A Comprehensive Study of Declarative Modelling Languages

B, Event-B, Alloy, Dash, TLA$^+$, PlusCal, AsmetaL

by

AMIN BANDALI

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2020

Abstract

Declarative behavioural modelling is a powerful modelling paradigm that enables users to model system functionality abstractly and formally. An abstract model is a concise and compact representation of key characteristics of a system, and enables the stakeholders to reason about the correctness of the system in the early stages of development.

There are many different declarative languages and they have greatly varying constructs for representing a transition system, and they sometimes differ in rather subtle ways. In this thesis, we compare seven formal declarative modelling languages B, EVENT-B, ALLOY, DASH, TLA$^+$, PLUSCAL, and ASMETAL on several criteria. We classify these criteria under three main categories: structuring transition systems (control modelling), data descriptions in transition systems (data modelling), and modularity aspects of modelling. We developed this comparison by completing a set of case studies across the data- vs. control-oriented spectrum in all of the above languages.

Structurally, a transition system is comprised of a snapshot declaration and snapshot space, initialization, and a transition relation potentially composed of individual transitions. We meticulously outline the differences between the languages with respect to how the modeller would express each of the above components of a transition system in each language, and include discussions regarding stuttering and inconsistencies in the transition relation. Data-related aspects of a formal model include use of basic and composite datatypes, well-formedness and typechecking, and separation of name spaces with respect to global and local variables. Modularity criteria includes subtransition systems and data decomposition. We employ a series of small and concise exemplars we have devised to highlight these differences in each language. To help modellers answer the important question of which declarative modelling language may be most suited for modelling their system, we present recommendations based on our observations about the differentiating characteristics of each of these languages.

Acknowledgements

I would like to thank all the little people who made this thesis possible. (TBD)

Dedication

TBD

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

Architects draw detailed plans before a brick is laid or a nail is hammered. Programmers and software engineers do not. Can this be why houses seldom collapse and programs often crash?
— Leslie Lamport, Turing Award Winner, 2013

Distinguished Computer Scientist Leslie Lamport explains that blueprints help architects make sure what they are planning to build will work. Furthermore, "working" means more than merely not collapsing, but rather serving the intended purpose. Engineers and architects use blueprints as a common language between themselves and their clients, and use it to precisely plan out ahead of time the structure that they will build. However, programmers and software engineers rarely even sketch out what their programs are intended to do, before starting to write the code. Even the simplest and most commonly taught algorithms such as binary sort sometimes require careful thinking about their intricate details to ensure their correctness, let alone complex systems with dozens or more subsystems and parts intended to work together.

When designing complex systems, Lamport argues, the need for formal specifications should be as obvious as the need for blueprints when designing a skyscraper. However, few programmers even know of existence of specification languages and their supporting tools, much less how to use and write specifications for the systems they design.

There are a number of reasons why formal specifications are important and useful. Most mainstream programming languages currently commonly used by programmers, such as C, C++, or Java, usually require programmers to be deep into the implementation details, imperatively specifying exactly how each task should be carried out. It is often challenging to keep in mind the big picture of what the parts are supposed to do individually and as a group, while getting bogged down with other less relevant details. Having formal specifications in one's toolbox enables abstracting away from irrelevant details, focusing only on the heart of the matter and making sure it is well understood, and that the proposed solution correctly fits and serves the required purpose.

1.1 Declarative Modelling Languages

The languages studied in this thesis are declarative modelling languages for behavioural formal specification using a state-machine-oriented/transition system approach. These languages in essence model a Kripke structure underneath. Declarative behavioural modelling is a powerful technique for modelling systems in a concise way, free of design or implementation details. Declarative modelling languages allow describing systems in a higher level of abstraction than permitted by typical mainstream programming languages like C or Java.

Models written using declarative behavioural modelling languages, such as Z [73], ALLOY [43], and TLA$^+$ [47], have the following general characteristics [12]:

1. they describe the transitions in a declarative manner using constraints, rather than through imperative calculations and/or statements;
2. they include user-defined and -axiomatized units of data, which can represent rich datatypes such as lists and trees;
3. they have a formal mathematical and logical foundation, usually first-order logic (FOL) and/or set theory; and
4. they allow writing models without specifying the size of sets (the scopes); the scopes may need to be specified for analysis.

We are motivated to do this study by the many applications and demonstrated usefulness of declarative modelling languages and model checking to help design systems or analyze and verify properties about the design of existing systems. Examples include Zave's use of ALLOY and Spin to find specification-level bugs in the specification of the Chord network protocol [77], Newcombe's report on the use of TLA$^+$ by engineers at Amazon which has helped find subtle bugs in complex real-world systems and prevent the bugs from reaching production [61], and the use of B by Huynh *et al.* for formalizing a new healthcare access control model with conflict resolution for managing patients' consent as to who can access their Electronic Health Records (EHR), while taking into account the regional laws and regulations of Québec and Canada that allow overriding patient consent regarding access to their EHR under certain strictly defined scenarios to protect the patient's life [41].

Declarative models allow modellers to sketch out and reason about systems and how they change over time in an abstract and declarative way, without having to worry about irrelevant details. An important question, given that there are a great many number of languages to choose from is how does one make a choice of which language to use? This thesis presents a comparison criteria and compares a number of popular declarative languages, highlighting the control and data modelling aspects, as well as the modularity of models in each language. Comparisons between modelling languages are useful to provide a means of discerning which modelling language is most suitable for modelling a system. Models in these languages may vary in length and organization, and by doing a number of case studies in multiple languages helps us better expose the differences in paradigms, conventions, and structures among the languages. Common constructs in the languages often vary in subtle ways, and through our case studies, we tease these differences apart for each language. The equivalence of our models is by observations, using a model checker and potentially other available tool support for each language to verify a series of properties about the models across all of the selected languages.

In comparison to other works comparing declarative modelling languages [61, 77, 34, 21], we focus on models of transition systems. We develop a set of categorized comparison criteria and examine in depth each language with respect to each of the criterion. Further, we use these criteria to compare a diverse range of examples on the data- vs. control-oriented characterization spectrum, modelling each of the five examples in all of the seven languages, producing a total of thirty-five models. We use these models and our observations from carrying out the case studies to make recommendations as to which language(s) we think would be the best fit for modelling various kinds of transition systems.

1.2 Selection of Languages

We selected the seven popular, declarative modelling languages B, EVENT-B, ALLOY, DASH, TLA$^+$, PLUSCAL, and ASMETAL for comparison. We chose languages that have tool support for model checking. However, we are not comparing the languages on their tool support, because tool support and the analysis performance of those tools are subject to change, and can change much more easily than the logic and semantics of the language. This point of view is also shared by Lamport, with TLA$^+$ existing for several years before the TLC model checker and the rest of the TLA$^+$ Toolbox were created for it.

We did not include other formal specification languages for various reasons. For instance, we omitted the Z [73] and VDM [45] specification languages, due to the lack of model checking tool support for them, and formalisms based on process algebras and Petri Nets, as they are at a different (often lower) level of abstraction compared to declarative modelling languages.

Statecharts-based languages and UML state machines [9] provide a graphical manner to describe system behaviour, but do not completely fall into the category of declarative specification languages: their semantics are often not fully formal and they lack support for declaring datatypes. OCL [8] is a declarative language that can be used in combination with UML to constrain the pre and postconditions of transitions.

The languages of model checkers such a SMV [60] and Spin [40] are lower-level descriptions than what is often convenient for a user. nuXmv [27] is a re-implementation and extension of SMV that adds support for verification of infinite datatypes, such as integers and reals, and incorporates a verification engine with state-of-the-art SAT-based algorithms. However, they all have limited support for user-declared datatypes. Furthermore, in SMV/nuXmv the scope of datatypes and relations must be set at modelling time, which forces users to modify their models every time they want to analyze and check properties in at larger scope.

1.3 Thesis Contributions

The contributions of this thesis are

- a set of criteria to compare declarative modelling languages;
- the comparison of the selected declarative modelling languages (B, EVENT-B, ALLOY, DASH, TLA$^+$, PLUSCAL, and ASMETAL) based on these criteria; and
- our recommendations for the choice of modelling language based on the characteristics of the transition system under description, rooted in our observations of the differences and similarities between the languages with respect to our comparison criteria from the several case studies we carried out.

The overview of the methodology for comparing the selected languages for this thesis is to do a number of case studies in each language, taking notes on their characteristics and differences while doing so. Each of the chosen case studies is a model previously done in one (or possibly more) of the selected languages. The choice of the order of languages for each case study was made randomly to address any potential concerns for bias in that regard.

1.4 Thesis Outline

Chapter 2 provides background on the seven declarative modelling languages used in this work. Chapter 3 describes our methodology for carrying out the research and modelling of the case studies across the languages, as well as addressing potential threats to validity. In Chapter 4, we discuss the control modelling aspects in each language, meaning the structuring of transition systems in terms of a individual transitions, in Chapter 5 we consider the data modelling aspects of each language and the structuring of transition systems in terms of their data descriptions, and in Chapter 6 we investigate the constructs relating to modularity of models in the languages and the structuring of transition systems on a larger scale in terms of files as well as subtransition systems. Chapter 7 provides an overview of our case studies, and how our comparison criteria highlighted interesting and/or different features and characteristics of each language while modelling each case study. Finally, Chapter 8 presents related work, and Chapter 9 provides concluding remarks, including recommendations for the choice of declarative modelling language based on the characteristics of the transition system under description.

