalisation and proofs.

Validating semantic preservation

There are several ways to consider the verification of the proposed transformation rules by the B2HLL tool: human inspection of the generated tool, different testing techniques, run-time verification by simulation and animation, and correctness proof. In our approach we have combined human inspection with the testing and proof approach to verify the correctness of the B2HLL tool. B2HLL provides additional functions to facilitate the human inspection of HLL generated models through traceability of source and target models. The HLL models are annotated with information referring to the original B code. A testing strategy, combined with the animation capabilities offered by the Isabelle/HOL models animator, is implemented to check and validate the generated HLL models from the B models.

We perform model execution combined with formal proof to link the B2HLL tool with the certified translation rules as defined in Isabelle/HOL. A typical scenario for such an animation in Isabelle/HOL is

Step 1. Translate the B models to HLL models using the B2HLL tool

Step 2. Set the B state variables to initial values

Step 3. Run the Isabelle/HOL encoded B models using the Isabelle/HOL animator

Step 4. Run the HLL models, obtained using the B2HLL tool, with the same variables values of Step 2 assigned to HLL streams

Step 5. Run the animation of the Isabelle/HOL HLL models, obtained using the Isabel/HOL translation function, with the same variables values of Step 2 assigned to HLL streams. It is the responsibility of the designer to discharge the verification condition in Isabelle/HOL using different tactics and to prove that the equivalence theorem holds.

Step 6. Inspect and analyse the final values of the B state variables and the obtained HLL streams.

The translator has been tested on several models. We first considered many representative examples of B implementation machines such as a room Booking system, Pixel moving on a screen. Then, complex machines with large state spaces, and many arithmetic and logic expressions issued from the railway domain as test cases. We validated, using the previously described validation scenario, B projects supplied by RATP with more then 100 B implementations and 50 basic machines (imported or seen machines). The current version of the B2HLL tool handles the implementation level of B models including imperative programming constructs corresponding to 5000 lines of B code. This approach allowed us to realise a first proven tool before its transfer. Indeed, industrialisation of B2HLL is ongoing at RATP targeting the translation of the entire B language based on our results. In order to automatise the entire verification chain, automatic export tools from B and HLL languages to Isabelle/HOL are under development. We have also provided proof of the correctness of the transformation rules implemented by the B2HLL tool.

To conclude this section, we mention that the interest of our approach is twofold. On the one hand, it offers an independent verification approach to B models that does not rely on B tools and, on the other hand, it supports an integrated verification framework with HLL in a single modelling language. In addition, the certification process summarised in this section asserts the correctness of the transformation in terms of semantic preservation. It may be used to check the correctness of specific transformations using the animation capabilities.

5.6 Conclusion

In this chapter we have presented a complete formal verification process of the transformation rules from B to HLL. The semantics of the two modelling languages B and HLL are expressed in Isabelle/ HOL and equivalence between the source and target modelling languages is checked. The correctness of the translation rules is proved in Isabelle/ HOL theorem prover. An equivalence proof between B and HLL semantics based on a bi-simulation relation has been established. The formalisation and related proofs presented in this work can be easily extended to other transformations from state-based languages to HLL.

Chapter 6

Transformation at work

Contents

To automate the approach, we proposed a translation from B to HLL language, implemented as the B2HLL tool. In this chapter, we present an architecture of the tool and give implementation details. Using this tool, we can apply a verification process to already developed models and check system safety properties on these models. The application of the proposed approach in an industrial context and how this tool articulates with the PERF toolkit is presented. Finally, we apply our tool to a case study and discuss the last step of the validation strategy of the B2HLL tool, the model animation.

6.1 B2HLL Tool

This section presents a general architecture of the prototype of the B2HLL tool, a translator for B formal models into HLL code, including technical challenges related to tool development. This in order to prove, with the help of the PERF toolkit, properties not present in a B model, which are the subject of the RATP evaluation of critical software. This tool demonstrates the feasibility of converting B to HLL.

The B2HLL tool is developed in C++ language according to the previously defined and validated translation rules. The decision to use the C++ language is due to the fact that commercial tools and translators for the B method have been developed in the C language. The tool considers a subset of the B language. It behaves like a compiler that applies a transformation rule when matching the text input to obtain HLL models, as shown in Figure 6.1.

Figure 6.1: The PERF verification workflow

The tool allows to:

- transform a B model at the implementation level and produce a representative HLL model
- use additional information present in B models to optimise the translation:
  - B elements useful for proving properties (preconditions, post-conditions, invariants, constraints, loop invariants, etc.) to effectively insert them into the HLL model
  - the architecture of the B model and the associated visibility rules

The tool architecture, as illustrated in Figure 6.2, is composed of three main phases:

- **Parsing.** The phase consists of a syntactic and semantic analysis on B input files to build a syntax tree. The parser translates one or more B model files (the source code of the model to be transformed) into a data structure that we can then manipulate,

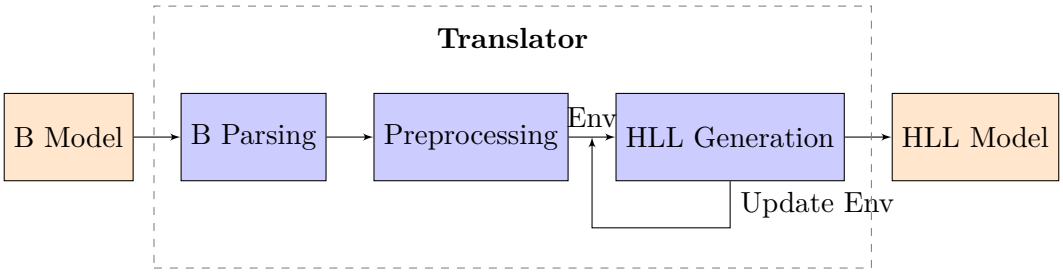

Figure 6.2: B2HLL Translation Workflow

called Abstract Syntax Tree (AST). During the syntax and semantics analysis, several checks are performed on the files received as input to decide whether they are valid with respect to the B language grammar. For this purpose, we have chosen a parser suitable for programming various transformations on trees/terms. More details about this phase can be found in 6.1.1.

- **Preprocessing.** After obtaining the data structure of the B-component, we annotate the previously generated AST with various information needed to translate it into HLL, such as the links between some syntactic objects, the type inference of the variables, and the definition of the visibility domain of the variables. We describe this phase in detail in 6.1.2.

- **HLL Generation.** In this step, the final HLL model for a B project is generated. First, we create HLL templates for each B component. Using appropriate rules, we perform the conversion of AST nodes and generate the corresponding HLL code. At this point of the process, a purely static translation of the constructs is performed, without considering dynamic information such as the correspondence between B and HLL variables and the execution order of the B statements. This step allows us to generate an HLL translation template for each B model. Further, the translation starts with the main machine of the B project to be translated and follows the sequence of operation calls to generate an HLL model that reflects the state changes of the original B model. The previously generated templates are instantiated with dynamic information such as variable indices and effective operation parameters. For more details, see Section 6.1.1.

The B2HLL tool requires as input a verified B model, more precisely, all generated POs must be proven to be correct. This is an important requirement of the tool, since generating code from an incorrect model can lead to undesirable behaviour. Furthermore,

the model must contain finite loops, be type-checked, and satisfy the well-definedness rules. This checking can be done using the AtelierB tool. Compliance with this requirement is not checked by the B2HLL tool. The translation shall give a conservative HLL model, meaning that any falsifiable assertion of the original B model shall be falsifiable in the HLL model.

The input of the B2HLL tool are B implementations [4, 48]. The B2HLL tool considers the following B constructions: variables (integers, booleans, arrays, enumerated sets), boolean and arithmetic expressions, substitutions (assignment, sequence, conditional, loop, operation call, local variable definition) and B composition clauses.

In practice, in a B project, there are abstract machines without associated refinement or implementation, to encapsulate specific types or statements which are not supported by B0. These machines are imported and used in B implementations. For our work, the translation of these machines is mandatory. Note that, the B2HLL tool does not support yet the transformation of B abstract concepts.

The tool has been tested on several case studies. First of all, examples of B projects known in the literature have been translated with the B2HLL tool (reservation system, pixel movement, etc.). This allowed us to fully translate B projects into HLL and later check properties on HLL models from this translation. HLL code generation was also tested on complex projects. The tool produced a valid HLL model, but comments were generated for the translation of B constructs, which are not implemented in the current version of the tool.

The objective of this prototype, developed with the help of two interns [162, 163], is not to propose a final solution but to serve to demonstrate the feasibility of the proposed solution. In order to apply the PERF method to validate critical software developed in B, it is necessary to develop an industrial version of the B2HLL tool. This tool will have to translate all the constructs of the B language, ensure the traceability of the variables between the B code and the obtained HLL code and guarantee the equivalence between the translation rules implemented in the tool and their formal definition.

The following sections describe the different steps of the translation process, from the moment the user selects a folder containing the B project to be translated and a root B machine, to the generation of the HLL models.

6.1.1 Parsing

In terms of tools and proof environments around the B method, a move towards open source for use in academia has been envisaged. Several tools have been developed that include

parsers for the B language. Table 6.1 summarises the existing tools that include the B parsing functionality.

Tool	Dev.Language
ABTools	Java/ANTLR
JBTools	Java
BRILLANT	Objective Caml
BComp	C

Table 6.1: B Parsers

The BRILLANT platform [156], an open-source environment that allows to manipulate B models from specification to code generation, includes a B parser developed in Objective Caml. The parser outputs an XML representation for B formal specifications. The generated XML file is further used as input for other tools.

ABTools [164] is a development environment for the B method, an open source tool created to provide the possibility to study and test extensions of the B language. The tool was developed with ANother Tool for Language Recognition (ANTLR)[1] compiler generator. It is made up of several components such as lexical and syntax analyser, a decompiler, a type-checker and a PO generator. Based on their defined B grammar, the parser generates syntax trees and manages these trees using a ANTLR syntactic tree management. The generated AST can be further output in ASCII, Latex or XML format.

Having a similar motivation with the previous work, jBTools platform [140] was designed to manage the B language. The tool was developed in Java and proposed a parser responsible for analysis and type-checking of B specifications. For each B file parsed, an XML file is generated and it is used for further processing to generate Java code. BSmart [141] tool uses the jBTools parser as front-end of its environment.

While working on my thesis work, I came across these B parsers that are no longer available. The given download links associated with these parsers do not support anymore or the given repositories are empty.

To develop our B2HLL tool, we use an existing B parser: BCompiler [2], an open source tool that provides complex parsing functions for syntactic and semantic analysis of B models. In the following, we give a brief description of the functionalities of the BCompiler and how it interfaces with our tool.

[1]ANTLR project: https://www.antlr.org/
[2]B Compiler project: https://sourceforge.net/projects/bcomp/

BCompiler is a syntactic and semantic B analyser developed in C++ language. The tool analyses the source file of a B component and it generates a data structure for this component: an annotated abstract syntax tree, called BeTree [165]. Syntax trees are managed and examined via BeTree managers.

The AST is the result of several analysis steps such as lexical, syntactic and semantic analysis. During these steps, a lexems flow is created and various cheques are implemented on the B machines, such as identifier resolution, type checking and visibility rule checking. Furthermore, B0 verification can be realised where compliance with certain B0 rules for translation into C, C+, ADA is checked. The BCompiler also provides functionality for traversing trees and attaching information to tree nodes.

B2HLL tool uses BCompiler as a API to collect B machines concepts to be translated such as constants, variables, properties, invariants and operations, as well as all the necessary information (e.g. typing information). BCompiler parses B source machines and outputs abstract syntax trees represented as BeTrees. The implementation of the B2HLL tool is based on the BeTree representation to create its own data structure. All this information is stored in a C++ class structure.

First, the B2HLL tool loads BCompiler tool and calls the analysis methods of the last one. The tool processes every B component that could be used and outputs a forest of pre-processed BeTrees. Lexical and syntactic analysis are performed where every machine, refinement or implementation of the given source B project is transformed into a BeTree representation. Further the tree is annotated with typing information for each B component. As a result, the B project given as input is parsed and the representation of B source models in the form of BeTrees is obtained.

Further, several classes are created to hold the information used to generate the HLL models. We choose to represent the information extracted from a B component by using *BComponent* class, as detailed further.

Before going further into the translation process we detail the class structure shown in Figure 6.3. The tool processes a given B source project with a root module as follows:

- Each B machine input file is represented as a *BComponent* class. The *BComponent* class allows to access the BeTree representation of the machine, as well as to consult the data structure of this machine. For each machine the data is filled in the pre-processing phase.

- A module is created with *BModule* class for each B specification. A module contains the list of all components for a given B specification, sorted from the most abstract to the most concrete one. For example, Main.mch, Main_r.ref and Main_i.imp are 3 components of the module Main.

- A project is created with *BProject* class and it contains the list of all components and modules present in the B source project.

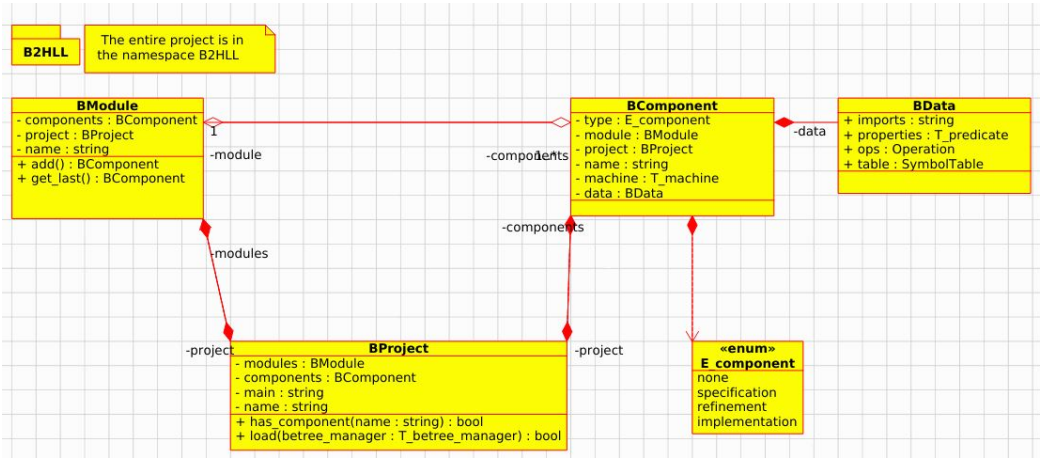

Figure 6.3: Class diagram of the B2HLL information structure

We have chosen to implement our own data structure rather than extending the BeTree structure with translation modules because this makes our tool easier to maintain, and to evolve. Next we continue with further processing on the obtained BProject class structure.