Chapter 2

Background

In this chapter, we provide background on the seven declarative modelling languages we selected to study and carry out our case studies in this work. For each language, we give a brief background about its history and origins, and its logic. Detailed information about the language will be covered in the comparisons of the following chapters. We also report on the tools we used to support our modelling efforts. Information about the version of each tool we used is available in Appendix A.

As we will be frequently referring to languages by name in the next chapters of this thesis, to make the language names easier to spot and discussions easier to follow, we adopt the language name stylizations shown in Table 2.1. Further, the keywords in each of the languages are colour-coded, to help make distinguishing between the code snippets across the languages easier in later chapters.

2.1 B

B [14, 29] is a formal method originally developed by Jean-Raymond Abrial in the 1980s as a successor to Z, and using an Abstract Machine Notation (AMN) for specification of systems. B is used in the development of correct by construction software, with tool support for specification, design, and verification (animating, model checking, and theorem proving) of software systems.

The logic of B is rooted in first-order logic and set theory. Sets are created using either set comprehension, or set operations (such as Cartesian product, intersection, power set, *etc.*) and can be used for updating a variable's value to create a transition system. Predicates can be axiomatized or defined using propositional logic operations and set predicates. Functions can be declared both explicitly using

Table 2.1: Language name stylizations and colours

Original	**Stylized**
B	B
Event-B	EVENT-B
Alloy	ALLOY
Dash	DASH
TLA$^+$	TLA$^+$
PlusCal	PLUSCAL
AsmetaL	ASMETAL

function types (partial, total, surjective, *etc.*), or implicitly by restricting relations. B supports *refinement* of machines, allowing the modeller to start at a high level of abstraction, and gradually refine their model to more concrete ones.

Given B's longevity, various tooling software has been developed over the years to support the B method and development of B models. These include the B-Toolkit originally developed by B-Core and now available as free software [7] under the 2-clause BSD license [6], the proprietary Atelier B industrial tool developed by ClearSy, and the ProB animator and model checker [55] released as free software under the EPL v1.0 license [4]. Of these tools, we used the ProB tool for writing and checking our B models.

2.2 Event-B

EVENT-B [15] is a simplification and extension of the B method. While B is largely used for specification and verification of software systems, EVENT-B was designed to enable the modelling of complete systems (software, hardware, as well as the surrounding environment), merging the gap between specification of the software and the rest of the system. EVENT-B is a successor of B, and its logic is very similar to that of B and is rooted in first-order logic and set theory.

Although EVENT-B is a successor of B and its syntax is a simplification and extension of that of B, where a B model is simply a plaintext file with its content directly matching the B syntax, an EVENT-B model is a series of multiple complex XML files. This means B models can easily be read and written using any text editor, but reading and writing EVENT-B models effectively requires a special-purpose text editor for parsing and modifying the XML files without getting in the way of the modeller. At the time of writing this work, the only available tool support for EVENT-B is the Rodin Platform [16, 44], an eclipse-based IDE released as free software under the EPL v1.0 license [5]. The Rodin Platform is comprised of a set of plugins, including a text editor for creating and editing EVENT-B models, plugins for integration with the ProB animator and model checker, a LaTeX exporter plugin for generating typeset documents from EVENT-B models, and more. Even though the Rodin Platform's model editor is capable of editing EVENT-B models, the user has to point and click or use several keyboard shortcuts to make various parts of the model description editable. We believe that this, along with the fact that the text editor is barely customizable, makes Rodin less suitable for writing larger models.

Lastly, the current latest release of the Rodin Platform as of the time of this writing is over two years old, and is based on a version of Eclipse that suffers from a known bug that causes it to crash immediately when used with any Java version newer than Java 8. Considering that many GNU/Linux distributions have been dropping this old release of Java from their repositories, and as time goes by more distributions continue to do so, this effectively means that the Rodin Platform cannot be used on newer machines and operating systems.

For the EVENT-B models of our case studies, we used the Rodin Platform, as well as the ProB plugin for Rodin, which supports exporting EVENT-B models from the Rodin Platform for analysis using the standalone ProB tool.

2.3 Alloy

ALLOY [42, 43] is a declarative modelling language designed for exploring and describing structures and their properties. ALLOY's logic is a relational logic with set theory, that is both powerful enough to express complex structures and constraints on them while allowing fully automated analysis of models written in the language.

ALLOY is primarily supported by the Alloy Analyzer [43] tool, a finite model finder for analyzing ALLOY models by finding satisfying instances for predicates or counterexamples to assertions in finite scopes. The Alloy Analyzer comes with a visualizer and evaluator, which are invaluable for visualizing generated instances and evaluating ALLOY expressions when writing or debugging ALLOY models. Properties to be checked are written in ALLOY itself as predicates or assertions along with the main specification of the model. The Alloy Analyzer through Kodkod [75] integrates with multiple SAT solvers, allowing the modeller to easily switch solvers and choose the one that performs the best for their use-case.

Besides the Alloy Analyzer, a variety of other tools have been developed by the Alloy community. These include Astra [13] for Alloy to SMT-LIB translation, as an alternative to Kodkod; and ALDB [1], a command-line tool for debugging transition system models written in ALLOY, which allows the modeller to step through the transitions of a transition system model, similar to stepping through the lines of code of a program using a debugger for a programming language.

2.4 Dash

DASH [69, 70] is a new modelling language for writing declarative behavioural models, combining the logic of ALLOY with common control-oriented modelling constructs of labelled control state hierarchy and named events, as introduced by Harel [38]. DASH is built as an extension to ALLOY, with its language being a superset of the ALLOY language. DASH provides syntactic constructs for specifying and factoring transitions. Transitions can be factored by states, like in Statecharts, or by events and/or conditions; making DASH a flexible language capable of accommodating different modelling paradigms. DASH implements transition comprehensions, enabling the description of a group of transitions using a single statement. DASH follows the usual semantics of Statecharts: transitions from states higher in the hierarchy have priority over those lower in the hierarchy, and concurrent states can each take one transition in response to an environmental input forming big steps (consisting of multiple transitions). Properties to be checked are written along with the main specification of the model either in the form of ALLOY expressions in `escape` blocks, or in a small domain-specific subset of DASH, based on the underlying CTL module used by DASH's tool support for model checking.

Tooling around DASH is built using Xtext [10], and includes a compiler for translating DASH to ALLOY, allowing the modeller to benefit from use of the Alloy Analyzer for model checking [68] and model finding like regular ALLOY models. A text editor with syntax highlighting for DASH is available as an Eclipse plugin. Also, the DASH website at `http://dash.uwaterloo.ca:8080/dash/` includes an online editor for writing DASH models and translating them to ALLOY in the browser.

2.5 TLA$^+$

TLA$^+$ is a formal specification language developed by Leslie Lamport, based on the idea that using simple mathematics is the best way to write formal descriptions; and that a specification language should provide the bare minimum required for writing simple mathematics to describe systems precisely. TLA$^+$ has first-order logic with an untyped classical set theory as its modelling language, and was originally designed for writing high-level specifications of reactive, distributed, and asynchronous systems. LTL properties to be checked are written along with the main specification of the model using the full TLA$^+$ language, including several temporal operators such as $\Diamond$ and $\Box$ for expressing temporal properties.

Tool support for TLA$^+$ is mainly the TLA$^+$ Toolbox, an Eclipse-based IDE tailored for writing and working with TLA$^+$ specifications. The TLA$^+$ Toolbox includes the SANY parser and semantic analyzer for TLA$^+$ [47], the TLC model checker [76, 47], the PLUSCAL algorithm language [49], the TLATEX pretty printer [47], and the TLAPS (TLA$^+$ Proof System) [28]. TLA$^+$ supports model refinement.

For the TLA$^+$ models of our case studies in this work, we used the TLA$^+$ Toolbox and the TLC model checker, which is already set to use multiple worker threads out of the box. For model checking very large models, TLC can run on a cluster of compute nodes.

2.6 PlusCal

PLUSCAL [49] is a formal specification language created by Leslie Lamport fifteen years after TLA$^+$ for describing and reasoning about algorithms, as an alternative to traditional informal pseudocode. PLUSCAL has two separate syntaxes: a C-syntax [50] similar to the C family of programming languages, and a more verbose but clearer P-syntax [51] which we have opted to use in this work. The verbosity of the P-syntax makes the meaning of the code clearer. While each of PLUSCAL's syntaxes resemble that of an imperative programming language, semantically PLUSCAL is more expressive than a programming language, since any mathematical formula that can be represented in TLA$^+$ may be used as a PLUSCAL expression. PLUSCAL models are translated into TLA$^+$, and may then be verified using the TLC model checker and the other TLA$^+$ tools. LTL properties to be checked are written using the full TLA$^+$ language, including several temporal operators such as $\Diamond$ and $\Box$ for expressing temporal properties.

For our PLUSCAL models, we used the TLA$^+$ Toolbox, which as described above, has a plugin for translating PLUSCAL to TLA$^+$. The PLUSCAL syntax is embedded in TLA$^+$ as a special block comment in a TLA$^+$ module.