6.1.2 Preprocessing

During preprocessing, the data structure created in the previous step is annotated with additional information useful for transformations for variable evolution, loop or module structuring. The translation environment is created and updated. Each B module (represented by the classes *BModule* and *BComponent*) is processed separately from the others, starting with the most abstract machine and progressing to the most concrete.

We create a data structure (*BData*) for each component to store information about that component. Figure 6.4 depicts the structure in the form of classes. The occurrence of various entities such as variable names, constants, sets, and operation names is recorded

in a symbol table (class *SymbolTable*) that contains all symbols (class *Symbol*) of a component. The symbol table primarily contains local symbols, but it may also include symbol copies from abstract components. This is achieved by traversing the following B clauses :*CONCRETE_VARIABLES, ABSTRACT_VARIABLES, CONCRETE_CONSTANTS, ABSTRACT_CONSTANTS, SETS, LOCAL_OPERATIONS and OPERATIONS*.

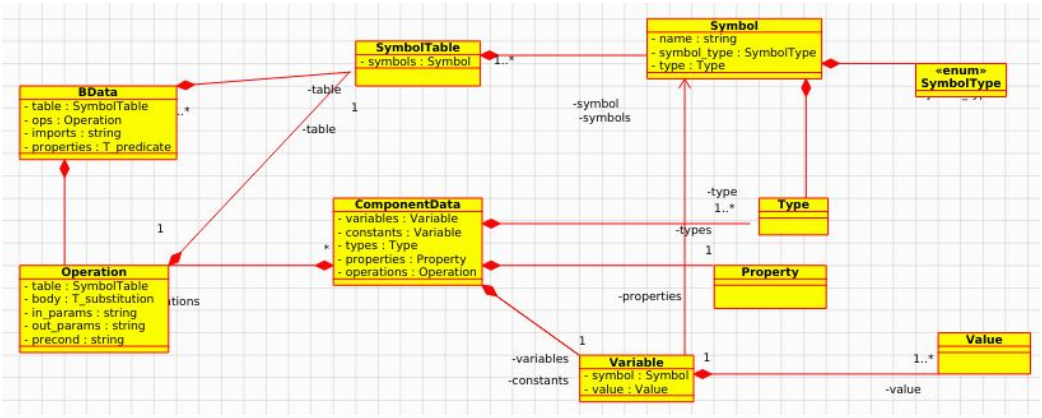

Figure 6.4: Class diagram of data structure

A symbol in the symbol table syntactically represents a B symbol with its type and value. For example, for a concrete integer variable *aa*, the class *Symbol* stores the `name` of the B variable, the `symbol_type` such as concrete constant, variable, enumeration, operation, etc. in our case it is a variable. The `type` of the variable is derived by analysing various static clauses like us : PROPERTIES, INVARIANTS or VALUES and the information required for HLL code generation. We have implemented a *Type* class that constructs and stores variable types in HLL. The translation of the types is also based on the type resolution provided by the BCompiler. Finally, the initial value of the symbol is stored. We find the data needed for a component by targeting the following B clauses:

- the type and value of B constants and sets from *PROPERTIES* and *VALUE* clauses
- the type of variables from *INVARIANTS* clause
- the B model properties from *INVARIANTS*, *PROPERTIES*, or *ASSERTIONS*
- the type of operations from *LOCAL_OPERATIONS* or *OPERATIONS* clause.

The symbol table is further used to verify if a variable is declared, to decide the type of the variable or to determine the scope of a name. The symbol table is implemented as a

hash table, where the symbol itself is the key for the search and it returns the information about the symbol.

For a BComponent, we collect all elements of the machine to be translated, e.g. enumerated sets, constants with their values, variables, invariants, and operations, as well as all necessary information (such as the typing invariant) from the abstract machines.

Further in preprocessing, expressions are flattened to extract essential information for translation. Predicates and expressions in B may be complex and they may be nested in useless parentheses stored in recursive binary predicates, which further complicates parsing. Moreover, there may be multiple typing predicates for the same symbol. All these technical challenges lead us to define rules to flatten predicates, and we separate them into 3 categories: typing, valuation and property. To fill the information, we iterate over the list of predicates of a B clause, we separate the elements properties, type and value and store them in the corresponding field of the symbol under study. In our implementation, we consider the following operators: :, =, <:, <<: for extracting type information. For evaluation, we consider the following operators: :=, ::, : and =.

The processing of the clause PROPERTIES iterates over the conjunction predicate, and each conjunction is processed based on the type category (typing, valuation, property) defined earlier. For example, a conjunction containing the = operator is used to determine the value of a constant and to set the valuation for the corresponding symbol in the symbol table.

Each operation has a table of symbols that initially contains input and output parameters. Local symbols of the operation are not added in this step. Input parameters are typed by analysing the precondition substitution of the operation or of its abstractions. As for typing output parameters, at this level of the process it is difficult to find their type because type deduction is needed. Once we have completed the type resolution and identified the required information for all symbols, we move on to the next step: code generation.

6.1.3 Code Generation

The final part of the process is HLL code generation. Since the two languages have different syntax and semantics, we will use rewriting rules for code generation. The translation rules have a recursive form on the structure of the syntax tree. The translation is designed as an extensible polymorphic type system. Each component of B is translated with its corresponding transformation function. Each transformation is context dependent.

The tool is implemented with a template generation approach. To avoid numerous translations of the same section of a B model, such as for operation calls, code generation is realised in two phases. First, we create templates for static sections of B models through a partial translation of each module. Second, we instantiate the templates with dynamic data to obtain the final HLL model corresponding to a B project. The implicit order provided by the chosen root machine guides the translation process and enables the translation of the imported B modules and the operations used in the model.

Template generation. In B, each machine has its own state (data space), therefore translating from B to HLL may result in translating the same block of instructions with different states. To avoid translation repetition, we generate a translation skeleton with generic data for each B component. This process is used for B machines, but also for B operations. The template is instantiated in the translation process for each new instance of a B machine or each new occurrence of an operation call.

The generated HLL skeleton containing the translation of B clauses without data space updates and environment mapping between B and HLL constructs is referred to as a B component template. A template contains placeholders that are replaced with concrete data from the execution trace of the B model when instantiated. These placeholders contain tokens with a key that may be used to perform specific processing based on the type of token.

Templates are stored using a class hierarchy, shown in Figure. 6.5. A template for a B component is stored using the class *ComponentTmpl* and for B operations the class *OpTmpl* is used. We have defined different types of template elements *IToken* that allow specific processing to generate the HLL code. *ComponentTmpl* stores the syntactic translation of all B clauses needed to instantiate a B component and generate the final HLL model. The HLL translation of the static B clauses (Sets, Constants, Properties, Values) are stored as attributes of this class (*types, constants, constraints*). The translation of a B component initialization clause and the additional assignments added for gluing between states are stored in *definitions*. The attribute *proofs* stores the corresponding HLL code for the invariant translation. A component template contains placeholders for the name of the machine and for state variables.

A B operation is mapped to an operation template stored in class *OpTmpl*, which contains the corresponding HLL code for the body of the operation with placeholders for

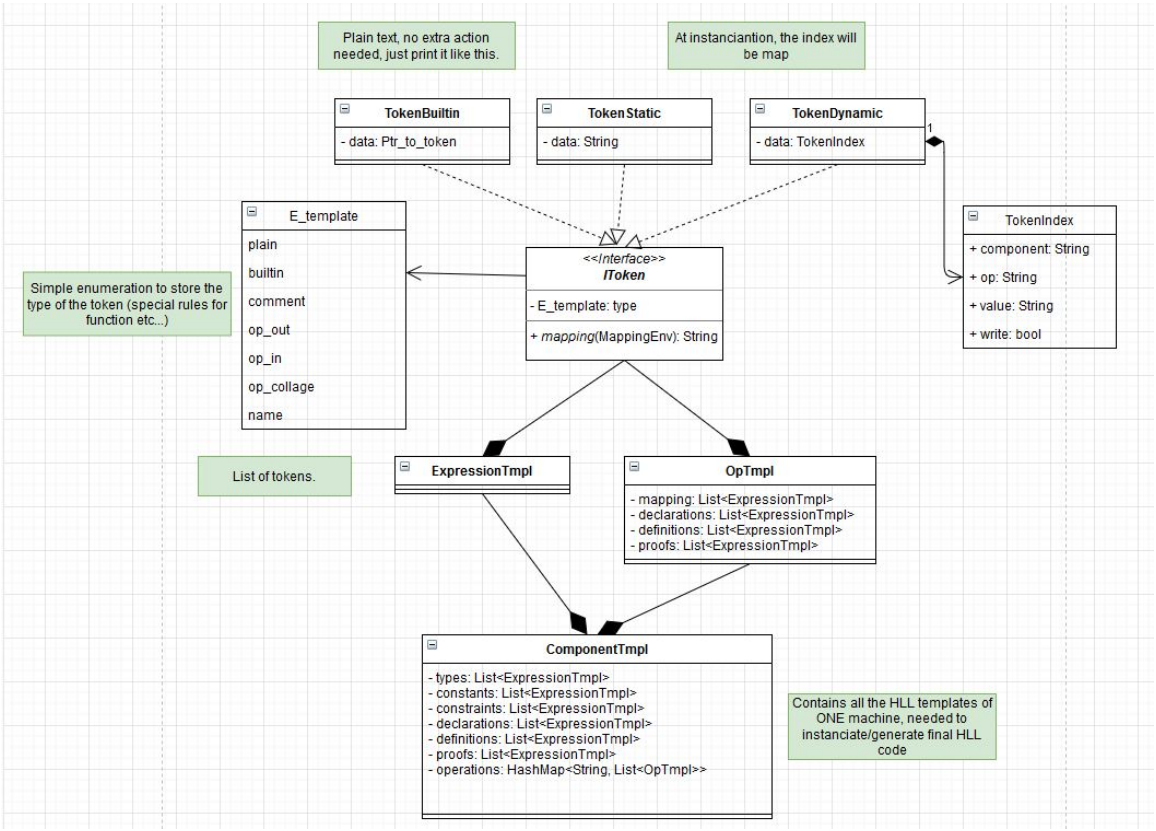

Figure 6.5: Class diagram of template structure

the operation name, parameters, and return type. The generated HLL includes operation body substitutions (attribute *definitions*), the mappings added for the correspondence with the state variables (attribute *mapping*), the generated proof obligations (attribute *proofs*) and the HLL variable declarations (attribute *declarations*). The placeholders are replaced by strings derived from the semantic metadata of the operation node. As a result, when the operation template is processed with the required placeholders, the HLL operation code is generated. Until these placeholders are replaced, the template only outlines the body of the operation.

For each leaf component of a B module, a template is produced. The translation rules implemented for template generation adhere to the concepts presented in Chapter 4. Template generation is implemented by a method, *generateTemplate*, which is the default handler. Based on the type of the node AST, it redirects the content to the concerned method, which translates it. The algorithmic representation of template generation for B

machines and operations is as follows:

```
foreach module M do
    if M is not yet translated then
        generateTemplate(M);
    end
end
Function generateTemplate:
    Translate B data;
    Define intermediate variables vars$<index>$ for each value change of vars ;
    Translate Initialisation ;
    foreach operation O in Operations do
        Generate a HLL Namespace : O$<index>$;
        Add additional code for state mapping between local and global variables;
        Translate the substitution body;
    end
```

Algorithm 1: B Template Generation

Instanciation. At this stage, we invoke the second component of the code generator that parses all the templates and connects them based on B level state changes to produce the final HLL model. In the preceding phase, during template creation, each machine is translated from a syntactic standpoint, without taking into account state changes caused, for example, by operation calls. To avoid instantiation of unused machines, a template is instantiated in the translation process for each new instance of a B machine or each new occurrence of an operation call.

A template is instantiated when its placeholders are replaced by strings containing the corresponding information from the translation environment. For example, when instantiating a template of an operation, this means realising the mapping between formal and effective parameters. The main machine guides the instantiation of templates, since the translation process must follow the exact sequence of state changes of the B model. On this basis, the syntactic influence cone resulting from the main machine allows the instantiation of templates and generates the final HLL. The order of dependencies between the components is respected, e.g. the template of the main machine is instantiated first and the elaboration of the used components is achieved recursively.

During instantiation, the variable bindings are implemented in such a way that the correct dependency order is preserved between the state of the B variables and the produced

HLL flows. To achieve this, we need to keep track of all the changing states of the variables during the translation process. The B2HLL tool keeps a table of symbols in the translation environment to hold the correspondences between B variables and HLL flows. This is implemented through the use of the class *Mapping*, which maps each B state variable or B operation parameter to the relevant version of the HLL identifier (function *Mapping* defined in Section 4.2).

When the root machine *Main* calls an operation O of the machine M, the translation environment is initialised with the mapping between B variables and HLL flows obtained after translating the initialization clause of this machine. Moreover, based on the translation environment passed as a parameter, the placeholders from the template of the operation O are filled with information such as: the index of the operations, the effective parameters of the operations, the version of the global variables. The translation environment corresponding to the machine M is updated with the new version of the variables, based on all the state changes of the variables from the operation O. The state of the calling machine, *Main*, is also updated. In this way, the translation follows the state changes of the different machines.

6.1.4 Translation rules at code level

In Chapter 4, we have presented the main translation rules for B basic constructs. In Chapter 5, we have shown the certification process for this transformation. In full-scale projects, B implementations address more B constructions allowing for more complex data structures. We consider that these complex constructs can be transformed into basic ones.

Types Translation. The built-in types in B, such as $\mathbb{Z}$ and BOOL, are fairly straightforward translated in HLL. Boolean values are represented as boolean variables in HLL and integers are implemented as HLL integers as long as the absence of over and underflow is guaranteed. To avoid overflow issues, the HLL tools provide extra checks, proof obligations are generated to catch undefined behaviours. Note that integers in HLL are finite and bounded while in B they are represented by the mathematical set. In practice, at implementation level, checks are done to verify that the values are within the range from MIN_INT to MAX_INT. This leads to a type definition in HLL. Other basic types such as enumerated sets are translated as enumerated type in HLL. Integer intervals become in HLL a sub-type of Integer. Range checks are realised to ensure that the value of variables are within the limits.

Functions are represented in HLL as arrays. The translation of arrays in B represents the creation of a type specific to the array definition and the declaration of a variable of this type. For example:

```
arr1 ∈ (0..4) → INT
```

```
Types: (int[0,4] -> int) type;
Declarations: type arr1_0;
```

Sets Translation A deferred set in HLL will be represented as an array of booleans. At implementation level, the deferred set in B shall be finite and with a specified cardinality. If no cardinality is specified, a default size will be used (which is controlled by a tool configuration variable).

When translating a set two different elements are generated. First, we define a HLL type definition corresponding to the set. Second, we define a variable having the type of the set and containing all the elements of the set. This variable is a boolean array indexed by the integer range. The membership of the set is represented by the value true for the elements that belong to the set.

Predicates Translation. For propositional operators, the translation to HLL is as expected. HLL supports constructs of propositional logic such as : $\wedge$, $\vee$, $\neg$. For quantifiers, i.e., $\forall$ and $\exists$, we need to extract the type of the bounded variable accordingly. These predicates are translated as HLL universal and existential quantified expressions.

6.2 Deployment at RATP: integration to the PERF project

The PERF methodology is developed at RATP for safety properties verification. It is an *a posteriori* proof based approach that can be applied to already developed software without modifying the existing software of the system (black-box verification). This approach has been applied to several RATP internal projects for CBTC and interlocking safety assessment and also to external missions [1, 49, 50]. The PERF approach is proved to be efficient for software safety assessment as it can detect unsafe bugs during the validation phases. The B2HLL tool is integrated at the same level as the other translators from the PERF toolkit. The primary task of the translators is to provide a semantics-preserving formalisation of the software under evaluation in HLL.

In the development of typical B industry projects, the safety-critical software part of a system is designed in B using as input the informal software specification documents.

These documents describe, at the software level, the safety requirements and the design of the software. From these documents, the decomposition of the software into elementary functions and their sequencing can be derived. Since these documents contain pseudo-code in B, we can point out that the specification documents at the software level are very similar to their counterparts in B models. Validation of B models against their software requirements is not covered by formal proofs and is usually realised by cross-reading, inspections, and by testing [44]. Although abstract B models and informal specifications are close, informal specifications can be error-prone and nothing prevents from transferring these errors to abstract B models. Moreover, formalisation errors can occur and a misunderstanding of the specification can lead to errors being introduced into B models.

In this work, we propose to support the safety evaluation of B models using the PERF method. Here, desired safety properties are proved on HLL models obtained from source B models. The PERF toolkit uses model checking techniques to prove the correctness of the safety properties extracted from the specification documents. The formalisation of the safety properties to be proved on B models is manual and is performed by a safety engineer. It is his task to code the right level of properties, depending on the validation objectives.

Input documents for system or software specification verification include both functional and safety requirements [73]. Typically, functional requirements are considered safety requirements if they have an impact on safety. When proving the safety of the system, a safety analysis must be performed to obtain a collection of safety properties extracted from these safety requirements. Depending on the level of expression and the purpose of the verification, these requirements are coded directly as HLL properties or reformulated from related requirements after further analysis by safety engineers. Writing relevant properties can be viewed as writing relevant requirements. Then, these properties are formalised together with the system behaviour to evaluate the safety of the system. During verification, potential violation of the properties is detected.

During the system design process, high-level safety requirements are defined. They are further refined and transformed into subsystem requirements to finally obtain software or code level requirements. System or high-level safety requirements should describe what the system should do. At this level, the properties associated with the system requirements that ensure the safety of the system should be specified. Software or low-level requirements describe how the system's code should be implemented. System safety properties can hardly be verified directly on the software. At the system level, abstract notions are used , and more

effort is required to model the environment and to map between system and software notions in order to realise formal verification. On the other hand, the difficulty at the software level is to find a property or set of properties that is strong enough to allow verification of the safety behaviour of the entire software programme without fully considering the behaviour of the model when formalising the property.

When it comes to verifying safety properties of large projects, one has to deal with the limitations of exhaustive verification tools or try to bridge the gap between high-level safety properties and software implementations. One of the limitations of model checkers is the explosion of the state space, since the size or complexity of a model has a direct impact on the exploration of states. To address this problem, some solutions have been proposed and implemented to reduce the size of the state space. There are also solutions to reduce the initial model by decomposition or abstraction.

One of these approaches to enable formal verification for complex industrial systems is to implement a decomposition of the proof based on software components and the scope of the property to be proved. High-level properties can be refined based on the functional decomposition of software. Moreover, software components are abstracted, i.e., the software components that are affected by these properties are identified, and the other parts are abstracted from the implementation. In this way, reasoning is facilitated and the existing limitations of tools are addressed [50].

Two use cases have been identified for the B-PERFect method: i) formal verification of equivalence of behaviours and ii) formal verification of safety properties at different levels, system and software. In the following, we describe different validation strategies and how the B2HLL tool together with the PERF method overcomes these challenges.

6.2.1 Non-intrusive component verification

Verification of B models against their informal software specification is realised manually through code review and testing. The B-PERFect method proposes an alternative to this manual process, a non-intrusive formal verification approach at the software component level for systems developed using the B-method.

To validate software components at code level, two approaches are applicable

1. by producing an equivalence proof between the design of the model (software specification including safety requirements at code level) and the code,

2. by defining and proving generic safety properties on the code model.

Equivalence checking between software specification modelling and code guarantees that they are functionally equivalent. When applying this approach to software components developed in B, the B models are first translated into HLL using the B2HLL tool. Then, the software specification is modelled in HLL. Proof obligations are expressed at the HLL level. They assert equivalence between the observable states in both models. Finally, the proof is realised using the toolkit PERF. Equivalence exists if both models produce the same outputs given the same inputs and guarantee that the design and code satisfy the same properties.

Usually, in this case, the correlation between these models is quite straightforward. Moreover, the environment model required for verification is simplified or redundant. This verification technique allows validation of the code against the design requirements, but does not provide information about the correctness of this specification.

To ensure the safety of the software component, the second approach expresses and verifies safety properties at the code level. For this purpose, safety properties are extracted from the specification and formalised in HLL. These properties are inserted into the HLL model generated by the B2HLL tool. It is also verified that the HLL model satisfies all the safety requirements/properties. At the software level, the expression of these properties can be very close to the code. The challenge is to identify a property or set of properties that is sufficient to verify the safety of the model without fully considering the behaviour of the model.

6.2.2 Non-intrusive components integration verification

System level verification. The B-PERFect approach, in addition to the formal verification performed on each software component, can be applied to check the requirements for each sub-system translated model or global system requirements expressed in HLL at system level. The interest of integrating the models at HLL level is twofold. First, it allows to check global properties at system level using a non intrusive approach and second, it allows to have a shared model obtained for various modelling languages (all the models supplied by the stakeholders are translated to HLL models). Indeed, the source models are not modified. They are integrated in a single modelling language.

The B method does not ensure the correctness of the algorithm implemented at code level with respect to system specifications. When speaking about safety assessment at system

level for safety critical systems in railway, several works propose to perform a safety analysis using Event-B formalisation [73, 75]. In [74], it is presented an approach that links Event-B system model with its implementation developed using B.

In [166], it is mentioned that the best level to extract formal properties for safety verification is the safety-related system requirements. System level safety properties can be obtained by performing a hazard analysis of the system. In the railway industry, system level safety properties can be expressed as follows: i). **no collision between trains**, ii) **possibility of overspeeding**, etc.

To prove the correctness of B software implementations with respect to system specifications, we propose the following approach. An important step of the process is to identify system safety properties from the input documents and formalise them in HLL. Then, the proof process described for component verification is performed.

Properties expressed in natural language at the system level can have different formalisation. Some properties can hardly be expressed directly with software terms. In fact, B software implementations manipulate concrete concepts that are not even mentioned in the high-level properties formulation. In this case, it is necessary to perform a decomposition of the property and to apply an additional effort to the environment modelling. The mapping between software variables and system-generic terms represented in the environment must be formalised in the HLL. This is usually done after the software model has been translated using the B2HLL tool. The verification is performed using the translated software model of the B implementations obtained using the B2HLL tool. As mentioned in [75], an important step in formalising safety properties is to justify the modelling decisions made and the hypotheses necessary to ensure the proof. The hypotheses must be stated directly or indirectly in the source documents.

Heterogeneous systems verification. In railway systems, it is often possible for subsystems to be developed in different programming languages. For example, in CBTC systems this may be the case where different manufacturers are responsible for developing the train on-board equipment and the train zone controllers. Consider the case where one of these subsystems was developed using the B method and the other using SCADE. Our approach can be applied as follows: The implementation of each subsystem is translated into HLL using the corresponding translator. The properties are encoded in HLL as safety predicates or assumptions about interactions with the external environment. The environment must

also be modelled. Some caution should be exercised since the correlation between these two implementations is not direct. Efforts should be made to express these integration properties and to analyse the problem of state space explosion by the model checker and reduce it, for example, by slicing or abstraction techniques. The methodology PERF using the B2HLL tool now allows to realise such an integration verification.

Our approach allows to validate safety at system level by integrating information from different components in a common environment and by building a bridge between system requirements and valid implementations.

6.3 Case Study. Train localisation in a CBTC system: the TRPL function

CBTC [167] is a complex system that uses two-way communication between on-board and wayside equipment to ensure a safe and high-performance service. It consists of various sub-systems which rely on the design, specification, configuration and development formalism used by each supplier. A CBTC system provides two main functions: 1) localisation (on-board), and 2) tracking of trains (wayside). *L*ocalisation computes the topological position of trains while *t*racking uses *L*ocalisation to construct train cartography across the entire network.

We are concerned in this case study with Train Reference-Point Localisation *T*RPL function, a *L*ocalisation sub-function. The *T*RPL method determines the new train location p' according to a topology of n line segments, a travelled length approximation d and a train position p referring to a segment identifier, an abscissa and an orientation (see Figure 6.6). To reduce the complexity of the *T*RPL function, environmental assumptions are considered i.e. 1) a railway line is considered as a sequence (consecutive) of segments of equal length associated with an identifier 2) the train orientation is one way and remains the same for all segments that make up the configuration.

Ideally, it is easier for system safety assessment to check the high level system requirements and to verify the safety properties at a high abstract system specification level instead of performing this verification at the implementation level. In general, this is not possible because either the system specification is too abstract to allow such safety requirements to be verified or the level of implementation is too detailed and makes the verification too complex. Compositional verification using assumptions is recommended.

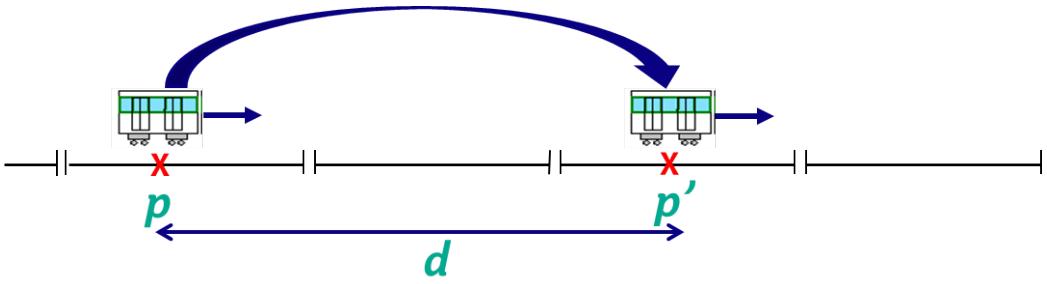

Figure 6.6: TRPL Description

*T*RPL is aligned with the following requirements. First **u**nitary requirements (tested in isolation for the *T*RPL function) and second **integration or system requirements**, including environmental assumptions, are presented.

6.3.1 Unitary requirements

UnitReq1	The train reference point position p' shall be computed according to the given orientation.
UnitReq2	The distance between the current reference point position p and the next reference point position p' shall be equal to the travelled distance d.
UnitReq3	The train reference point position p shall not change when the new position goes beyond the known segments zone n.

6.3.2 Integration or system requirements

For current CBTC systems, trains permanently measure their exact position, speed and travel direction and transmit this information to the wayside devices. Such data, conforming to the configuration data (i.e. data representing the environment where the system evolves) enables the measurement of the region potentially occupied by the train. This information is used in the verification of safety properties such as *"The train is delocalised (i.e. its localisation cannot be used by other system functions) if the position is invalid (e.g. due to wrong or outdated sensing values)"*. To establish this safety property, it is required to guarantee that the train positions are valid ones. These positions are computed from the

maximum train speed, the real length of the train and the configuration of track segments information.

The system requirements on the *T*RPL function expresses that on each consecutive segment crossed by a train, the reference point position of this train is calculated. This function requires that the next segment is free. The following system requirement *SystReq* can be written.

SystReq	The next reference point position shall be on the next segment (adjacent to the current segment position) in the given orientation

This system requirement is related to

- *UnitReq2* which states that the next reference train point position is beyond the current train point position with a distance $d > 0$
- The data computed by the CBTC and conforming to the configuration data enforcing the next reference point position of the train to be in the next section.

We note that, such a system requirement cannot be verified on the standalone *TRPL* function. This requirement is a refinement of a system requirement resulted from the integration of other system functions and of the description of the system environment. This analysis is realised after the development of the system. In order to check this at B level we should modify the actual model and integrate the definition of the environment. This approach is not possible in the actual industrial context and implies to reprove the B code, a time consuming activity.

6.3.3 Formal models for the TRPL case study

In chapter 4, we have described the general transformation principles from B to HLL. Below, we show the B model corresponding to the case study of Section 6.3 and the obtained HLL model with B2HLL tool presented in 6.1.