2.7 AsmetaL

ASMETAL [67, 35, 36] (Asmeta Language) is a modelling language developed by Gargantini *et al.* as part of the Asmeta framework, based on the Abstract State Machines (ASMs) [25, 37] formal method. The logic of ASMETAL is first-order logic, with added semantics for snapshot variable updates and transition definitions, as formalized in [25]. Properties to be checked are written along with the main specification of the model using ASMETAL expressions, as well as additional LTL and CTL temporal operators available from libraries distributed with the model checker. Asmeta is a framework comprised of a variety of tools to aid with validation and verification of ASM models. At the time of writing this work, Asmeta is an active research project consisting of an extensive collection of tools for verification and validation of ASMs, released as free software [7] under the GPLv2+ license [2, 3].

The most relevant of these tools for our comparison are the **Asmee** editor for ASMETAL, the **AsmetaLc** compiler and parser for ASMETAL, the **AsmetaS** simulator, the **AsmetaA** animator, and the **AsmetaSMV** model checker based on NuSMV. As our work is centred around a comparison of modelling languages with model checking tool support, we were hoping to make extensive use of the various Asmeta tools, especially the AsmetaSMV model checker, which supports model checking both LTL and CTL properties. Unfortunately, we found that AsmetaSMV currently only supports a limited subset of the ASMETAL language, and we ran into these limitations with our models. Further, we learned that the AsmetaA animator and AsmetaS simulator suffer from similar limitations. For our case studies, since our goal is to compare the modelling languages and not their available tool support, we opted to use the full capabilities of the ASMETAL language, which is not yet fully supported by the tooling as of today. As such, for verifying our ASMETAL models we rely on the parser and typechecker for ASMETAL, and additionally the animator and simulator when possible.

Chapter 3

Methodology

In this chapter we describe the methodology used for carrying out the research and modelling of the case studies across the languages. We describe the principles put into designing our methodology, including how we devised the comparison criteria, and the measures we took to avoid bias in the work as much as possible.

To compare declarative modelling languages for writing models of transition systems, we began by selecting six relatively small examples and modelled them in three declarative modelling languages, with help from my colleagues and now former students Ali Abbassi and Jose Serna. We collaborated while writing the models, answering each other's questions and providing clarifications about any aspect of the models. These examples are those *without* an asterisk in Figure 3.1 of the data- vs. control-oriented characterization spectrum. A model is more control-oriented if it has complex conditions for when a transition is relevant that are naturally expressed using modes, control states, or concurrency. A model is more data-oriented if it has complex constructions of data. Based on our experience modelling these examples, we described the differences and similarities we observed across the languages, forming an initial set of comparison criteria to compare the languages against; and published our results [12]. These criteria included datatypes and typechecking, expressions, constructs for specifying the structure of transition systems and their semantics, and scalability of models.

Next, we expanded on our initial set of comparison criteria to include other interesting characteristics of declarative modelling languages that we did not previously consider. Further, we expanded our set of chosen declarative modelling languages from the initial three languages (B, DASH, and TLA$^+$) to include ASMETAL, ALLOY, EVENT-B, and PLUSCAL as well. We then chose a diverse set of the three most interesting examples from our initial set of examples across the characterization spectrum,

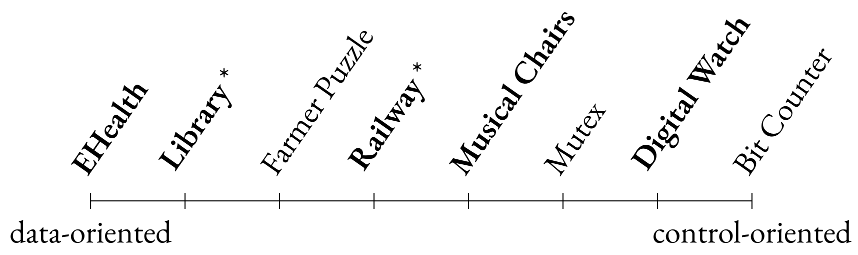

Figure 3.1: Our case studies, across the data- vs. control-oriented characterization spectrum

and modelled them in the newly added languages as well. This helped us make sure we correctly set up the tool support available for each of the languages.

Having made sure we are set up to use the supporting tools available for each language by modelling our previous examples in the newly added languages, we expanded our set of examples with two new larger systems. These two are the Library and Railway systems, marked with an asterisk in Figure 3.1. Collectively, we increased our number of models from eighteen (six models in three languages) small models to thirty-five (five models in seven languages) including larger examples. Our final case studies are those typeset in **bold** in Figure 3.1, presented in this thesis. Table 7.1 in Chapter 7 shows the sizes of the models in the common measuring unit Lines of Code, highlighting our use of larger and realistic examples for our case studies in this work.

Table 3.1 shows the case studies and languages considered in this work. For each case study (row), the number in each column indicates the order in which the model was completed in that language. We tried to make the table such that most columns include either a 1 or an E, meaning that at least one of the case studies was first modelled by us in that language, or originated in that language. We then used that version as the reference model for the next models of that case study ported to the other languages. Our first three models of EHealth, digital watch, and musical chairs were completed concurrently, and thus are all labelled '1' in the table. For the case studies where no reference model was available, we used the informal or semi-formal description of the system from the originating paper to write a first model, and used that model as our reference for the subsequent models of the case study. To avoid any bias, we assigned a random order to each model of each case study in each of the languages. We carried out the case studies top to bottom, starting with the EHealth system and ending with Railway, learning new languages as needed. While doing so, we observed and took notes about various characteristics of the modelling languages according to our comparison criteria, focusing on those used in the current model, and how we believed they affected the modelling process. With the help of these notes, we later developed isolated exemplars, which are small examples meant to demonstrate the differences between languages with respect to a particular criterion. If a model needed updating (*e.g.* addition of new properties, new criteria, new insights, *etc.*), we iterated through the versions of the model in all languages, updating those accordingly as well. Through this process we further expanded our set of comparison criteria.

Table 3.1: Order of modelling case studies across languages

Case study \ Language	B	EVENT-B	ALLOY	DASH	TLA$^+$	PLUSCAL	ASMETAL
EHealth [64]	1	2	3	1	1	4	5
Musical Chairs [62]	1	4	E [32]	1	1	3	2
Digital Watch [38]	1	2	5	1	1	4	3
Library [33, 34]	E [34]	1	E [34]	3	5	2	4
Railway [59]	1	7	3	5	6	4	2

Legend: E indicates Existing models, *i.e.* those that we had no influence on. The numbers in each row indicate the order of languages the case study was done in.

Our methodology was designed to limit bias towards one language or another when drawing our conclusions regarding language comparisons and recommendations based on them. For threats to validity, a possible threat to *internal validity* is that not all languages were the source/reference language in which a case study was modelled in. To partially alleviate this, we made sure that at least most of chosen languages were the source/reference for at least one case study. To further minimize threats to internal validity, we randomized the order in which we carried out each case study across the languages. A possible threat to *external validity* is that we may be missing comparison criteria that could have risen have we had done more case studies. Further, we only have five examples as part of our case studies. To partially remedy this, we made sure to use a diverse set of examples across the data- vs. control-oriented characterization spectrum, and of different sizes. A possible threat to *construct validity* is that our recommendations for each case study may be biased by our previous knowledge of some language, as we are more familiar with some languages more than others. To alleviate this bias, we have included multiple new languages and new case studies, that we were not previously familiar with. Threats to *conclusion validity* are those concerning the validity of our conclusions and recommendations about the choice of modelling language. To address these, we devised an extensive set of comparison criteria to compare the languages against while carrying out the case studies, and used a diverse set of examples for our case studies based on which we make our recommendations.

Chapter 4

Control Modelling

In this chapter, we discuss in depth the control aspects and the structure of transition systems as modelled in each of our selected modelling languages. Each section starts with a brief summary of the characteristics of the languages with respect to the section's comparison criterion, followed by detailed discussion of the criterion across the languages. Table 4.1 summarizes the differences in the structure of transition systems as modelled in each of our selected declarative modelling languages in alphabetical order.

As a reminder, the main concern of this thesis is comparison of the selected modelling languages for modelling transition systems, even though some of these languages may be used for more general-purpose modelling. Namely, the PLUSCAL algorithm language is a language designed for formal specification of algorithms and a counterpoint to pseudo-code, and ALLOY is a declarative specification language capable of modelling complex structural and behavioural constraints of software systems.

4.0 Terminology

As choice of terminology varies from one language to the next, we first define some standard terminology that we will use throughout this chapter when comparing the seven languages. We will use the words *implicit* and *explicit* carefully when referring to language constructs in this chapter: we say a construct is *explicit* in a language if the language has a textual representation corresponding to that construct; and otherwise we say it is *implicit*, which may potentially be built/calculated by the tool support behind the scenes.

Table 4.1: Structuring Transition Systems - Summary

Language / Criteria	B	EVENT-B	ALLOY	DASH	TLA$^+$	PLUSCAL	ASMETAL
Snapshot variables	`VARIABLES`	`VARIABLES`	varies	in `state`	`VARIABLES`	`variables`	`signature`
Initialization	`INITIALISATION`	`INITIALISATION`	`init`	`init`	`Init`	`variables`	`default init`
TR representation	I	I	E	I	ME	I	ME
Control state hierarchy	—	—	—	✓	—	—	—
Deadlock	P	P	P	P	P	P	P
Contradictory *TR*	Ind	Ind	P	Ind	P	Ind	P
Contradictory TP	P	P	P	P	P	P	P
Stuttering	E	E	E	I(m) + E	I(a) + E	I(a) + E	I(m) + E
Frame problem	unchanged	unchanged	may change	`env`	`UNCHANGED`	unchanged	`monitored`

Legend:
TR: Transition Relation. **TP**: Transition Postcondition. **E**: Explicit. **ME**: Mostly Explicit. **I**: Implicit. **P**: Possible. **NP**: Not Possible. **I(a)**: Implicit with all variables unchanged. **I(m)**: Implicit with monitored variables unchanged. **Ind**: Indirectly possible: when no other transition is ever enabled, contradictory TP results in a contradictory *TR*.