6.3.4 A B model for TRPL

The B model associated to the TRPL is composed of several machines and refinements. All the properties verified in this B model are safety properties and show that the trains does not

exceed the known zone of block identifiers. The low and up values of a train displacement are respected. Also, we have shown that the train position is changed with respect to initial requirements.

```
IMPLEMENTATION TrainPositioning
...
INVARIANT
/*Typ1*/t_segment = 1..c_nb_segments
/*Typ2*/t_deplacement = 0..c_max_dep
/*Typ3*/t_abscisse = 0..c_segment_length
/*Inv1*/v_segment ∈ t_segment
/*Inv2*/v_segment_before ∈ t_segment
/*Inv3*/v_absOnSegment ∈ t_abscisse
/*Inv4*/v_absOnSegment_before ∈t_abscisse
```

Listing 6.1: B TRPL Context Invariants

Listings 6.1-6.3 show the obtained implementation of the last level of a B refinement. The B model associated to the TRPL defines the context of the model by introducing constants for predefined limits (i.e. maximum number of segments, the length of a segment, maximal distance of displacement). These constants are used to define a topolgy of the railway network in the $Typ1$, $Typ2$ and $Typ3$ invariants. The state variables are: $v\_segment$- a new segment identifier; $v\_segment\_before$ - a previous segment identifier; $v\_absOnSegment$ - an abscissa on the current segment; $v\_absOnSegment\_before$ - a previous abscissa on the segment; and $v\_is\_segment\_found$ - to state if a new position is found in the limit of known zone of segments. They are typed in $Inv1$, $Inv2$, $Inv3$ and $Inv4$.

```
/*UnitReq1*/
(v_segment_before * c_segment_length) + v_absOnSegment_before ≤
(v_segment * c_segment_length) + v_absOnSegment
/*UnitReq2*/
(v_isSegmentFound = TRUE ⇒ ∃dd. ( dd ∈ t_deplacement ∧
(v_segment − v_segment_before) * c_segment_length +v_absOnSegment=
    v_absOnSegment_before + dd))
/*UnitReq3*/
(v_isSegmentFound = FALSE ⇒
(v_segment_before * c_segment_length) + v_absOnSegment_before =
 (v_segment * c_segment_length) + v_absOnSegment)
...
```

Listing 6.2: B TRPL Safety Invariants

Unitary requirements defined in Section 6.3.1 are modelled in the invariant clause as safety properties ($UnitReq1$, $UnitReq2$ and $UnitReq3$). As the travelled distance is always

positive and only one direction is given, *UnitReq1* invariant checks that the previous train position is always less than or equal to the current position. The *UnitReq2* verifies the length of the path between the previous and the actual position if a new position is computed. The *UnitReq3* verifies that the position is unchanged when a new position is not available.

The *INITIALISATION* clause initialises the variables of the model to a value within their range of values. Several operations are introduced, they modify the state variables. The preconditions ensure that the input parameters are type consistent and the new train reference point is correctly computed. The properties from the invariant clause shall always be maintained by the operation. To illustrate our approach, only the implementation level of the $findLoc$ procedure is presented. When the new position of a train remains in its bounds, the *findLoc* procedure changes the train reference point position based on a displacement *i_dep*, a previous segment *i_seg* and abscissa *i_abs* given as parameters.

The *t_segment* variable models the known zone of line segments. The invariant *Inv1* describes that the segment identifier belongs to the type *t_segment* and it does not allow to go beyond the know zone of segments. The *t_abscisse* variable defines an interval of values representing the possible values of an abscissa on segment. Invariant *Inv3* states implicitly that the abscissa shall not exceed the length of a segment.

This main machine describes the move of the current reference point for a train displacement given by odometry devices. In *doLoc* operation is refined this behaviour. The latter imports *TrainPositioning* machine and calls the operation *findLoc* passing as parameters the current block identifier, position on this block and the train displacement. The reference point will be updated accordingly to the displacement *l_xDep*, if its new value is in the known zone of blocks. As the operation *findLoc* modifies the state of the machine, the computation output is returned by *readSegment* and *readAbscissa* operations.

```
OPERATIONS
findLoc (i_seg, i_abs, i_dep) =
 VAR l_x, l_seg IN
   l_x := i_abs + i_dep
 ; l_seg := i_seg
 ; WHILE c_segment_length < l_x ∧ (l_seg < c_nb_segments) DO
     l_x := l_x − c_segment_length
   ; l_seg := l_seg + 1
   INVARIANT
     l_seg ∈ t_segment ∧ l_x ∈ NAT
   ∧ i_seg ∈ t_segment ∧ i_abs ∈ t_abscisse
   ∧ i_dep ∈ t_deplacement
   ∧ (l_seg − i_seg)∗c_segment_length+l_x = i_abs + i_dep
   VARIANT l_x
   END
 ; v_isSegmentFound := bool (c_segment_length ≥ l_x ∧
   (l_seg ≤ c_nb_segments))
 ; IF (v_isSegmentFound = TRUE) THEN
     v_absOnSegment := l_x
   ; v_segment      := l_seg
END END
...
END
```

Listing 6.3: B TRPL Operations

A *while loop* ensures that the segment identifier is *i*ncreased when the train displacement is *g*reater than the length of a segment. The *loop invariant* states that the input parameters respect their typing properties and the new train reference point is correctly computed by preserving the model invariants. The operation *findLoc* is triggered by the main program that models the current TRPL updates for a train displacement given by odometry devices. The developed B model is successfully proved using Atelier B [48] ensuring invariant preservation and thus fulfilling the unitary requirements.

6.3.5 A HLL model for TRPL

This section describes the HLL model that results from translating the B example given above according to the transformation principles defined in Chapter 4.

```
Namespaces: "TrainPositioning_0"{...
//Context Definition
Types:
int [1, c_nb_segments] t_segment;
int [0, c_max_dep]  t_deplacement;
int [0, c_segment_length] t_abscisse;
Declarations:
t_deplacement x_deplacement;
Declarations:
t_abscisse v_absOnSeg;
t_segment  v_seg;
Definitions: //State Variables
v_seg_0       := 1, v_seg;
v_absOnSeg_0 := 0, v_absOnSeg;
v_seg         := v_segment;
v_absOnSeg    := v_absOnSegment;
 ...
//Invariants
Proof Obligations:
(1 ≤ v_segment &
 v_segment ≤ c_nb_segments); //Inv1
(0 ≤ v_absOnSegment &         //Inv3
v_absOnSegment ≤c_segment_length);
//UnitReq1
(pre (v_segment, v_seg_0)*
    c_segment_length +
 pre (v_absOnSegment, v_absOnSeg_0) ≤
v_segment*c_segment_length+
    v_absOnSegment)
//UnitReq2
v_isSegmentFound_1 ->
(((v_segment − pre(v_segment, v_seg_0))
*c_segment_length + v_absOnSegment) =
pre (v_absOnSegment, v_absOnSeg_0) +
::doLoc_0::l_xDep_0)
//UnitReq3
~(v_isSegmentFound_1) ->
(pre (v_segment, v_seg_0) *
c_segment_length +
pre (v_absOnSegment, v_absOnSeg_0) =
v_segment * c_segment_length  +
    v_absOnSegment);
//SystReq
v_seg = pre (v_seg,1) ∨
v_seg = pre (v_seg,1) + 1;
```

Listing 6.4: HLL TRPL Context

```
//Computations
Namespaces: "findLoc_0"{
Definitions: //mapping input parameters
i_dep_0 := l_xDep_0;
i_abs_0 := v_absOnSeg_0;
i_seg_0 := v_seg_0;
//distance computation
l_x_0  := i_abs_0 + i_dep_0;
l_seg_0 := i_seg_0;
//Begin While Iter 0
l_x_1 := l_x_0 − c_segment_length;
l_seg_1 := l_seg_0 + 1;
l_x_2   := if c_segment_length < l_x_0 &
             (l_seg_0 < c_nb_segments)
           then l_x_1 else l_x_0;
l_seg_2 := if c_segment_length < l_x_0 &
             (l_seg_0 < c_nb_segments)
           then l_seg_1 else l_seg_0;
... //End Iter 0
v_absOnSegment_1 := l_x_20; //IF cond
v_segment_1      := l_seg_20; //IF body
v_segment_2 :=if(v_isSegmentFound_1==true)
              then v_segment_1
              else v_segment_0;
v_absOnSegment_2 :=if (v_isSegmentFound_1
    == true)
                    then v_absOnSegment_1
                    else v_absOnSegment_0;
...}
Definitions: //State variables updates
v_segment_1 := ::"findLoc_0"::v_segment_2;
v_isSegmentFound_1 := ::"findLoc_0"::
    v_isSegmentFound_1;
v_absOnSegment_1 := ::"findLoc_0"::
    v_absOnSegment_2;
v_isSegmentFound    := v_isSegmentFound_1;
v_absOnSegment      := v_absOnSegment_1;
v_segment           := v_segment_1;
...}
```

Listing 6.5: HLL TRPL Computations

The HLL model associated to the TRPL is composed of *Namespaces* sections, each of them corresponding to the translation of the last level of B refinement (implementations). For example, $Train\_Positioning\_0$ namespace is the translation of $Train\_Positioning$ machine. Here, we only present an excerpt from the translated HLL model.

The HLL model in Listings 6.4-6.5 is structured as follows: context definition, specification of invariants (proof obligations) and system behaviour description. In Listing 6.4, we show the context definition that is composed of types definitions, variables declaration and cyclic variables definition (those variables updated by the cyclic execution of the system). Note that the real execution of B models is realised in a loop where the computed state variables of an iteration are used as inputs for computations in the next iterations. The HLL model begins with defining types corresponding to $Typ1$, $Typ2$ and $Typ3$ typing invariants originated from the source B model. The B constants used to define the topology of the railway network are modelled as HLL *constants*.

A HLL model describes a system based on input, output and safety properties. We consider the train displacement as input of the TRPL model. The output of the system are the state variables of the B main program. B state variables are represented in the HLL model as flows with a cyclic definition. State variables flows are initialised according to the B *initialisation* clause. The next values in the state variables flows are produced from the transformation of the B operations and the corresponding programming constructs. This behaviour is exemplified in Listing 6.4. For example, the v_seg B state variable is updated with respect to the computed value in the B operation $findLoc$. The new computed values of the variable v_seg_0 are used as input in the model $findLoc\_0$ to compute the next possible values of the variable v_seg. The value of streams representing the train position is modified at each instant according to the new computations given by the cyclic execution of the system.

The unitary requirements of the TRPL model are defined in the *Proof Obligations* section. They correspond to the source B invariants $UnitReq1$, $UnitReq2$, $UnitReq3$. Note that the invariants are described using the *pre* operator ensuring the verification of properties from one execution cycle to the next one. Translating the invariants ensures that the properties verified in the B code are also subject to verification in the generated HLL model.

Listing 6.5 represents the computation of the new train position. For each B operation, a corresponding HLL namespace section, such as "$findLoc\_0$" that represents the translation of the B operation $findLoc$, is generated. Note that each namespace has an index to

count the different calls of the operation. The mapping between formal and effective input parameters is exemplified by the assignment variables: *i_dep_0*, *i_abs_0* and *i_seg_0*. Each B assignment is translated as a HLL stream assignment with a new version of the modified variables.

6.3.6 System analysis

Up to now, all the properties established in B are also expressed as properties in the HLL model. One may ask what is the added value of such a transformation. The interest of integrating the models in the HLL framework is twofold. First, it allows to have a shared model obtained for various modelling languages (all the models supplied by the stakeholders are translated to HLL models) and second, it allows to check global properties at system level using a non intrusive approach. Indeed, the source models are not modified. They are integrated in a single modelling language. In addition to the formal verification performed on each source model using the source modelling verification procedure, it is possible to check the requirements for each translated model or global system requirements expressed in HLL at system level. In the case of supplied B models, they are integrated (composed) into the models already developed.

For example, the system requirement *SystReq* encoded in HLL (not expressed in the B model) as presented in Listing 6.6 requires that, when a train moves, the next segment associated to the new train position is either the same one or the consecutive one. The requirement does not allow trains to move forward to any segment. Only consecutive segment changes are allowed.

```
Proof Obligations: //SystReq
v_seg = pre (v_seg,1) ∨ v_seg = pre (v_seg,1) + 1;
```

Listing 6.6: System level Requirement

This requirement is not fulfilled by the produced HLL model shown above and the proof engine revealed a counter-example. The corresponding scenario was analysed to understand the risk related to this property violation. This analysis revealed a possible environment restriction hypothesis related to the limitation of the maximum distance travelled (i.e the speed, the period of sensing position, etc.) in the cycle. As mentioned in Section 6.3.2, to show that this property holds, in this particular setting, the proof relies on the model (B

translated code) and the hypothesis or exported constraints about the data configuration. When all this information is integrated the property holds.

6.4 Model Animation. The transformation at work

The last step of the formal verification and validation process we have set up when using the PERF framework is described in this section. The critical industrial application context requires an assessment of the quality of the defined transformation. We address the certification of the defined transformation process. The certification consists in formally guaranteeing semantic preservation after translation i.e. we prove that the transformation of a B model to a HLL model is semantic preserving. Fig. 6.7 shows the overall approach based on the translation rules presented in chapter 4 and their formalisation in Isabelle/HOL shown in chapter 5.

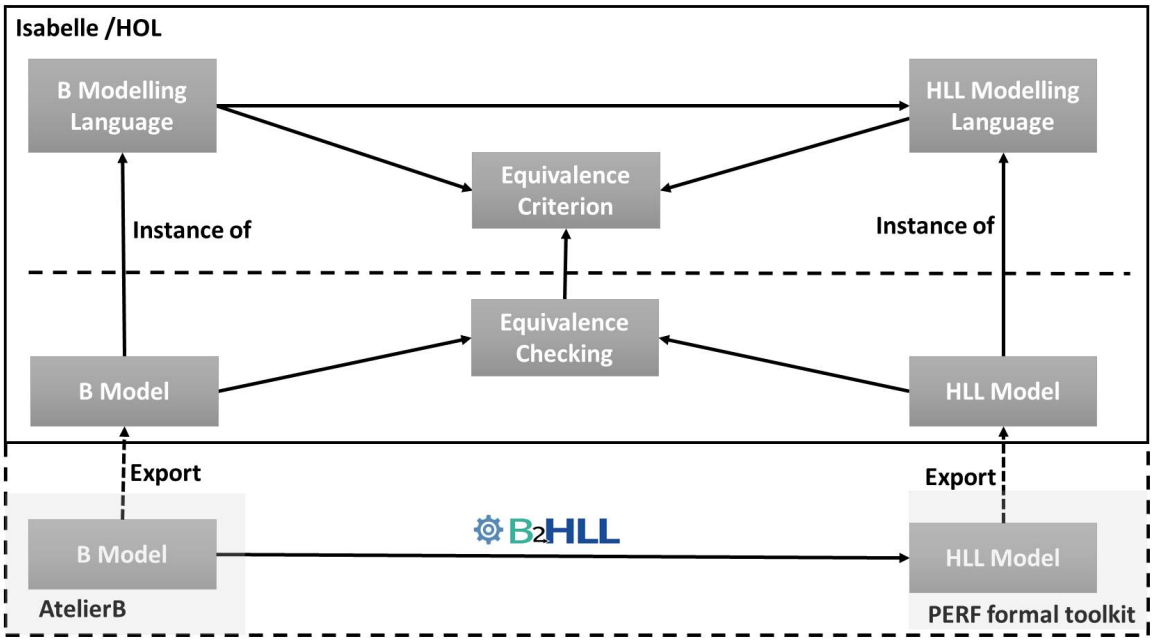

Figure 6.7: A formal framework of certified translator B2HLL

In the process of safety assessment, the validation of the translator is an important step. In order to achieve this step, the execution of models has been considered as one mean to arrive at this end [168, 169].

The concept of model animation is widely studied and represents a key feature for model based development tools. For instance, the idea of executing HOL specifications and

checking formal specifications on a set of randomly generated test cases is presented in [170]. Animation was considered for B and Event-B [23] formal models in tools like: ProB [40], JetB[171] or Brama [172]. These tools use animation on early stages of the development process for specification validation. In the context of formal specification validation like [173], [174] or [175] they focus on the correctness of the formal model with respect to the high level specification. We aim at validating formal models encoded in Isabelle/HOL with respect to their software implementations.

The approach proposed in this section is similar to [176] where an attempt to bridge the gap between code and formal specifications using model animation is presented. However, their focus is on test cases generation for numerical computations.

The previous Chapters 4 and 5, showed a complete transformation process together with a proof of equivalence. Theorem proving is a rigorous standard approach that can be used to prove model properties in form of lemmas and theorems by checking possible states of a system thanks to the availability of deferred sets and symbolic manipulation. Trying to prove an incorrect proposition may lead to dead-ends or considerable time loss. Therefore, the idea of debugging proofs by testing the conjunctures is helpful. Model animation is a powerful technique to perform such tests. We have used a model animator, available in the Isabelle/HOL tool, to validate our transformation on several examples, they helped to identify the right formalisation of definitions, lemmas and theorems. Animation allows to observe the behaviour of formal models and to validate it through instantiating the given model. When using symbolic evaluation of input values in the developed formal models, counter examples can be identified and axiom witnesses can be provided for checking consistence.

In our work, we apply animation techniques provided by the Isabelle/HOL framework,

- to debug incorrect goal by pointing out inadequate formalisation, hypothesis inconsistencies;
- to execute our formal specification in order to monitor whether the output of the B2HLL tool corresponds to the formal specification.

We realised a model animation of the main equivalence theorem of section 5.5.3 by showing first that the transformation of each syntactic category of B is transformed into the desired HLL code and second that the states of both languages are equivalent (bi-similar).

Moreover, the transformation function definition from B to HLL is also validated by means of animation of code examples.

Even though the B to HLL transformation is automatic, the model animation is still interactive, i.e step by step.

Application to the TRPL case study

We illustrate our model animation approach on the *T*RPL case study presented in Section 6.3 for the models defined in sections 6.3.4 and 6.3.5. With the help of B2HLL tool we transformed the B model into HLL model. The formalised Isabelle/HOL models of B and HLL modelling languages are animated following two approaches.

- First, the HLL code is obtained formally by applying the transformation function on the B model.
- Second, embedding of the HLL model in Isabelle/HOL resulting from the B2HLL tool.

In this way we can show that the HLL model obtained by the B2HLL tool is equivalent to the HLL model obtained from the Isabelle/HOL transformation. By animating the main equivalence theorem using instantiation, we have shown that the output computed by the B2HLL tool is equal to the output corresponding to the execution of the B model, with respect to a given mapping.

B Model. The developed B models of the **TRPL** case study from Section 6.3.4 are embedded (exported as an instance) in Isabelle/HOL using the formalisation of B language presented in Chapter 5. Listing 6.7 shows an excerpt from the B implementation model of the TRPL example, the *findLoc* procedure. All the TRPL operations are directly encoded in Isabelle/HOL applying the formalised B abstract syntax. Listing 6.8 presents the corresponding Isabelle/HOL instance model defined the TRPL example.

Each B instruction presented in Listing 6.7 is formalised as an Isabelle/HOL $definition$. B operation *findLoc* is defined in Isabelle/HOL by *TrainPositioning_findLoc* with a $b.Bl$ block of instructions. The operation body is represented as the composition of two definitions: first a block of two assignments for l_x and l_seg ($findLoc\_locals\_assigns$) and second a loop instruction block ($findLoc\_while$). The loop condition is encoded as $findLoc\_while\_cond$ using Boolean expressions syntax b_exp as presented in Listing 5.3.

The *findLoc_while_body* definition defines the loop body and finally the loop is defined as an unfolded recursive conditional in the *findLoc_while* definition using the *b_while_to_if* Isabelle/HOL function of Listing 5.7. To keep the coherence with the initial B models, notions as machine and operation name are encoded through variable name prefixes.

```
OPERATIONS
findLoc (i_seg, i_abs,
    i_dep) =
 VAR l_x, l_seg IN
   l_x := i_abs + i_dep
 ; l_seg := i_seg
 ; WHILE c_segment_length <
     l_x ∧
       (l_seg <
    c_nb_segments)
   DO
     l_x := l_x −
    c_segment_length
   ; l_seg := l_seg + 1
   INVARIANT ...
   VARIANT l_x
   ...
 END
```

Listing 6.7: B TRPL Operation from Listing 6.3

```
    (* findLoc (...) *)
definition "TrainPositioning_findLoc =
 b.Bl[findLoc_locals_assign, findLoc_while]"
    (*l_x := ...; l_seg := ...*)
definition "findLoc_locals_assign = b.Bl[
 (b.Assign findLoc_l_x (b.Plus findLoc_i_abs
    findLoc_i_dep)),
 (b.Assign findLoc_l_seg (b.aexp.AVar findLoc_i_seg)
    )]"
    (* while condition *)
definition "findLoc_while_cond = b.bexp.And (
 b.bexp.Lt (TypingPos_c_segment_length  findLoc_l_x)
 b.bexp.Lt (findLoc_l_seg TypingPos_c_nb_segments))"
    (* while body: l_x := ...; l_seg := ...  *)
definition "findLoc_while_body = b.Bl[
 (b.Assign findLoc_l_x (b.Minus findLoc_l_x
    TypingPos_c_segment_length)),
 (b.Assign findLoc_l_seg (b.Plus (findLoc_l_seg) (b.
    aexp.Value 1)))]"
    (* while execution*)
definition "findLoc_while =
  b.b_while_to_if n findLoc_while_cond
    findLoc_while_body"
...
```

Listing 6.8: B TRPL model in Isabelle/HOL

HLL model. The resulting HLL model, obtained by transformation with B2HLL, is exemplified in Listing 6.9. This is an excerpt of the model presented in Section 6.3.5 and corresponds to the B code shown in Listing 6.7. All the state variables are flattened. Note that the HLL formalisation in Isabelle/HOL does not take into account the notion of *Namespaces*. To address this issue the HLL variable names are prefixed with the name of the namespace where they are declared.

In state-based languages, a major challenge is relative to sharing states. In presence of sharing, the translation must preserve the initial reasoning about the values of variables. B state variables are represented as HLL streams. Each state variable named $VarName$ is duplicated using an integer i suffix $VarName\_i$ to avoid side effects and to allow the HLL

model to observe all the internal variables behaviours.

Here again Isabelle/HOL definitions are used to set up the HLL models. Listing 6.10 shows the Isabelle/HOL embedded HLL model. *find_Loc_0* namespace is formalised as *TrainPositioning_findLoc_Hll* definition for block of instructions. The HLL stream variable names are prefixed with the name of the namespace where they are declared as *findLoc_l_x[i]* and *findLoc_l_seg[j]*.

```
Namespaces: "findLoc_0"{
Definitions:
    // init
l_x_0  := i_abs_0 + i_dep_0;
l_seg_0 := i_seg_0;


    //Iter 0, while body
l_x_1 := l_x_0 -
    c_segment_length;
l_seg_1 := l_seg_0 + 1;

    //Iter 0, while execution
l_x_2 := if c_segment_length
    < l_x_0 &
          (l_seg_0 <
    c_nb_segments)
       then l_x_1 else
    l_x_0;
l_seg_2:= if c_segment_length
     < l_x_0 &
           (l_seg_0 <
    c_nb_segments)
        then l_seg_1 else
    l_seg_0;
...}
```

Listing 6.9: HLL TRPL Operation from Listing 6.5

```
    (*Namespace findLoc_0*)
definition "TrainPositioning_findLoc_Hll =
 hll.Bl[ findLoc_locals_assign_Hll,
    findLoc_while_body_Hll0, findLoc_while_Hll0,
    ...]"

    (*init: l_x_0 := ...; l_seg_0 := ...;*)
definition "findLoc_locals_assign_Hll = hll.Bl[
 (hll.Assign findLoc_l_x0 (hll.Plus findLoc_i_abs0
    findLoc_i_dep0))),
 (hll.Assign findLoc_l_seg0 (hll.aexp.AVar
    findLoc_i_seg0))]"

    (*if condition:  *)
definition "findLoc_while_cond_Hll0 = hll.bexp.And
 (hll.bexp.Lt (TypingPos_c_segment_length0
    findLoc_l_x0)
 (hll.bexp.Lt (findLoc_l_seg0
    TypingPos_c_nb_segments0))"

    (*Iter 0, while body: l_x_1:=...; l_seg_1:=...;*)
definition "findLoc_while_body_Hll0 = ...

    (*Iter 0, while execution: l_x_2:=...; l_seg_2
    :=...;*)
definition "findLoc_while_Hll0 = hll.Bl[
 hll.Assign findLoc_l_x2
  (hll.exp.If findLoc_while_cond_Hll0 findLoc_l_x1
    findLoc_l_x0),
 hll.Assign findLoc_l_seg2
  (hll.exp.If findLoc_while_cond_Hll0 findLoc_l_seg1
    findLoc_l_seg0)]"
...
```

Listing 6.10: HLL TRPL model in Isabelle/HOL

This behaviour is exemplified for the *l_x_0* and *l_seg_0* streams corresponding to *findLoc_l_x0* and *findLoc_l_seg0* in Listing 6.10. Similar to B, assignments are expressed in Isabelle/HOL using the functions defined in Section 5.3.1.

The B loop (*while*) construct is transformed into a recursive conditional statement. All the variables modified in the body of the loop from $findLoc$ operation become HLL assignements with stream value conditioned by the condition of the while (see variables l_x_2, l_seg_2 from Listing 6.5).

Conditional expressions are transformed in two steps. First the *then* and *else* branches are translated, and then the conditional expression is built. The condition of the HLL IF expression from Listing 6.9 is formalised as $findLoc\_while\_cond\_Hll0$ definition. The $findLoc\_locals\_assign\_Hll$, $findLoc\_while\_body\_Hll0$ and $findLoc\_while\_Hll0$ encode respectively the initialisation of the loop, its body and the recursive condition to run the loop of Listing 6.9.

Validation scenarios. The validation of the translation rules implemented in the B2HLL tool can be achieved by a correct translator specification and implementation. During the formalisation, a significant amount of time is spent on debugging specifications and theorems. Usually, inconsistencies are discovered during failed proof attempts. One solution is to validate by "running" the specifications using assigned values to state variables in order to conjecture and evaluate them.

Firstly, we instantiate the formal specification defined and proven in Isabelle/HOL theorem prover. Listing 6.11 shows two initial states σ_B for B, and σ_{HLL} HLL. Each state is represented by a pair *(variableName, value)*. Initially, the B and HLL variables are associated to undefined value.

```
abbreviation
"(σB::b.env) ≡ (λv. (case (snd v) of
                        Tval.Bool ⇒ b.B
    undefined
                      | Tval.Int  ⇒ b.I
    undefined))"
abbreviation
"(σHLL::hll.env) ≡ (λv. (case (snd (fst v)
    ) of
         Tval.Bool ⇒ hll.B (λi. undefined)
       | Tval.Int  ⇒ hll.I  (λi. undefined
    )))"
```

Listing 6.11: Initial states

```
lemma
    TrainPositioning_findLoc_equivalence
    :
"(G,m) = Transformation
    TrainPositioning_findLoc n⟹
 (b.meaning_instruction
    TrainPositioning_findLoc σB)
                ≅m
 (hll.meaning_instruction G σHLL)"
by
 (rule Equivalence[OF ...],simp)
```

Listing 6.12: Equivalence theorem simulation

Listing 6.12 shows the instantiation of the bi-simulation relation for the TRPL model

defined in section 6.3.3. The HLL code (G) and the mapping (m) are formally obtained by applying the transformation function ($Transformation$) on the source B model ($TrainPositioning\_findLoc$, shown in Listing 6.8). The interpretation of B and HLL models ($b.meaning\_instruction$ and $hll.meaning\_instruction$, respectively), are applied at initial states. The $\cong$ equivalence relation holds according to the obtained mapping m thanks to the global equivalence theorem $Equivalence$.

Secondly, we use model animation to show that the output HLL model computed by the B2HLL tool is equivalent to the source B model, with respect to the defined transformation rules. We show that the state of the obtained HLL model is equivalent to the original B state. Listing 6.14 defines the animation of the presented B and HLL **TRPL** models and the simulation of the defined equivalence relation we have proposed in Chapter 5.

```
(*Resulting mapping of the
    transformation*)
definition "t = Map.empty(...
 findLoc_i_abs ↦ (findLoc_i_abs0,
    findLoc_i_abs0),
 findLoc_i_seg ↦ (findLoc_i_seg0,
    findLoc_i_seg0),
 findLoc_i_dep ↦ (findLoc_i_dep0,
    findLoc_i_dep0),
 findLoc_l_x   ↦ (findLoc_l_x20,
    findLoc_l_x20),
 findLoc_l_seg ↦ (findLoc_l_seg20,
    findLoc_l_seg20) ...)"
```

Listing 6.13: Mapping definition between B and HLL

```
lemma
"(b.meaning_instruction
    TrainPositioning_findLoc σ_B) ≅_t
 (hll.meaning_instruction
    TrainPositioning_findLoc_Hll σ_HLL)"
...
  apply (subst def_findLoc)
  apply (subst b.meaning_instruction.simps)
  apply (simp only:b_while_to_if.simps)+
...
  apply (subst findLoc_call_Hll_def)
  apply (subst hll.meaning_instruction.simps
    )
  apply (subst meaning_equiv_def)
  apply (simp)
  done
```

Listing 6.14: Equivalence between B and the HLL from B2HLL tool

The $\cong$ equivalence relation holds on the semantic interpretation of the $TrainPositioning\_findLoc$ (model B) and $TrainPositioning\_findLoc\_Hll$ (model HLL) in the initial states σ_B and σ_{HLL} and the mapping t between B variables and pairs of read and write HLL streams. We show that the state obtained after the execution of the B TRPL model is equivalent to the HLL state obtained after the execution of the generated HLL model, with respect to the mapping t defined in Listing 6.13 associating B variables to pairs of read and write HLL streams. The evaluation of expressions is achieved using term substitutions in the proof goal using the Isabelle's *simplifier* built-in term rewriting engine. At this level, it becomes possible to observe step by step states evolutions (i.e. traces) after

expanding the corresponding definitions.