- **Snapshot**: is a mapping of variables to values.
- **Snapshot space**: is the set of all possible snapshots of a transition system, *i.e.* the cross product of variable values.
- **Transition System**: a transition system TS is a tuple (S, TR, I) where S is a set of snapshots, $TR \subseteq S \times S$ a transition relation, and $I \subseteq S$ a set of initial snapshots. A model in a declarative modelling language defines a transition system that starts in an initial snapshot $s_0 \in I$ and progresses from a snapshot s to the next snapshot s' for $(s, s') \in TR$.
- **Labelled control state**: is a distinguished set of variables with a finite set of values, that are used to control when a transition can be taken. Languages with labelled control states can have **control state hierarchy** and concurrency.
- **Transition**: a transition relation may be composed of a set of transitions, each $T \subseteq TR \subseteq S \times S$ with potentially multiple (s, s') mappings from source snapshot s to destination snapshot s'.
- **Step**: is a pair (s, s') of snapshots where $(s, s') \in TR$.
- **Monitored variable**: also referred to as an environmental variable, is one that can be only observed by the model but not changed by it — they are only changed by the environment. Monitored

variables are commonly used for modelling environmental phenomena such as temperature as obtained from a sensor. Conversely, a **controlled variable** (a non-environmental variable) is one that may be both observed and changed by the model.

- **Frame problem**: refers to the issue of how snapshot variables that are not explicitly changed in a transition may or may not change from one snapshot to the next. The frame problem is particularly an issue in declarative languages that rely on logical constraints on variables for describing the changed and unchanged variables in a transition. Since in all studied languages only one transition is taken per step, the frame problem can be discussed at the transition level (Section 4.4). If in a language more than one transition may be taken in a step, the frame problem would need to be discussed in the context of the transition relation as well.

- **Stuttering step**: is often used to allow a change in the external environment. The exact semantics of stuttering steps — when they might occur, and which variables they allow to change and which ones they keep unchanged — may differ between two declarative modelling languages, and from one model to the next. A declarative modelling language may have an implicit notion of stuttering, in which case it will also have accompanying semantics as to when stuttering steps may occur: whether stuttering steps have a lower 'priority' and may occur only when no other transitions are enabled, or if they have the same priority as other transitions and may occur even when one or more other transitions are enabled. In a language with implicit stuttering, if it differentiates between monitored and controlled variables then usually its stuttering steps will allow monitored variables to change while keeping controlled variables unchanged, otherwise its stuttering steps will keep all variables unchanged (each stuttering step is a self-loop from a snapshot back onto itself). In languages without implicit stuttering, a stuttering step must be modelled using an explicit transition definition, and the modeller may decide which variables may change and which are kept unchanged. Explicitly-added stuttering steps (as transition definitions) always have the same priority as other transitions. Adding stuttering steps is one way of ensuring the totality of a transition relation.

4.1 Snapshot and Snapshot Space

A snapshot is a mapping from variables to values. The first criterion we will compare across the languages is representation of snapshots and variables of a transition system in each language. Note that the more data-oriented aspects of snapshots and the snapshot space are discussed in Chapter 5.

In B, EVENT-B, TLA$^+$, PLUSCAL, and ASMETAL a model has a clause for declaring variables, in DASH variables are declared at the top of the main snapshot block, and in ALLOY the declaration location of variables varies depending on the choice of snapshot representation. Figure 4.1 has exemplars showing snapshot declarations across the languages.

- In B, the variables are declared in the `VARIABLES` clause of the machine.
- In EVENT-B, the variables are declared in the `VARIABLES` part of the machine.
- ALLOY is a more general-purpose modelling language not dedicated specifically to modelling transition systems, and does not have an explicit construct/keyword for declaring a snapshot or its variables. Thus, the choice of snapshot representation is with the modeller. A common paradigm for representing snapshots in ALLOY is using a `State` signature, with its fields corresponding to variables. See [74] for other techniques for snapshot modelling in ALLOY.
- In DASH, a snapshot is explicitly declared using the `state` keyword, and variables are usually declared at the top of each `state` block. As DASH supports labelled control state hierarchies, snapshot definitions can be distributed through nested `state` blocks. Also, DASH has explicit syntax for marking a variable as part of the environment, using the `env` keyword.
- In TLA$^+$, the variables are declared using `VARIABLES`.
- In PLUSCAL, global variables are declared using the `variables` keyword in an algorithm.
- In ASMETAL, variables are declared in a `signature` block. Variables fall into two general categories of `static` constants — which do not change during a machine's run — and `dynamic` variables — which may be changed by an agent's actions or updates. Dynamic variables are the snapshot variables, and are further classified into `monitored` and `controlled`.

4.2 Initialization

This section discusses how to specify the value of the variables for the initial snapshot(s) of the transition system in each language. All the studied languages distinguish syntactically variable initialization from variable declaration, except for PLUSCAL which not only allows but recommends specifying the initial value of each variable together with its declaration. Further, all the languages except ASMETAL

```
// B
ABSTRACT_VARIABLES
  loan,
  member,
  book,
  reservation
```

```
// Dash
conc state Library {
    env in_m: lone MemberID
    env in_b: lone BookID
    members: set MemberID
    books: set BookID
    loans: books one -> one
     (members)
    reservations: books one
     -> one (seq members)
  ...
}
```

```
\* TLA+
VARIABLES members, books,
     loans, reservations
```

```
// Event-B
VARIABLES
  loans
  members
  books
  reservations
```

```
// Alloy
sig Lib {
  members:set Member,
  books: set Book ,
  loan: books -> members,
  membersReservingOneBook: seq
      members -> books,
  Renew: books -> members
}
```

```
\* PlusCal
variables loans, members,
          books, reservations;
```

```
// AsmetaL
signature:
  ...
  controlled members: Powerset(MemberID)
  controlled books: Powerset(BookID)
  controlled loans: Powerset(Prod(BookID, Powerset(MemberID)))
  controlled reservations: Powerset(Prod(BookID, Seq(MemberID)))
  ...
```

Figure 4.1: Snapshots and variables

support some form of nondeterministic assignment for initialization of variables. Figure 4.2's exemplars demonstrate initialization across languages.

- In B, the **INITIALISATION** clause is used to assign values to variables for the initial snapshot(s). The **INITIALISATION** keyword is followed by one or more *substitutions* (see Chapter 6 of [29]), using any of the := (becomes equal substitution), :∈ (becomes part of substitution), and :() (becomes such that substitution) operators.
- In EVENT-B, the **INITIALISATION** clause is used to assign values to variables for the initial snapshot(s), with actions using any of the := (deterministic assignment), :∈ (nondeterministic assignment of a set member), and :| (nondeterministic assignment with a before-after predicate) operators (see Section 3.3.8 of [44]).
- ALLOY does not have an explicit construct/keyword for initializing variables. Commonly, a predicate, conventionally named `init`, is used to constrain the variable values for the initial snapshot(s). Any ALLOY expression over the variables may be used to constrain each variable.
- In DASH, an **init** block is used within a **conc state** to declare the values of the variables in the initial snapshot(s). Any ALLOY expression may be used for constraining the variables.
- In TLA$^+$, a predicate, conventionally named `Init`, is used to describe the values of variables in the initial snapshot(s). Any TLA$^+$ formula may be used to constrain the values of the variables in the `Init` predicate. Note that there is nothing special about the name `Init`, and any other name may be used, since it is not a built-in keyword.
- In the PLUSCAL algorithm language, the initial values of the variables are often specified along with their declaration in the `variables` part of the algorithm. Typical TLA$^+$ formulas may be used for constraining the variables.
- In ASMETAL, the initial values of the variables are given in a `default init` block, using the = operator. ASMETAL does not support any form of nondeterministic variable initialization.

4.3 Transition Relation

This section describes how each language allows a modeller to create a transition relation, *TR*, of a transition system. We describe the languages with respect to this criterion, ranging from those wherein

```
// B
INITIALISATION
  loan := {} ||
  book := {} ||
  member := {} ||
  reservation := {}
```

```
// Dash
conc state Library {
  ...
  init {
    no members
    no books
    no loans
    no reservations
  }
}
```

```
\* TLA+
Init ==
  /\ members = {}
  /\ books = {}
  /\ loans = <<>>
  /\ reservations = <<>>
```

```
// Event-B
INITIALISATION
THEN
  loans, members, books,
     reservations := {}, {},
     {}, {}  // act1
END
```

```
// Alloy
// Lib is the snapshot sig
pred Init [L: Lib] {
  no L.books
  no L.members
  no L.loan
  no L.membersReservingOneBook
  no L.Renew
}
```

```
\* PlusCal
variables loans = <<>>,
          members = {},
          books = {},
          reservations = <<>>;
```

```
// AsmetaL
default init s0:
  function members = {}
  function books = {}
  function loans = {}
  function reservations = {}
```

Figure 4.2: Initialization

the representation of *TR* is the most explicit, to those where it is the least explicit. For each language, we examine its constructs for defining a transition relation, and how stuttering steps are represented in the language. In the next section, Section 4.4, we will examine each language's constructs for defining transitions that are composed together to form *TR*.