At this level both HLL codes, Isabelle/HOL HLL code obtained by the $Transformation$ function (Listing 6.12) and HLL code obtained by the B2HLL tool (Listing 6.10), are described in Isabelle/HOL. Last, for validation purpose, when state variables are valued, the Isabelle/HOL animation tool shows that these two pieces of code give the same results.

Remark. One may ask why animation is performed at the HLL model level and not on the B models. Indeed, powerful animation tools like ProB [40] or Brama [172] are available for B models. Animating HLL models offers a double interest. First, such animation makes it possible to observe intermediate states that are not accessible at the B models level without refactoring the B models themselves. Second, it offers the capability to animate B models integrated to other HLL models in the PERF framework. Finally, animation offers a technique for testing that semantic preservation holds.

This approach is applied to several case studies provided by RATP and the industrialisation of the tool is ongoing.

6.5 Summary

This chapter illustrates the use of the HLL language as a basis for safety properties verification in order to bridge the gap between the software specification, such as the formal development in B, and the verification techniques on system level. The development of the B2HLL tool and its use in the overall process was described. The proposed approach was exemplified on a case study and it has been integrated in the global PERF framework available at RATP.

Part III

Conclusion

Chapter 7

Conclusion and perspectives

7.1 Conclusion

This thesis investigated the applicability of PERF, an industrial toolset that allows formal verification of systems independent of their development process, to software developed in B. To this end, this thesis presented our approach to generate verifiable HLL code from a model described as B code. We focus on the core concepts to ensure semantics preservation when translating B implementations into the dataflow language HLL. The semantic differences between the two studied languages are pointed out and a general translation scheme is proposed. We describe a translation process and a set of translation principles for the constructs that require a particular attention.

Our concern is validation and verification of systems developed by different stakeholders using their own modelling languages and development processes. We have investigated black-box validation and verification procedures. We have shown that formal modelling techniques provide a rigorous solution to enable integrated verification and validation activities.

A complete formal verification process for checking software and system level requirements on B models is shown. The approach consists of integrating B models, environment assumptions and constraints into a single modelling framework (HLL). In our work, a formal technique related to model transformation has been defined to verify and validate safety critical software developed using the B modelling language. The HLL language is used as a basis for safety property verification to bridge the gap between software specification, such as formal development in B, and system-level verification techniques. This work makes an important contribution to the integration of information from different components of the

system and an integrated verification process.

Moreover, in our work, a formal framework has been proposed to guarantee the correctness of the translation from B models to HLL models. The correctness of the translation rules is proved in Isabelle/ HOL theorem prover. An equivalence proof between B and HLL semantics based on a bi-simulation relation was established. It guarantees that the translation rules implemented in the B2HLL tool are correct, i.e. semantics-preserving according to the defined equivalence relation. The formalisation and associated proofs presented in this work can be easily extended to other transformations of state-based languages to HLL.

One of the most interesting results of our approach is the capability to animate the models on the formal Isabelle/ HOL modelling side. This capability provides a way to formally debug the models of the systems. It can also be used to check the correctness of transformations and to tune the transformation rules.

A tool B2HLL for automatic translation has been developed. The prototype developed allows machines written with the B language at the implementation level to be translated into HLL. This tool still needs to be further developed and improved with the help of new translation rules in order to be used in an industrial process.

Code generation for B is not a new concept, as there are already other tools that allow users to translate B models into different programming languages. The purpose of our tool is not to obtain executable code, but to propose a different approach for validating these models. On the other hand, our tool is validated and this gives the user another layer of confidence when it comes to safety verification of safety critical systems.

This work was carried out as part of an industrial project within RATP. The proposed approach was integrated with the global PERF framework available at RATP. The results obtained were so promising that the decision was taken to industrialise the proposed approach. In fact, the developed approach is currently being integrated into the PERF tool suite used at the RATP company.

The results of this work could be of interest to both researchers and safety departments of industrial companies looking for a method to verify the safety, security and correctness properties of their system models.

In an industrial setting this work allows to

- increase confidence in the security of critical software
- reduce the cost of safety assessment by reducing the number of requirements to model

and prove (due to the rise in abstraction).

7.2 Future Work

The work presented in this thesis can be extended in different ways. Extensions can be implemented at the transformation level, at the tool level, or at the general proof process level using the PERF methodology. We explain the proposed future work in the following.

- One of our objectives is to extend the verification process proposed in this thesis to higher abstraction levels of B developments (refinements). Such an extension offers the capability to perform formal verification at early stages of the development and avoid time and resource consuming verification at code level due to the potentially high number of state variables. In addition, cross models system requirements could be checked before code level is reached saving development resources. This implies formalising transformation rules for abstract constructions such as B abstract substitutions `ANY`,`CHOICE`, `SELECT` or relations operators. Indeed, abstract levels may contain non-deterministic actions which need to be adressed.

- We plan to implement a complete transformation process together with an equivalence proof to show the correctness of the transformations treated at the B abstract level. To this end, we expect to complete the defined semantics of B and HLL in Isabelle/HOL, to characterise the semantics of all abstract B constructions as well as those of all HLL constructions, and validate the transformation from B to HLL at this level.

- The B2HLL developed prototype allows machines written with the B language to be translated into HLL. This tool will need to evolve and to be improved with the help of other translation rules in order to be used in an industrial process.

- An important remaining issue in our work is the well known state space explosion problem, for model checking tools. Depending on the complexity of the B models, this work may require translation optimisations or the use of decomposition techniques to reduce the state space at the HLL level after translation from B. The proposed approach is to consider operations (functions and procedures) as black boxes abstracted by their before-after predicates.

By doing so, the system is modelled only on observable states and state changes are translated in HLL as BAP of B operations. This can be implemented at different B abstraction levels, e.g., implementation level or abstract machine level. The advantage here is that when checking a property in HLL that has implications for certain functions, the other functions can be abstracted by their BAP predicates and these predicates can be transformed into HLL instead of transforming the whole function. The main obstacle remains the formalisation of gluing invariants that glue abstract and concrete variables together.

We plan to work on a tool that extracts these BAP from the B operations to automate the process and thus add this option to the B2HLL tool.

- Currently, the encoding of B and HLL specific models into the Isabelle/ HOL environment is realised manually. We plan to develop a tool that transforms B and HLL models into the semantics defined in Isabelle/HOL, so that the process of animating models becomes fully automatic and so that a certificate can be generated directly confirming the correctness of the transformation from B to HLL for a specific model.

- B2HLL translates B loops as IF constructions into HLL. Since the transformation of these while loops must be bounded, we use the maximum number of iterations of the loop to unfold them. B2HLL cannot automatically optimise the number of iterations needed to terminate the loop. We plan to explore, using a constraint system, the approximate number of iterations for each loop. This could be further helpful because it reduces the number of states at the HLL level, making the model checker's tasks easier.

- Currently, the PERF methodology is used for safety assessments of safety-critical software. We plan to extend the software-level work done in this thesis for system-level verification. We propose to realise an abstraction of the system and validate it with formal proofs. Furthermore, the safety properties validated at the system level could be verified at the software level. Thus, a global safety proof from the system to the code can be realised.

- The feedback on the application of the PERF method shows that the validation time of the models is proportional to their sizes. Issues concerning the translator's performance and complexity should be addressed. In particular, reducing the state space, reducing

time and memory consumption, and analysing multi-cycle execution traces need to be addressed. As HLL models are further fed into a model checker, optimizations need to be made to avoid limitations of these types of techniques.

- As a result of my study, further research could be to investigate a more complex system using B2HLL tool, involving safety critical railway system definition experts, HLL experts and proof experts. An interesting case study is to validate a complete CBTC system with on-board and trackside subsystems and verify the properties of the integration of these systems using the PERF methodology.

Bibliography

[1] Nazim Benaissa, David Bonvoisin, Abderrahmane Feliachi, and Julien Ordioni. The perf approach for formal verification. In *RSSRail*, pages 203–214, 2016.

[2] Julien Ordioni, Nicolas Breton, and Jean-Louis Colaço. HLL v.2.7 Modelling Language Specification. Technical report, RATP, 2018.

[3] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9):1305–1320, 1991.

[4] J. R. Abrial. *The B-book: assigning programs to meanings*. Cambridge Univ. Press, 1996.

[5] Tobias Nipkow, Markus Wenzel, and Lawrence C. Paulson. *Isabelle/HOL: A Proof Assistant for Higher-order Logic*. Springer-Verlag, 2002.

[6] Alexandra Halchin, Abderrahmane Feliachi, Neeraj Kumar Singh, Yamine Aït Ameur, and Julien Ordioni. B-perfect - applying the PERF approach to B based system developments. In *RSSRail*, pages 160–172, 2017.

[7] Alexandra Halchin, Yamine Ait Ameur, Neeraj Singh, Abderrahmane Feliachi, and Julien Ordioni. Certified Embedding of B Models in an Integrated Verification Framework (regular paper). In *International Symposium on Theoretical Aspects of Software Engineering (TASE 2019), Guilin, Chine*, 2019.

[8] Alexandra Halchin, Neeraj Kumar Singh, Yamine Aït Ameur, Julien Ordioni, and Abderrahmane Feliachi. Validation of Formal Models Transformation through Animation. 2019.

[9] Alexandra Halchin, Yamine Aït Ameur, Neeraj Kumar Singh, Julien Ordioni, and Abderrahmane Feliachi. Handling B models in the PERF integrated verification framework: Formalised and certified embedding. *Sci. Comput. Program.*, 196:102477, 2020.

[10] Alexander Backlund. The definition of system. *Kybernetes*, 29:444–451, 2000.

[11] En 50128 standard: Railway applications - communication, signalling and processing systems - safety related electronic systems for signalling. Standard, European Committee for Electrotechnical Standardization, 2011.

[12] Software considerations in airborne systems and equipment certification. Standard, RTCA Inc., 2011.

[13] Jonathan Bowen and Victoria Stavridou. Safety-critical systems, formal methods and standards. *Software Engineering Journal*, 8:189–209, 07 1993.

[14] John A. McDermid. 7 - formal methods: use and relevance for the development of safety-critical systems. In Phil Bennett, editor, *Safety Aspects of Computer Control*, pages 96–153. Butterworth-Heinemann, 1993.

[15] D. Bjørner. Development of transportation systems. In *ISoLA*, 2007.

[16] Yamine Ait-Ameur and Dominique Méry. Making explicit domain knowledge in formal system development. *Science of Computer Programming*, 121:100–127, 2016. Special Issue on Knowledge-based Software Engineering.

[17] R. Milner. *Communication and Concurrency*. Prentice-Hall, Inc., USA, 1989.

[18] Robin Milner. *A Calculus of Communicating Systems*, volume 92 of *Lecture Notes in Computer Science*. Springer Netherlands, Netherlands, 1980.

[19] P. Van Eijk and Michel Diaz. *Formal Description Technique Lotos: Results of the Esprit Sedos Project*. Elsevier Science Inc., USA, 1989.

[20] Antony Galton, editor. *Temporal Logics and Their Applications*. Academic Press Professional, Inc., USA, 1987.

[21] J. M. Spivey. *The Z Notation: A Reference Manual*. Prentice-Hall, Inc., USA, 1989.

[22] *The Vienna Development Method: The Meta-Language*, Berlin, Heidelberg, 1978. Springer-Verlag.

[23] Jean-Raymond Abrial. *Modeling in Event-B: System and Software Engineering*. Cambridge University Press, 1st edition, 2010.

[24] Yves Bertot and Pierre Castéran. *Interactive theorem proving and program development. Coq'Art: The Calculus of inductive constructions.* 2004.

[25] S. Owre, J. M. Rushby, and N. Shankar. Pvs: A prototype verification system. In Deepak Kapur, editor, *Automated Deduction—CADE-11*, pages 748–752, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

[26] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, August 1975.

[27] C. A. R. Hoare. An axiomatic basis for computer programming. *Commun. ACM*, 12(10):576–580, October 1969.

[28] Edmund M Clarke, Orna Grumberg, and Doron A. Peled. *Model checking*. MIT Press, London, Cambridge, 1999.

[29] Christel Baier and Joost-Pieter Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[30] Tomás Grimm, Djones Lettnin, and Michael Hübner. A survey on formal verification techniques for safety-critical systems-on-chip. *Electronics*, 7:81, 05 2018.

[31] Michael J. C. Gordon, Robin Milner, and Christopher P. Wadsworth. *Edinburgh LCF*, volume 78 of *Lecture Notes in Computer Science*. Springer, 1979.

[32] Markus Wenzel. *Isabelle, Isar - a versatile environment for human readable formal proof documents*. PhD thesis, Technical University Munich, Germany, 2002.

[33] Stephan Merz. Model checking: A tutorial overview. In F. Cassez et al., editor, *Modeling and Verification of Parallel Processes*, volume 2067 of *Lecture Notes in Computer Science*, pages 3–38. Springer-Verlag, Berlin, 2001.

[34] Edmund M. Clarke, William Klieber, Miloš Nováček, and Paolo Zuliani. *Model Checking and the State Explosion Problem*, pages 1–30. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[35] Nina Amla, Xiaoqun Du, Andreas Kuehlmann, Robert P. Kurshan, and Kenneth L. McMillan. An analysis of sat-based model checking techniques in an industrial environment. In Dominique Borrione and Wolfgang Paul, editors, *Correct Hardware Design and Verification Methods*, pages 254–268, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[36] Mukul R. Prasad, Armin Biere, and Aarti Gupta. A survey of recent advances in SAT-based formal verification. *International Journal on Software Tools for Technology Transfer*, 7(2):156–173, 2005.

[37] R. Sebastiani. Lazy satisability modulo theories. *J. Satisf. Boolean Model. Comput.*, 3:141–224, 2007.

[38] Alessandro Cimatti, Edmund Clarke, Enrico Giunchiglia, Fausto Giunchiglia, Marco Pistore, Marco Roveri, Roberto Sebastiani, and Armando Tacchella. Nusmv 2: An opensource tool for symbolic model checking. In Ed Brinksma and Kim Guldstrand Larsen, editors, *Computer Aided Verification*, pages 359–364, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[39] Roberto Cavada, Alessandro Cimatti, Michele Dorigatti, Alberto Griggio, Alessandro Mariotti, Andrea Micheli, Sergio Mover, Marco Roveri, and Stefano Tonetta. The nuxmv symbolic model checker. In Armin Biere and Roderick Bloem, editors, *Computer Aided Verification*, pages 334–342, Cham, 2014. Springer International Publishing.

[40] Michael Leuschel and Michael Butler. Prob: A model checker for b. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods*, pages 855–874, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[41] Mathieu Clabaut, Ning Ge, Nicolas BRETON, Eric Jenn, Rémi Delmas, and Yoann Fonteneau. Industrial Grade Model Checking Use Cases, Constraints, Tools and

Applications. In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France, January 2016.

[42] A. Fantechi, W. Fokkink, and A. Morzenti. Some trends in formal methods applications to railway signaling. In *FMICS 2012*, 2012.

[43] Patrick Behm, Paul Benoit, Alain Faivre, and Jean-Marc Meynadier. *Météor: A Successful Application of B in a Large Project*, pages 369–387. Springer Berlin, 1999.

[44] Frédéric Badeau and Arnaud Amelot. Using B as a High Level Programming Language in an Industrial Project: Roissy VAL. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, pages 334–354, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[45] Didier Essamé and Daniel Dollé. B in Large-Scale Projects: The Canarsie Line CBTC Experience. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, pages 252–254, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[46] David Bonvoisin. 25 years of formal methods at RATP. Technical report, IRSC, 2016.

[47] P. Chapront and C. Galivel. Results of a safety software validation: Sacem. In J.-P. PERRIN, editor, *Control, Computers, Communications in Transportation*, IFAC Symposia Series, pages 91–98. Pergamon, Oxford, 1990.

[48] ClearSy. Atelier B User Manual Version 4.0. 2009.

[49] David Bonvoisin and Nazim Benaissa. Utilisation de la méthode de preuve formelle perf de la ratp sur le projet peee. In *Revue générale des chemins de fer*, 2015.

[50] Abderrahmane Feliachi, David Bonvoisin, Chaou Samira, and Julien Ordioni. Formal verification of system-level safety properties on railway software. 2016.

[51] Jean-Marc Mota, Evguenia Dmitrieva, Amel Mammar, Paul Caspi, Salimeh Behnia, Nicolas Breton, and Pascal Raymond. Safety demonstration for a rail signaling application in nominal and degraded modes using formal proof. In *FM 2014*, 2014.

[52] Bruno Tatibouët, Antoine Requet, Jean-Christophe Voisinet, and Ahmed Hammad. *Java Card Code Generation from B Specifications*, pages 306–318. Springer Berlin, 2003.

[53] Andrew C. Storey and Howard P. Haughton. *A strategy for the production of verifiable code using the B Method*, pages 346–365. Springer Berlin, 1994.

[54] Marie-Laure Potet and Yann Rouzaud. Composition and refinement in the b-method. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method, Second International B Conference, Montpellier, France, April 22-24, 1998, Proceedings*, volume 1393 of *Lecture Notes in Computer Science*, pages 46–65. Springer, 1998.

[55] Yann Rouzaud. Interpreting the b-method in the refinement calculus. In *Proceedings of the Wold Congress on Formal Methods in the Development of Computing Systems-Volume I - Volume I*, FM '99, page 411–430, Berlin, Heidelberg, 1999. Springer-Verlag.

[56] Brian Matthews, Brian Ritchie, and Juan Bicarregui. Synthesising structure from flat specifications. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, pages 148–161, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[57] Theodosis Dimitrakos, Juan Bicarregui, Brian Matthews, and T. S. E. Maibaum. Compositional structuring in the b-method: A logical viewpoint of the static context. In *Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, ZB '00, page 107–126, Berlin, Heidelberg, 2000. Springer-Verlag.

[58] Pierre Bontron and Marie-Laure Potet. Automatic construction of validated b components from structured developments. In *ZB 2000: Formal Specification and Development in Z and B*, pages 127–147, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[59] Edsger W. Dijkstra. *A Discipline of Programming.* Prentice-Hall, 1976.

[60] Dominique Cansell and Dominique Méry. Foundations of the b method. *Computers and Artificial Intelligence*, 22:221–256, 2003.

[61] Antoine Requet. Bart: A tool for automatic refinement. In Egon Börger, Michael Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, pages 345–345, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[62] Thierry Lecomte. Applying a formal method in industry: A 15-year trajectory. In María Alpuente, Byron Cook, and Christophe Joubert, editors, *Formal Methods for Industrial Critical Systems*, pages 26–34, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[63] Michael Leuschel and Michael Butler. Prob: An automated analysis toolset for the b method. *Int. J. Softw. Tools Technol. Transf.*, 10(2):185–203, February 2008.

[64] Michael Butler, Philipp Körner, Sebastian Krings, Thierry Lecomte, Michael Leuschel, Luis-Fernando Mejia, and Laurent Voisin. The first twenty-five years of industrial use of the b-method. In Maurice H. ter Beek and Dejan Ničković, editors, *Formal Methods for Industrial Critical Systems*, pages 189–209, Cham, 2020. Springer International Publishing.

[65] J. Abrial. Formal methods: Theory becoming practice. *J. Univers. Comput. Sci.*, 13:619–628, 2007.

[66] JCP Woodcock, Peter Larsen, Juan Bicarregui, and JS Fitzgerald. Formal methods: Practice and experience. *ACM Computing Surveys*, 41, 10 2009.

[67] Patrick Behm, Pierre Desforges, and Jean Marc Meynadier. Météor: An industrial success in formal development. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, pages 26–26, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[68] Jean-Louis Boulanger. *Formal Methods Applied to Industrial Complex Systems: Implementation of the B Method*. Wiley-IEEE Press, 1st edition, 2014.

[69] Manel Fredj, Sven Leger, Abderrahmane Feliachi, and Julien Ordioni. OVADO - enhancing data validation for safety-critical railway systems. In Alessandro Fantechi, Thierry Lecomte, and Alexander B. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Second International Conference, RSSRail 2017, Pistoia, Italy, November 14-16, 2017,*

Proceedings, volume 10598 of *Lecture Notes in Computer Science*, pages 87–98. Springer, 2017.

[70] Robert Abo and Laurent Voisin. Formal implementation of data validation for railway safety-related systems with ovado. In Steve Counsell and Manuel Núñez, editors, *Software Engineering and Formal Methods*, pages 221–236, Cham, 2014. Springer International Publishing.

[71] Dominik Hansen, David Schneider, and Michael Leuschel. Using b and prob for data validation projects. In *Proceedings of the 5th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z - Volume 9675*, ABZ 2016, page 167–182, Berlin, Heidelberg, 2016. Springer-Verlag.

[72] T. Lecomte, L. Burdy, and M. Leuschel. Formally checking large data sets in the railways. *ArXiv*, abs/1210.6815, 2012.

[73] Denis Sabatier. Using formal proof and b method at system level for industrial projects. In Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 20–31, Cham, 2016. Springer International Publishing.

[74] Mathieu Comptier, Michael Leuschel, Luis-Fernando Mejia, Julien Molinero Perez, and Mareike Mutz. Property-based modelling and validation of a cbtc zone controller in event-b. In Simon Collart-Dutilleul, Thierry Lecomte, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 202–212, Cham, 2019. Springer International Publishing.

[75] Mathieu Comptier, David Deharbe, Julien Molinero Perez, Louis Mussat, Thibaut Pierre, and Denis Sabatier. Safety analysis of a cbtc system: A rigorous approach with event-b. In Alessandro Fantechi, Thierry Lecomte, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 148–159, Cham, 2017. Springer International Publishing.

[76] A. Benveniste, P. Caspi, Stephen Edwards, Nicolas Halbwachs, P. Guernic, and Robert Simone. The synchronous languages 12 years later. *Proceedings of the IEEE*, 91:64 – 83, 02 2003.

[77] Julien Forget. *A Synchronous Language for Critical Embedded Systems with Multiple Real-Time Constraints*. Theses, Institut Supérieur de l'Aéronautique et de l'Espace, November 2009.

[78] Albert Benveniste, Paul Le Guernic, and Christian Jacquemot. Synchronous programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, 1991.

[79] Gérard Berry and Georges Gonthier. The esterel synchronous programming language: design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[80] Benjamin C. Pierce. *Types and Programming Languages*. The MIT Press, 1st edition, 2002.

[81] Nicolas Halbwachs, Fabienne Lagnier, and Pascal Raymond. Synchronous observers and the verification of reactive systems. In Maurice Nivat, Charles Rattray, Teodor Rus, and Giuseppe Scollo, editors, *Algebraic Methodology and Software Technology (AMAST'93)*, pages 83–96, London, 1994. Springer London.

[82] Nicolas Breton and Yoann Fonteneau. S3: Proving the safety of critical systems. In Thierry Lecomte, Ralf Pinger, and Alexander Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification*, pages 231–242, Cham, 2016. Springer International Publishing.

[83] Armin Biere, Alessandro Cimatti, Edmund Clarke, and Yunshan Zhu. Symbolic model checking without bdds. In W. Rance Cleaveland, editor, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 193–207, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[84] Martin Brain, Saurabh Joshi, Daniel Kroening, and Peter Schrammel. Safety verification and refutation by k-invariants and k-induction. In Sandrine Blazy and Thomas Jensen, editors, *Static Analysis*, pages 145–161, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[85] Mary Sheeran, Satnam Singh, and Gunnar Stålmarck. Checking safety properties using induction and a sat-solver. In Warren A. Hunt and Steven D. Johnson, editors, *Formal Methods in Computer-Aided Design*, pages 127–144, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg.

[86] Jack B. Dennis. *Petri Nets*, pages 1525–1530. Springer US, Boston, MA, 2011.

[87] Naïm Aber, B. Blanc, Nathalie Ferkane, Mohand Meziani, and Julien Ordioni. Rbs2hll - a formal modeling of relay-based interlocking. In *RSSRail*, 2019.

[88] Marielle Petit-Doche, Nicolas Breton, Roméo Courbis, Yoann Fonteneau, and Matthias Güdemann. Formal Verification of Industrial Critical Software. In Manuel Núñez and Matthias Güdemann, editors, *FMICS 2015*, pages 1–11. Springer International, 2015.

[89] Robert Walker Gunnar Smith. AUTOMATED VERIFICATION AND VALIDATION OF SIGNALING SYSTEMS IN PTC AND CBTC ENVIRONMENTS. Technical report, Prover Technology, 2017.

[90] Camille Parillaud, Yoann Fonteneau, and Fabien Belmonte. Interlocking formal verification at alstom signalling. In Simon Collart Dutilleul, Thierry Lecomte, and Alexander B. Romanovsky, editors, *Reliability, Safety, and Security of Railway Systems. Modelling, Analysis, Verification, and Certification - Third International Conference, RSSRail 2019, Lille, France, June 4-6, 2019, Proceedings*, volume 11495 of *Lecture Notes in Computer Science*, pages 215–225. Springer, 2019.

[91] Pierre Chartier. Formalisation of b in isabelle/hol. In *Proceedings of the Second International B Conference on Recent Advances in the Development and Use of the B Method*, pages 66–82, 1998.

[92] Jean Paul Bodeveix, Mamoun Filali, and César A. Muñoz. A formalization of the b-method in coq and pvs. *LNCS*, 1709:33–49, 1999.

[93] Éric Jaeger and Catherine Dubois. Why would you trust b? *CoRR*, abs/0902.3858, 2009.

[94] David Déharbe and Stephan Merz. Software component design with the b method – a formalization in isabelle/hol. In *Revised Selected Papers of the 12th International*

Conference on Formal Aspects of Component Software - Volume 9539, FACS 2015, page 31–47, Berlin, Heidelberg, 2015. Springer-Verlag.

[95] Jeremy E. Dawson. Formalising generalised substitutions. In Klaus Schneider and Jens Brandt, editors, *Theorem Proving in Higher Order Logics*, pages 54–69, Berlin, Heidelberg, 2007. Springer Berlin Heidelberg.

[96] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[97] Richard A. Kelsey. A correspondence between continuation passing style and static single assignment form. In *Papers from the 1995 ACM SIGPLAN Workshop on Intermediate Representations*, IR '95, page 13–22, New York, NY, USA, 1995. Association for Computing Machinery.

[98] Sebastian Pop. *The SSA Representation Framework: Semantics, Analyses and GCC Implementation*. Theses, École Nationale Supérieure des Mines de Paris, December 2006.

[99] Karl J. Ottenstein, Robert A. Ballance, and Arthur B. MacCabe. The program dependence web: A representation supporting control-, data-, and demand-driven interpretation of imperative languages. *SIGPLAN Not.*, 25(6):257–271, June 1990.

[100] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[101] Salimeh Behnia and Hélène Waeselynck. Test criteria definition for b models. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 509–528, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[102] Maulik Dave. Compiler verification: A bibliography. *SIGSOFT Softw. Eng. Notes*, 28:2–2, 11 2003.

[103] Adam Chlipala. A verified compiler for an impure functional language. volume 45, pages 93–106, 01 2010.

[104] Moussa Amrani, L. Lucio, Gehan M. K. Selim, Benoît Combemale, J. Dingel, H. Vangheluwe, Y. L. Traon, and J. Cordy. A tridimensional approach for studying the formal verification of model transformations. *2012 IEEE Fifth International Conference on Software Testing, Verification and Validation*, pages 921–928, 2012.

[105] John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman. *Introduction to Automata Theory, Languages, and Computation (3rd Edition)*. Addison-Wesley Longman Publishing Co., Inc., USA, 2006.