Of the languages studied in this thesis, ALLOY, TLA$^+$, and ASMETAL require a more explicit representation of a transition relation, where *TR* is defined mostly in the model text. Figure 4.4 shows exemplars defining a transition relation in each of these three languages. Because *TR* is described via a set of constraints, and not imperative definitions, a declarative modelling language may allow expressing inconsistent transition relations.

In ALLOY, the transition relation, *TR*, is defined completely explicitly, and its form can vary greatly depending on how the snapshot, variables, and transitions are defined. For instance, with a `State` signature as the snapshot representation and its fields as variables, *TR* can be decomposed into predicates that can be viewed as transitions. In Section 4.4, we will discuss conventions used for defining transitions in ALLOY. Stuttering in ALLOY may be modelled explicitly, by writing a transition that constrains some or all of the variables (depending on whether the model divides the variables into separate monitored and controlled variables) to remain unchanged.

In TLA$^+$, the transition relation is defined mostly explicitly. By convention, the transition relation is a predicate `Next` defined as the disjunction of all of the model's transition predicates, which is the method best supported by TLC, the accompanying model checker for TLA$^+$. Though in TLA$^+$ one could write models not using disjunctions to join the transition predicates, TLC is not optimized to handle such models as well, because it rewrites the transition relation as a disjunction of as many simple subactions as possible [76]. If *TR* is not a disjunction, two possible issues arise: First, from a debugging perspective, the visualizer will *not* show *TR* as a composition of smaller transition predicates, thereby making it harder to reason about the model and its behaviour. Second, from a model checking performance perspective, TLC spawns a worker thread for each subaction to explore the snapshot space, but because *TR* was *not* broken down into smaller subactions, TLC may not spawn an optimum number of worker threads. For predicates taking one or more arguments, existential quantification over the corresponding set(s) may be used to bind an element from that set and pass to the predicate. TLA$^+$ has implicit stuttering, and stuttering steps may occur between any two transitions, including when the system has not reached a deadlock. This implicit addition of stuttering to *TR* does not change the meaning of a model, since all TLA formulas are invariant under stuttering (*i.e.* adding or removing stuttering steps does not affect whether or not a behaviour satisfies a temporal formula).

In ASMETAL, the transition relation *TR* is defined mostly explicitly using a special `rule` named r_Main, the *main rule*. The main rule is by convention broken down into smaller rules, each a transition. The main rule specifies when and how each transition will be called. ASMETAL is an imperative-style language, and does not support defining the transition relation as a disjunction of multiple transitions like the other languages with explicit *TR* representation do. Thus, to write a transition relation that would take a randomly-chosen transition each time, we have to declare an enumerated set with each element corresponding to one transition, use the choose rule in the definition of the transition relation to choose an element from that set, and use a switch with a case for each transition, executing the transition corresponding to the chosen element of the enumerated set. This can be cumbersome and error-prone in a model with a large number of transitions, since the modeller may forget to update the switch cases when adding or removing transitions. Stuttering in ASMETAL can be added explicitly, using the skip rule, as well as occurring implicitly in the case of a deadlock, which happens when the update set corresponding to a transition is empty (*i.e.* no variable assignments in the transition) or when the update set is inconsistent (*i.e.* there are conflicting assignments to the same variable, one form of which is simultaneous assignment two the same variable in a parallel block, as shown on Figure 4.3. Implicit stuttering in an ASM allows changes to the monitored variables by the environment, so as to enable potential further progress of the transition system [71]. This stuttering behaviour added implicitly to *TR* also ensures that *TR* is total.

The remaining languages — B, EVENT-B, PLUSCAL, and DASH — each have an explicit construct for defining transitions, which are implicitly composed together to form a transition relation. In each language, only one transition is allowed to be taken in each step, and all different interleavings of the transitions are considered for modelling concurrency.

The transition relation in B, EVENT-B, and PLUSCAL is implicitly formed as follows: at any step, any transition whose precondition is satisfied (*i.e.* is enabled) may be chosen to be taken. There is no requirement on the preconditions of the transitions to be non-overlapping, and more than one transition may be enabled at the same time, resulting in a branch in the snapshot space graph. In the remainder of this section, we will refer to this formation of *TR* as TR_{IMP}.

In B, the transition relation is implicitly formed, per TR_{IMP}, and is a composition of transitions, referred to as operations in B, which may have zero or more preconditions, and one or more substitutions from which postconditions may be derived. A stuttering step in B must be represented explicitly, using the skip generalized substitution — also referred to as the identity substitution — which takes no action.

In EVENT-B, the transition relation is formed implicitly, per TR_{IMP}, and consists of transitions, referred to as events in EVENT-B, which may have zero or more preconditions, and one or more actions from which postconditions may be derived. In EVENT-B, a stuttering step must be denoted explicitly,

```
// AsmetaL
asm inconupd
import StandardLibrary

signature:
  controlled var: Integer

definitions:
  rule r_varinc = var := var + 1
  rule r_vardec = var := var - 1

  main rule r_Main =
    par
      r_varinc[]
      r_vardec[]
    endpar

default init s0:
  function var = 42
```

Figure 4.3: Inconsistent update of a snapshot variable in ASMETAL

using a `skip` event, which is a transition that is always enabled (its guard is **`TRUE`**) and does nothing (it has no actions).

In PLUSCAL, the transition relation is formed implicitly, per TR_{IMP}, wherein the transitions are implicitly disjoined together, A stuttering step in a PLUSCAL algorithm may be represented explicitly in a `process`, using the `skip` atomic instruction, or implicitly, between any two PLUSCAL steps (defined in Section 4.4 below). Since PLUSCAL specifications are ultimately translated to TLA^{+} and checked by TLC, one may opt to not use the automatically generated transition relation and write their own. In the next section, for PLUSCAL we will look at how transitions are created from the `process` descriptions.

In DASH, the transition relation is formed implicitly following the semantics of concurrent, hierarchical state machines. For a transition to be taken, the snapshot must include the source state of the transition, and transitions exiting states at a higher level in the hierarchy have priority over lower states. Particularly distinct from the other languages is the concurrent and hierarchical states found in a DASH model. Because of this concurrency, DASH makes the distinction between big steps and small steps in the transition relation, as shown in Figure 4.5. Big steps consist of multiple small steps, which are each one transition. In a big step, at most one transition per concurrent region can be taken. Monitored (**`env`**ironmental) events can change only at big step boundaries (called a stable snapshot),

```
\* TLA+
Next ==
  \/ \E b \in BookID:
      \/ Acquire(b)
      \/ Discard(b)
      \/ Return(b)
  \/ \E m \in MemberID:
      \/ Join(m)
      \/ Leave(m)
  \/ \E m \in MemberID, b \in BookID:
      \/ Cancel(m, b)
      \/ Lend(m, b)
      \/ Renew(m, b)
      \/ Reserve(m, b)
      \/ Take(m, b)
```

```
// Alloy
pred TransLCR[m: Member]
{
  all l: Lib - LibOrd/last |
    LCR[m, l, l.LibOrd/next]
}
pred LCR[m: Member, L, L': Lib]
{
  some b: Book |
    Cancel[m, b, L, L']
    or Return[m, b, L, L']
    // For test switch Leave and BuggyLeave
    or Leave[m, L, L']
    // or BuggyLeave[L, L']
}
```

```
// AsmetaL
main rule r_Main =
  choose $b in BookID, $m in MemberID, $rule in RuleId with true do
    switch($rule)
      case ACQUIRE:
        r_Acquire[$b]
      case CANCEL:
        r_Cancel[$m, $b]
      case DISCARD:
        r_Discard[$b]
      case JOIN:
        r_Join[$m]
      case LEAVE:
        r_Leave[$m]
      case LEND:
        r_Lend[$m, $b]
      case RENEW:
        r_Renew[$m, $b]
      case RESERVE:
        r_Reserve[$m, $b]
      case RETURN:
        r_Return[$b]
      case TAKE:
        r_Take[$m, $b]
      endswitch
```

Figure 4.4: Explicit transition relation

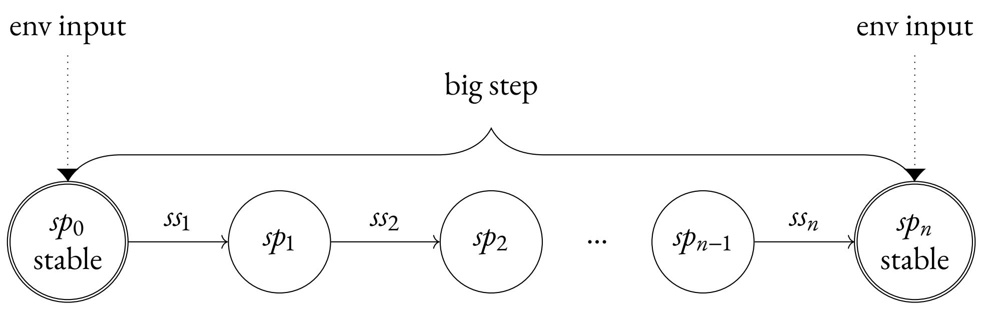

Figure 4.5: Big step (*sp* is a snapshot; *ss* is a small step)

$$
\begin{aligned}
&T(s, s') \\
&= \big[pre_T(s)\big] \overset{?}{\underline{\quad}} \big[post_T(s, s')\big] \\
&= \big[guard_T(s) \land src_T(s) \land evt_T(s)\big] \overset{?}{\underline{\quad}} \big[act_T(s, s') \land dest_T(s') \land genevt_T(s')\big]
\end{aligned}
$$

Figure 4.6: Decomposition of a transition T

so the occurrence of a monitored event can trigger multiple transitions as long as the transitions are in different concurrent regions. Events generated by one transition can trigger other transitions (in different concurrent regions) within the same big step. Implicit stuttering in DASH happens only at the big step boundaries, when no more transitions may be taken. For stuttering when one or more transitions are enabled, an explicit stuttering transition must be used.

4.4 Transitions

This section describes how each language can represent a transition T of a transition system. As we are working with declarative modelling languages, each transition describes a set of pairs $(s, s') \in T$. To create a transition $T \subseteq TR \subseteq S \times S$ we may use two-snapshot predicates, assignment operators, or both; depending on the language. A predicate corresponding to a transition T describing a set of pairs $(s, s') \in T$ can be broken down into pre_T on the source snapshot and $post_T$ on the source and destination snapshots, combined using some logical connective $\overset{?}{\underline{\quad}}$, as shown in Figure 4.6. The transition may only be taken when pre_T is true. Each of pre_T and $post_T$ may in turn consist of one or more explicit language constructs. The preconditions pre_T of the transition may consist of guards $guard_T$, a source labelled control state src_T, and triggering event evt_T; and the postconditions $post_T$ may consist of actions act_T, destination control state $dest_T$, and generated events $genevt_T$, constraining the snapshot and variables after the transition is taken. We will describe the languages in increasing order of explicit language constructs for describing a transition. Figure 4.7 shows exemplars for declaring transitions in each language.

ALLOY does not have a special construct for defining transitions. Commonly, a transition is modelled as a predicate — defined using the `pred` keyword — over unprimed and primed variables, constraining the value of the variables in the source and destination snapshots. Unlike the other languages, primed variables do not carry any special meaning in ALLOY, and are used as a common modelling convention. Though a transition may be represented using a single predicate, Farheen's guidelines [32] separate a transition definition into two separate predicates — one for preconditions over the variables in the source snapshot, and another for postconditions over the variables in the source and destination snapshots — to promote structure. Commonly, the pre and post predicates for each transition are conjoined together to form the main transition predicate (*i.e.* $\overset{?}{\_\_}$ in Figure 4.6 would be $\wedge$), and the transition predicates are then disjoined together to form the transition relation. This is referred to as the *disjunctive modelling method* in Farheen's guidelines. The guidelines recommend this method for decomposing the transition relation and transitions of a transition system because using this method, adding a transition does not change the behaviour of other existing transitions, and is thus more likely to produce a transition relation expected by the modeller. With respect to the frame problem, in ALLOY any variable not constrained in a transition predicate may change nondeterministically from the source to the destination snapshot, and there is no distinction between monitored and controlled variables.

In TLA$^+$, a transition (an *action* in TLA$^+$ terminology) is a two-snapshot predicate over unprimed and primed versions of all variables, constraining each variable in the source and destination snapshots. Similarly to ALLOY, in TLA$^+$ a transition may be further broken down into separate pre and post parts for clarity. The preconditions of a transition are commonly a series of one or more conjoined formulas over unprimed variables in the source snapshot and optionally over the transition arguments if any, and the postconditions or actions of a transition are conjoined formulas over primed and unprimed variables, constraining each variable in the destination snapshot. The pre and postconditions are often conjoined together, *i.e.* $\overset{?}{\_\_}$ in Figure 4.6 would be $\wedge$. With regards to the frame problem, TLA$^+$ requires that all transitions constrain the value of every variable, either by constraints on the primed and unprimed variables or by marking them with the `UNCHANGED` keyword. TLA$^+$ does not make a distinction between monitored and controlled variables.

ASMETAL does not have any construct for decomposing a transition into separate pre and post parts; and a transition definition consists only of an action — a `rule` in ASMETAL terminology. An action may be one of several rules, producing a set of assignments to `controlled` variables. The transition rule may be a simple rule such as the `skip` rule or the update rule (*i.e.* variable assignment, using the := operator), or a more complex rule such as `par`allel or `seq`uential block, `if`, `case`, `forall`, `choose`, *etc.*, which enables composition and combinations of other rules together. A transition in ASMETAL does not have preconditions on when it may be taken, and conditional rules such as `if` and `case` may be used to constrain the execution of the action(s) of that transition. In ASMETAL, for $(s, s') \in T$ the transformation of s to s' is defined as the effect of applying as an atomic step the result of a consistent update set (one with no conflicting variable assignments) on s. With respect to the frame problem, any controlled variable not assigned to in a transition is unchanged by that transition, and retains its value from the source snapshot.

In B, a transition is referred to as an *operation*, each consisting of one or more actions — called a *generalized substitution* — which may take several forms (see Chapter 6 of [29] for an exhaustive list of the forms). An action may be a compound substitution — such as a block substitution (`BEGIN S END`) or a preconditioned substitution (`PRE P THEN S END`) with precondition `P` required to be true before calling the operation (otherwise the operation will not be enabled and cannot be executed) and `S` the body of the substitution — or a simple one like := (becomes equal), :∈ (becomes part of), or :( ) (becomes such that). The simpler substitutions can either be the entire body of a substitution, or be used in one of the compound substitutions like the ones mentioned above. Note that := and :∈ are used similar to assignment in typical programming languages, whereas with :( ) the modeller may refer to the value of the variable before substitution (in the source snapshot) using the `$0` suffix. For example, `x$0` would refer to the value of `x` in the source snapshot, and `x` to its value in the destination snapshot. With respect to the frame problem, in B any variable not assigned to in a substitution retains its value and is unchanged by that transition. B does not distinguish between monitored and controlled variables.

In EVENT-B, a transition is referred to as an *event*, and is a simplification of a B *operation*. In contrast to B, in EVENT-B a transition has only one general form, consisting of one or more of the following parts: parameters, guards, and actions. A *guard*, defined in the **`WHERE`** clause of the transition and the only form of precondition in EVENT-B, is the precondition required to be true for the transition to be enabled. A transition may have an arbitrary number of *parameter*s. Like a snapshot variable, each parameter has a type — which must be declared as a guard of that transition — and a unique name. An *action*, consisting of an assignment, describes how the source and destination snapshots relate, and is a simplification of B's generalized substitutions. Each action has a label, used for identifying and/or referring to that action. Labels are used throughout EVENT-B tool support, and are useful for identifying the role/kind of a construct involved in a machine or proof about the machine. By convention, an action label starts with the *act* prefix, and a guard label starts with the *grd* prefix. An EVENT-B assignment operator is either deterministic or nondeterministic. The $:=$ operator is for deterministic assignment of the right-hand side value to the left-hand side variable, $:\in$ for nondeterministic assignment of an element of the right-hand side set to the left-hand side variable, and $:|$ for (nondeterministically) constraining the value of the left-hand side variable with the before-after predicate given on the right-hand side. Similarly to B, $:=$ and $:\in$ are used without primed variables, and $:|$ is used with primed version of variables referring to their value in the destination snapshot. Regarding the frame problem, similarly to B, in EVENT-B any variable not assigned to using an assignment operator is unchanged by the transition and retains its value. No distinction is made between monitored and controlled variables in EVENT-B.

In PLUSCAL, we can leverage the language's constructs for expressing concurrency and nondeterminism to model a transition system in the form of an algorithm. Similar to a programming language, PLUSCAL has a control flow semantics that defines the meaning for the algorithm text and how it may be executed. When modelling a transition system in PLUSCAL, a single-*step* `process` will be a transition in the resulting transition system. The preconditions of a transition are specified using an `await` (or `when`) statement which acts like a guard, allowing the process body to be executed only when the guard expression evaluates to `TRUE`. In PLUSCAL, a single *step*, referred to as an atomic action, corresponds to the execution of the statements contained between one *label* and the next. A label is an identifier marking a location in a PLUSCAL algorithm, similar to a label in a traditional programming language like C. An algorithm has a program counter pointing to the current label being executed, and may be thought of as corresponding to non-hierarchical control states. Since normally the body of a PLUSCAL process is executed only once, whereas a transition in a transition system may be taken any number of times as long as it is enabled, we add a `goto` statement at the very end of each process to jump back to the label at the very beginning of that process, allowing the process to be taken again. In addition to global (`algorithm`-wide) variables, each PLUSCAL `process` may have local variables declared using the `variables` keyword. A local or global variable *x* can be initialized to *expr* using a declaration of the form `variable` *x* `=` *expr*. In process body, variables may be assigned to using the $:=$ operator, and will otherwise retain their value from the source snapshot. PLUSCAL does not distinguish between monitored and controlled variables.

DASH is the language with the most language constructs for modelling a transition of a transition system studied in this thesis. A transition in DASH is defined using the **`trans`** keyword, optionally consisting of the parts described below. The preconditions of a DASH transition may be divided into three parts:

1. The guard condition of the transition, denoted using the **`when`** keyword, which is an ALLOY

expression over the snapshot variables that when true the transition would be enabled.

2. The source labelled control state of the transition, denoted using the **`from`** keyword. Since DASH has explicit control state representation using **`state`** blocks, labelled control state blocks may be nested to form a CONTROL STATE HIERARCHY, useful for grouping related states together. In the absence of an explicit **`from`** part, the most immediate state containing the transition definition will be used as the source labelled control state of that transition.