[106] Xavier Leroy. Formal certification of a compiler back-end or: Programming a compiler with a proof assistant. In *Conference Record of the 33rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '06, page 42–54, New York, NY, USA, 2006. Association for Computing Machinery.

[107] Xavier Leroy. Formal verification of a realistic compiler. *Commun. ACM*, 52(7):107–115, 2009.

[108] Gerwin Klein and Tobias Nipkow. A machine-checked model for a java-like language, virtual machine, and compiler. *ACM Trans. Program. Lang. Syst.*, 28(4):619–695, July 2006.

[109] Wolf Zimmermann. On the correctness of transformations in compiler back-ends. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods*, pages 74–95, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[110] Jianzhou Zhao, Santosh Nagarakatte, Milo M.K. Martin, and Steve Zdancewic. Formal verification of ssa-based optimizations for llvm. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 175–186, New York, NY, USA, 2013. Association for Computing Machinery.

[111] A. Ahmed. Verified compilers for a multi-language world. In *SNAPL*, 2015.

[112] Timothy Bourke, Lélio Brun, Pierre-Évariste Dagand, Xavier Leroy, Marc Pouzet, and Lionel Rieg. A Formally Verified Compiler for Lustre. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 586–601. ACM, 2017.

[113] Dariusz Biernacki, Jean louis Colaco, and Marc Pouzet. Clock-directed modular code generation from synchronous block diagrams. In *APGES*, 2007.

[114] George C. Necula. Proof-carrying code. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, page 106–119, New York, NY, USA, 1997. Association for Computing Machinery.

[115] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Proceedings of the 4th International Conference on Tools and Algorithms for Construction and Analysis of Systems*, TACAS '98, page 151–166, Berlin, Heidelberg, 1998. Springer-Verlag.

[116] George C. Necula. Translation validation for an optimizing compiler. *SIGPLAN Not.*, 35(5):83–94, May 2000.

[117] Lenore Zuck, Amir Pnueli, Yi Fang, and Benjamin Goldberg. Voc: A translation validator for optimizing compilers. *Electronic Notes in Theoretical Computer Science*, 65(2):2–18, 2002. COCV'02, Compiler Optimization Meets Compiler Verification (Satellite Event of ETAPS 2002).

[118] Jean-Baptiste Tristan and Xavier Leroy. Formal verification of translation validators: A case study on instruction scheduling optimizations. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, page 17–27, New York, NY, USA, 2008. Association for Computing Machinery.

[119] Van-Chan Ngo, Jean-Pierre Talpin, Thierry Gautier, Loïc Besnard, and Paul Le Guernic. Modular translation validation of a full-sized synchronous compiler using off-the-shelf verification tools. In *Proceedings of the 18th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '15, pages 109–112, New York, NY, USA, 2015. ACM.

[120] Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. Translation validation for a verified os kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, page 471–482, New York, NY, USA, 2013. Association for Computing Machinery.

[121] Michael Ryabtsev and Ofer Strichman. Translation Validation: From Simulink to C. In *Computer Aided Verification*, pages 696–701, 2009.

[122] Amir Pnueli, Ofer Shtrichman, and Michael Siegel. Translation validation: From signal to c. In *Correct System Design, Recent Insight and Advances*, pages 231–255. Springer-Verlag, 1999.

[123] Martin Strecker. Formal verification of a java compiler in isabelle. In Andrei Voronkov, editor, *Automated Deduction—CADE-18*, pages 63–77. Springer Berlin Heidelberg, 2002.

[124] Tobias Nipkow, David von Oheimb, and Cornelia Pusch. μJava: Embedding a programming language in a theorem prover. In *Foundations of Secure Computation, volume 175 of NATO Science Series F: Computer and Systems Sciences*, pages 117–144. IOS Press, 2000.

[125] Jan Olaf Blech and Arnd Poetzsch-Heffter. A certifying code generation phase. *Electron. Notes Theor. Comput. Sci.*, 190(4):65–82, November 2007.

[126] Loïc Besnard, Thierry Gautier, Matthieu Moy, Jean-Pierre Talpin, Kenneth Johnson, and Florence Maraninchi. Automatic translation of c/c++ parallel code into synchronous formalism using an ssa intermediate form. volume 23, 2009.

[127] Abdoulaye Gamati. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification.*

[128] Gilles Barthe, Delphine Demange, and David Pichardie. A formally verified ssa-based middle-end. In Helmut Seidl, editor, *Programming Languages and Systems*, pages 47–66, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[129] Sebastian Pop, Pierre Jouvelot, and George André Silber. In and Out of SSA : a Denotational Specification. In *Workshop Static Single-Assignment Form Seminar*, 2009.

[130] Jan Olaf Blech and Sabine Glesner. A Formal Correctness Proof for Code Generation from SSA Form in Isabelle/HOL. In *GI Jahrestagung*, 2004.

[131] Sigurd Schneider, Gert Smolka, and Sebastian Hack. A linear first-order functional intermediate language for verified compilers. In Christian Urban and Xingyuan Zhang, editors, *Interactive Theorem Proving*, pages 344–358, Cham, 2015. Springer International Publishing.

[132] Alexei Iliasov, Ilya Lopatkin, and Alexander Romanovsky. The safecap project on railway safety verification and capacity simulation. volume 8166, pages 125–132, 10 2013.

[133] Pengfei SUN. *Model based system engineering for safety of railway critical systems.* Theses, Ecole Centrale de Lille, July 2015.

[134] Steve A. Schneider and Helen Treharne. Csp theorems for communicating b machines. *Formal Aspects of Computing*, 17:390–422, 2005.

[135] Michael Butler. csp2b: A practical approach to combining csp and b. In Jeannette M. Wing, Jim Woodcock, and Jim Davies, editors, *FM'99 — Formal Methods*, pages 490–508, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[136] Colin Snook and Michael Butler. Uml-b: Formal modeling and design aided by uml. *ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, January 2006.

[137] Amel Mammar and Régine Laleau. From a B formal specification to an executable code: application to the relational database domain. *Info. & Soft. Technology 2006*, 48(4), 2006.

[138] Steve Jeffrey Tueno Fotso, Marc Frappier, Amel Mammar, and Régine Laleau. From sysml/kaos domain models to b system specifications. *ArXiv*, abs/1803.01972, 2018.

[139] Didier Bert, Sylvain Boulmé, Marie-Laure Potet, Antoine Requet, and Laurent Voisin. Adaptable Translator of B Specifications to Embedded C Programs. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003*, pages 94–113. Springer Berlin, 2003.

[140] J. C. Voisinet. Jbtools: An experimental platform for the formal b method. In *Proceedings of the Inaugural Conference on the Principles and Practice of Programming, 2002 and Proceedings of the Second Workshop on Intermediate Representation Engineering for Virtual Machines, 2002*, PPPJ '02/IRE '02, page 137–139, Maynooth, County Kildare, IRL, 2002. National University of Ireland.

[141] David Déharbe, Bruno Gomes, and Anamaria Moreira. Bsmart: A tool for the development of java card applications with the b method. In Egon Börger, Michael

Butler, Jonathan P. Bowen, and Paul Boca, editors, *Abstract State Machines, B and Z*, pages 351–352, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[142] Richard Bonichon, David Déharbe, Thierry Lecomte, and Valério Medeiros. *LLVM-Based Code Generation for B*, pages 1–16. Springer International, 2015.

[143] Dominique Méry and Neeraj Kumar Singh. Automatic Code Generation from Event-B Models. In *SoICT '11*, pages 179–188. ACM, 2011.

[144] Andreas Fürst, Thai Son Hoang, David Basin, Krishnaji Desai, Naoto Sato, and Kunihiko Miyazaki. Code Generation for Event-B. In Elvira Albert and Emil Sekerinski, editors, *IFM 2014*, pages 323–338. Springer International, 2014.

[145] A. Aljer, J. Boulanger, P. Devienne, G. Mariano, and S. Tison. Bhdl: Circuit design in b. In *2010 10th International Conference on Application of Concurrency to System Design*, page 241, Los Alamitos, CA, USA, jun 2003. IEEE Computer Society.

[146] Ning Ge, Arnaud Dieumegard, E. Jenn, and L. Voisin. Correct-by-construction specification to verified code. *Journal of Software: Evolution and Process*, 30, 2018.

[147] Néstor Cataño, Tim Wahls, Camilo Rueda, Víctor Rivera, and Danni Yu. Translating b machines to jml specifications. In *Proceedings of the 27th Annual ACM Symposium on Applied Computing*, SAC '12, pages 1271–1277. ACM, 2012.

[148] Víctor Rivera, Néstor Cataño, Tim Wahls, and Camilo Rueda. Code generation for event-b. *Int. J. Softw. Tools Technol. Transf.*, 19(1):31–52, February 2017.

[149] Fabian Vu, Dominik Hansen, Philipp Körner, and Michael Leuschel. A multi-target code generator for high-level b. In Wolfgang Ahrendt and Silvia Lizeth Tapia Tarifa, editors, *Integrated Formal Methods*, pages 456–473, Cham, 2019. Springer International Publishing.

[150] Pamela Zave and Michael Jackson. Four dark corners of requirements engineering. *ACM Trans. Softw. Eng. Methodol.*, 6(1):1–30, January 1997.

[151] ClearSy. B Language reference Manual 1.8.6. 2009.

[152] Patrick Behm, Lilian Burdy, and Jean Marc Meynadier. Well defined b. In Didier Bert, editor, *B'98: Recent Advances in the Development and Use of the B Method*, pages 29–45, Berlin, Heidelberg, 1998. Springer Berlin Heidelberg.

[153] Marc Brandis and Hanspeter Mössenböck. Single-pass generation of static single-assignment form for structured languages. *ACM Trans. Program. Lang. Syst.*, 16:1684–1698, 11 1994.

[154] Bjarne Steensgaard. Sparse functional stores for imperative programs. *SIGPLAN Not.*, 30(3):62–70, March 1995.

[155] Salimeh Behnia. *Test de modèles formels en B : cadre théorique et critères de couverture.* PhD thesis, 2000. Thèse de doctorat dirigée par Thévenod-Fosse, Pascale Informatique et télécommunications Toulouse, INPT 2000.

[156] Samuel Colin, Dorian Petit, Vincent Poirriez, Jerome Rocheteau, Rafael Marcano, and Georges Mariano. Brillant: An open source and xml-based platform for rigourous software development. In *Proceedings of the Third IEEE International Conference on Software Engineering and Formal Methods*, SEFM '05, page 373–382, USA, 2005. IEEE Computer Society.

[157] Yann Zimmermann and Diana Toma. Component reuse in b using acl2. In Helen Treharne, Steve King, Martin Henson, and Steve Schneider, editors, *ZB 2005: Formal Specification and Development in Z and B*, pages 279–298, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[158] Frédéric Badeau, Didier Bert, Sylvain Boulmé, Christophe Métayer, Marie-Laure Potet, Nicolas Stouls, and Laurent Voisin. Adaptabilité et validation de la traduction de B vers c. points de vue et résultats du projet BOM. *Technique et Science Informatiques*, 23(7):879–903, 2004.

[159] K. Schneider. Embedding imperative synchronous languages in interactive theorem provers. In *Proceedings Second International Conference on Application of Concurrency to System Design*, pages 143–154, 2001.

[160] Zhibin Yang, Jean-Paul Bodeveix, and Mamoun Filali. A comparative study of two formal semantics of the SIGNAL language. *Frontiers of Computer Science*, 7(5):673–693, Oct 2013.

[161] Davide Sangiorgi. On the bisimulation proof method. *Mathematical Structures in Computer Science*, 8(5):447–479, 1998.

[162] Anas CHARAFI. Development of a tool for the translation of b models to hll. Technical report, National Polytechnic Institute of Toulouse, 2017. Master project report.

[163] Valentin Paris. Conception document. Technical report, Ecole 42, 2019. Master project report.

[164] J. . Boulanger. Abtools: another b tool. In *Third International Conference on Application of Concurrency to System Design, 2003. Proceedings.*, pages 231–232, 2003.

[165] ClearSy. B Compiler General Conception. 2009.

[166] Darren Cofer and Steven Miller. Do-333 certification case studies. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods*, pages 1–15, Cham, 2014. Springer International Publishing.

[167] IEEE Standard for Communications-Based Train Control (CBTC) Performance and Functional Requirements. *IEEE Std 1474.1-1999*, 1999.

[168] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Refinement-animation for event-b — towards a method of validation. In Marc Frappier, Uwe Glässer, Sarfraz Khurshid, Régine Laleau, and Steve Reeves, editors, *Abstract State Machines, Alloy, B and Z*, pages 287–301, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[169] Atif Mashkoor, Jean-Pierre Jacquot, and Jeanine Souquières. Transformation Heuristics for Formal Requirements Validation by Animation. In *2nd International Workshop on the Certification of Safety-Critical Software Controlled Systems - SafeCert 2009*, York, United Kingdom, March 2009.

[170] S. Berghofer and T. Nipkow. Random testing in isabelle/hol. In *Proceedings of the Second International Conference on Software Engineering and Formal Methods*, pages 230–239, Sept 2004.

[171] F. Yang, J. Jacquot, and J. Souquières. JeB: Safe Simulation of Event-B Models in JavaScript. In *2013 20th Asia-Pacific Software Engineering Conference (APSEC)*, volume 1, pages 571–576, 2013.

[172] Thierry Servat. BRAMA: A New Graphic Animation Tool for B Models. In Jacques Julliand and Olga Kouchnarenko, editors, *B 2007: Formal Specification and Development in B*, pages 274–276. Springer Berlin Heidelberg, 2006.

[173] Dominique Méry and Neeraj Kumar Singh. Real-time animation for formal specification. In Marc Aiguier, Francis Bretaudeau, and Daniel Krob, editors, *Complex Systems Design & Management*, pages 49–60. Springer Berlin Heidelberg, 2010.

[174] Stefan Hallerstede, Michael Leuschel, and Daniel Plagge. Validation of formal models by refinement animation. *Science of Computer Programming - SCP*, 78, 03 2013.

[175] Atif Mashkoor and Jean-Pierre Jacquot. Validation of formal specifications through transformation and animation. *Requirements Engineering*, 22(4):433–451, Nov 2017.

[176] Aaron M. Dutle, César A. Muñoz, Anthony J. Narkawicz, and Ricky W. Butler. Software validation via model animation. In Jasmin Christian Blanchette and Nikolai Kosmatov, editors, *Tests and Proofs*, pages 92–108. Springer International Publishing, 2015.