3. The event triggering the transition, denoted using the **`on`** keyword, for triggering the transition whenever a certain event is fired. Events are useful for modelling broadcast communication and cascading effects. The (optional) **`on`** part indicates the name of the event triggering the transition.

The postconditions of a DASH transition may be divided into three parts:

1. The actions of the transition, denoted using the **`do`** keyword, are two-snapshot ALLOY expressions over unprimed and primed variables describing the value of each variable in the destination snapshot, modelling the effects of executing the transition.

2. The destination labelled control state of the transition, denoted using the **`goto`** keyword. In the absence of an explicit **`goto`** part, the most immediate state containing the transition definition will be used as the destination labelled control state of that transition.

3. The generated events, denoted using the **`send`** keyword, for firing one or more events upon the execution of the transition. The (optional) **`send`** part denotes the name of an event or a set of comma-separated events to be generated when the transition is taken.

DASH semantics define two kinds of steps: *big steps* and *small steps*. A big step consists of one or more sequential small steps meant to represent the transition system reacting to the external environment or its own internal changes. Each small step corresponds to a transition, and small steps are taken until the system cannot take anymore transitions, at which point it is said to have become *stable*. With respect to the frame problem, monitored variables (marked with **`env`**) in DASH are allowed to change from one snapshot to the next when the next snapshot is stable (*i.e.* at the big step boundaries), and will otherwise retain their values from the source snapshot. For controlled variables, if the primed version is mentioned in the action of a transition, it is assumed that the action will constrain it; otherwise, they are forced to retain their value from the source snapshot.

```
// B
Join (member_) =
PRE
   member_ : MEMBERID &
   member_ /: member
THEN
   member := member \/ { member_ }
END;
```

```
// Dash
trans join {
  when !(in_m in members)
  do members' = members + in_m
}
```

```
\* TLA+
Join(member) ==
  /\ member \in MemberID
  /\ member \notin members
  /\ members' = members \union {member}
  /\ UNCHANGED << books, loans, reservations >>
```

```
// Event-B
Join:  // not extended ordinary
  ANY
    member
  WHERE
    member : MemberID not theorem   // grd1
    member /: members not theorem   // grd2
  THEN
    members := members \/ {member}  // act1
END
```

```
// Alloy
// Lib is the snapshot sig
/*-------------------
   Join
-------------------*/
pred CanJoin[m: Member, L: Lib] {
  // m does not exist in the Library.
  no (m & L.members)
}

pred Join[m: Member, L, L': Lib] {
  ----Precondition-----
  CanJoin[m, L]
  -----Postcondition------
  // add the m in the set of members
  L'.members = L.members + m
  ------Nochanges-----
  NoChangebooks[L, L']
  NoChangeloan[L, L']
  NoChangeSeqBook[L, L']
  NochangeRenew[L, L']
}
```

```
\* PlusCal
process Join = "Join"
variable member \in MemberID
begin
  join:
    when member \notin members;
    members := members \union {member};
    goto join
end process
```

```
// AsmetaL
rule r_Join($m in MemberID) =
  if (notin(members, $m)) then
    members := union(members, {$m})
  endif
```

Figure 4.7: Transitions

4.5 Invariants

This section describes how invariants for a transition system are represented in each language. An invariant of a transition system is a formula that is true in every snapshot of the transition system. In other words, an invariant must hold in the initial snapshot and after every transition.

In B and EVENT-B, invariants are written in the `INVARIANT` and `INVARIANTS` section of a machine respectively. In a model checking setting, *e.g.* using ProB, the invariants are checked to hold in every snapshot. In a theorem proving setting, the invariants must be proven to be established by the initialization, and be preserved by every transition.

In ALLOY and DASH, invariants are written in `fact` blocks. DASH additionally has `invariant` blocks. In both languages, invariants are constraints on snapshots and limit the reachable snapshot space of the model.

In TLA$^+$ and PLUSCAL, invariants are defined as TLA$^+$ predicates. In a model checking setting, these predicates must be added to the `Invariants` portion of the TLC settings for the model, which will then be checked to hold in every snapshot. In a theorem proving setting, invariants appear as consequent of an implication in a `THEOREM`.

In ASMETAL, invariants are defined using the `invariant over` keywords, in the `definitions` section of an ASM after the `rule` (transition) definitions. ASMETAL invariants are verified to hold in every snapshot of the model during analysis (*e.g.* animating or model checking).

4.6 Inconsistency

In declarative models, inconsistency stems from contradictions in logical formulas. In the context of declarative transition systems, we consider three kinds of inconsistency that may occur in the description of a transition system:

- **Deadlock**: refers to a snapshot that has no explicitly modelled outgoing transitions. In the absence of stuttering steps in a declarative modelling language, this means a non-total transition relation. Some languages perform implicit stuttering steps when a transition system reaches a deadlock, thereby ensuring the totality of the transition relation and enabling potential further progress of the transition system. Another method for ensuring the totality of a transition relation is making the transition preconditions complete; *i.e.* setting up the transitions such that the disjunction of the preconditions of all transitions combined is true.

- **Contradictory** *TR*: when the logical formula corresponding to the transition relation, *TR*, is a contradiction (*i.e.* is never true), and therefore there does not exist a snapshot trace satisfying the constraints of *TR*'s definition. In B, EVENT-B, PLUSCAL, and DASH, creating a contradictory *TR* explicitly is not a concern, since the transition relation of the model is created implicitly from the elements described by the modeller, and the above languages do not have an explicit language construct for defining *TR* in the model text.

- **Contradictory transition postcondition**: when the logical formula corresponding to the postcondition of a transition is a contradiction. One possible form of contradictory transition postconditions is simultaneous assignment (also sometimes referred to as parallel assignment) to the same variable in a transition.

In ALLOY and DASH, it is possible to have a non-total transition relation that reaches a deadlock wherein no transitions can be taken, or a contradictory *TR* for which no satisfying instances exists. In ALLOY, *TR* is written explicitly by the modeller and thus may be the direct source of inconsistency; whereas in DASH, *TR* is constructed implicitly, and *TR*-related inconsistencies would be those trickling up from the individual transitions. It is also possible to write contradictory transition postconditions in both ALLOY and DASH, possibly resulting in a contradictory *TR* (when no other transition is enabled). Though DASH has implicit stuttering at the big step boundaries, they only happen when all the transition preconditions are false; and they will not help when a transition's precondition is true but its postcondition is false. Figure 4.9 is an exemplar of a transition system with a contradictory transition postcondition. A particular case of contradictory *TR* in ALLOY and DASH happens when ALLOY's `util/ordering` module is used to impose an order on the snapshot signature, and the scope specified for the snapshot signature does not match *exactly* the number of snapshots needed for the snapshot trace for the property being checked. For example, if the scope of a signature `S` is set to 4, then `open util/ordering[S]` will force `S` to have four elements, and creates a linear ordering over those four elements. This is because the ordering module constrains all the atoms permitted by the given scope to exist, to enable an optimization on the internal representation of the order. DASH's use of the ordering module by default implies this behaviour, but it can be disabled by unchecking the "Path based instances" option in the DASH editor. Figure 4.8 shows an example of this type of inconsistency in both ALLOY and DASH. One possible fix would be to change the scope of `Snapshot` from 4 to 3.

In TLA$^+$, it is possible to write a transition relation that eventually reaches a deadlock, as well as writing a contradictory *TR* for which no satisfying instances exists. A contradictory transition postcondition results in that transition never being enabled. Inconsistency in TLA$^+$ comes in several

```
// Alloy
open util/ordering[Snapshot] as SO

abstract sig A {}
one sig a1, a2, a3 extends A {}

sig Snapshot {
  v: one A
}

pred a1_to_a2[s, s': Snapshot] {
  s.v = a1
  s'.v = a2
}

pred a2_to_a3[s, s': Snapshot] {
  s.v = a2
  s'.v = a3
}

pred init[s: Snapshot] {
  s.v = a1
}

pred next[s, s': Snapshot] {
     a1_to_a2[s, s']
  or a2_to_a3[s, s']
}

pred path {
  init[SO/first]
  all s: Snapshot, s': s.SO/next |
    next[s, s']
}

run path for 4 Snapshot
```

```
// Dash
abstract sig A {}
one sig a1, a2, a3 extends A {}

conc state Example {
  v: one A

  trans a1_to_a2 {
    when { v = a1 }
    do { v' = a2 }
  }

  trans a2_to_a3 {
    when { v = a2 }
    do { v' = a3 }
  }

  init {
    v = a1
  }
}

// with the ``Snapshot scope'' option set to 4; i.e.
//   run path for 4 Snapshot, 0 EventLabel expect 1
// in the generated alloy translation
```

Figure 4.8: Inconsistency due to use of `util/ordering` and scope not matching *exactly*

```
abstract sig A {}
one sig a1, a2, a3 extends A {}

conc state Example {
  v: one A
  trans a1_to_a2 {
    when { v = a1 }
    do { v' = a2 }
  }
  trans a2_to_a3 {
    when { v = a2 }
    // contradictory postcondition:
    do {
      v' = a2
      v' = a3
    }
  }
  init { v = a1 }
}
```

Figure 4.9: DASH model with a contradictory transition postcondition

forms. The simplest is an obviously false specification — for instance $Init \triangleq 1 = 2$, or $Init \triangleq a \in \varnothing$ — which results in TLC errors like "The spec is trivially false because Init is false" or "Init is never enabled".

In ASMETAL, it is possible to write a transition relation for an ASM that eventually reaches a deadlock. Figure 4.10 is an example of an ASM that eventually reaches a deadlock. The interpretation of the deadlock depends on the tooling used: for instance, the interactive runner for Asmeta considers the deadlock snapshot the end of execution and will terminate the run when that snapshot is reached, whereas the Asmeta animator will stutter infinitely when the deadlock state is reached.

In B and EVENT-B, a transition with a contradictory postcondition (*e.g.* an action/substitution like $x :\in \varnothing$ that is always false) is effectively never enabled (same as a transition with a false precondition), and the transition system would thus reach a deadlock if no other transition is enabled. In Figure 4.11, the transitions `contra_pre` and `contra_post` in both B and EVENT-B are always disabled and each transition system is deadlocked. Both B and EVENT-B employ static checks as part of their type system to catch various modelling errors like type mismatches (*e.g.* `a` $= \varnothing$ in the above exemplar) or simultaneous assignment to the same variable (*e.g.* `a := a1; a := a2`) in the same transition.

For PLUSCAL, the general conditions described for TLA$^+$ above may apply to PLUSCAL models

```
asm example

import StandardLibrary

signature:
  domain Coord subsetof Integer
  controlled v: Coord

definitions:
  domain Coord = {1..5}

  rule r_vinc =
    while v < 3 do
      v := v + 1

  main rule r_Main = r_vinc[]

default init s0:
  function v = 1
```

Figure 4.10: ASM that eventually reaches deadlock

as well. However, PLUSCAL has additional safeguards to make certain kinds of inconsistency unrepresentable in valid models. This is achieved by using PLUSCAL *labels* to impose restrictions on valid statements and expressions. For instance, each labelled section may only contain at most one assignment statement for each variable. This eliminates a case of contradictory transition postcondition corresponding to the logical formula $a' = b \land a' = c$ where $b \neq c$. Figure 4.12 shows an example of this, and PLUSCAL will show an error about a missing label between lines 8 and 9.

```
// B
MACHINE example
SETS
  A = {a1, a2}
VARIABLES
  a
INVARIANT
  a: A
INITIALISATION
  a := a1
OPERATIONS
  contra_pre =
  PRE
    FALSE = TRUE
  THEN
    a := a2
  END;
  contra_post =
  PRE
    a = a1
  THEN
    a :: {}
  END
END
```

```
// Event-B
CONTEXT
  c0
SETS
  A
CONSTANTS
  a1
  a2
AXIOMS
  partition(A, {a1}, {a2}) // axm1
```

```
// Event-B (continued)
MACHINE
  m0
SEES
  c0
VARIABLES
  a
INVARIANTS
  a : A // inv1
EVENTS
  INITIALISATION:
  THEN
    a := a1 // act1
  END
  contra_pre:
  WHERE
    FALSE = TRUE // grd1
  THEN
    a := a2
  END
  contra_post:
  WHERE
    a = a1   // grd1
  THEN
    a :: {} // act
  END
```

Figure 4.11: A deadlocked transition system in B and EVENT-B

```
EXTENDS Naturals

(* --algorithm example
variables a = 1;
process contra_post = "contra_post"
begin
    contra_post:
        a := 2;
        a := 3
end process
end algorithm *)
```

Figure 4.12: Invalid contradictory transition postcondition thanks to PLUSCAL's *label* semantics

Chapter 5

Data Modelling

In this chapter, we discuss in depth the description of the data aspects of models in each of the declarative modelling languages examined in this thesis. This data is constrained in the guards and actions of transitions and in the invariants of the model. We first lay out the terminology we use in this chapter in Section 5.0. In Section 5.1 and Section 5.2 we discuss the primitives and constructors of composite units of data in each language respectively. In Section 5.3 we discuss the expression syntax of each language and notable differences between them. Section 5.4 discusses the notion of events and how they can be modelled in modelling languages that do not have that notion. Section 5.5 discusses the declaration of constants across the languages. Section 5.6 discuses well-formedness conditions and typechecking of type signatures in each language. In Section 5.7 we look at how the sizes of the sets used in the models is set in each programming language. Finally, in Section 5.8 we mention some missing features and constructs that are absent in all the languages we studied. Table 5.1 summarizes the differences in the data aspects of the selected declarative modelling languages with respect to modelling transition systems.

5.0 Terminology

The following are the terminology we use to describe the data-related characteristics of declarative modelling languages.

- **Primitives:** are the smallest unit of data in a language, usually consisting of scalars and sets.

Table 5.1: Summary of comparison of data aspects of languages

Criteria \ Language	B	EVENT-B	ALLOY	DASH	TLA$^+$	PLUSCAL	ASMETAL
Primitives	scalars, sets	scalars, sets	sets	same as ALLOY	scalars, sets	same as TLA$^+$	scalars, sets
Constructors	fun, rel, rec, multiplicity	fun, rel, multiplicity	rel, multiplicity	same as ALLOY	fun, rel	same as TLA$^+$	rel
Built-ins	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$, $\mathbb{B}$, str, seq, tree	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{N}_1$, $\mathbb{B}$	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{B}$, str, seq, ord, graph	same as ALLOY	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$, $\mathbb{B}$, str, rec, seq, bag	same as TLA$^+$	$\mathbb{Z}$, $\mathbb{N}$, $\mathbb{R}$, $\mathbb{C}$, $\mathbb{B}$, char, str, seq, bag, map, `Undef`
Events	—	—	—	`event`	—	—	—
Constants	`CONSTANTS`	`CONSTANTS`	—	—	`CONSTANTS`	`CONSTANTS`	`static`
Type signatures & typechecking	✓	✓	✓	✓	—	—	✓
Subtypes	—	✓	✓	✓	✓	✓	✓
Scopes	✓	✓	✓	✓	✓	✓	—

Legend:
$\mathbb{Z}$: Integers, $\mathbb{N}$: Naturals, $\mathbb{N}_1$: Naturals excluding zero, $\mathbb{R}$: Reals, $\mathbb{C}$: Complex numbers, $\mathbb{B}$: Booleans.
fun: function, **rel**: relation, **rec**: record, **seq**: sequence, **ord**: ordering, **str**: string, **char**: character.

- **Constructors:** are operators that create **composite units of data** from primitives or other composite data units. For example, functions, relations, and records are constructors. Constructors include **multiplicities**, which impose constraints that limit the values in the composite data being constructed.

- **Built-ins:** of a language are the names of particular primitives or particular composite data units that are part of the syntax of the language or part of its standard libraries, and are available to the modeller without declaration in the model.

- **Standard library:** of a language is the collection of one or more modules that are distributed with the tool support for the language, and include the built-ins and utility functions that may or may not be hard-coded and/or built into the language.

- **Event:** denotes an occurrence at a moment in time. In some languages, an event may be used as a precondition of a transition. Events are often helpful in describing the abstract behaviour of reactive systems.

- **Constant:** is a mapping from a name to a value, either a primitive or a composite data unit. In contrast to a snapshot variable, a constant retains its value throughout all of the transitions of the transition system. Not all of the languages distinguish between a variables and constants.

- **Type signature:** is syntax in a language to denote the kind of object contained in a snapshot variable, constant, or quantified variable as either a primitive or a composite data unit.

- **Typechecking:** refers to checking that the constraints expressed in the type signatures are consistent with the use of the data in the formulas.

- **Subtype:** is the name of a subset of values of another set that can be used in a type signature.

- **Scope:** is the size of a set of objects; *i.e.* the total number of distinct elements it contains.

5.1 Primitives

In this section, we examine the primitives of each language. Primitives are the smallest unit of data in a language, and usually consist of scalars and sets. In declarative modelling languages, similar to programming languages, scalars are objects like numbers, Booleans, strings, or a user-declared object.

A set is an unordered collection of distinct objects, and is itself an object as well. Sets are a critically important part of declarative modelling languages, and are the fundamental building blocks for more complex units of data. We also examine whether each language supports subtypes, indicating whether or not it is possible to declare a set that is a subset of another set and can be used in a type signature.

The primitives of B consist of scalars and sets. In B, sets are declared in the **SETS** section of the machine, and are of two kinds: enumerated and deferred. An enumerated set is one where its elements are listed in the declaration of the set, whereas the declaration of a deferred set does not include a list of its elements. B does not support subtypes, but they can be modelled using membership predicates. For example, a parent set `A` is declared on line 3, and a constant predicate `isC` is declared and constrained on lines 8 and 11 to model the subtyping. `isC` must be used as a precondition for any use of the `R1` relation. To declare a snapshot variable to be a scalar from a given set, we add a type signature using the '`:`' set element operator and the name of the set, as on line 16 of the B code block in Figure 5.1. To declare a snapshot variable whose value is a set of elements from a given set, we use the **POW** operator (for powerset) in the type signature, as on line 15.

The primitives of EVENT-B consist of scalars and sets. Sets are declared in the **SETS** section of a *context*, separate from the *machine* definition. There are two kinds of sets in EVENT-B: carrier and enumerated. A carrier set can be declared simply by adding its name to the **SETS** section of the context. To declare an enumerated set, in addition to adding its name to the **SETS** section, we list its elements under the **AXIOMS** section, using the `partition` operator. The first argument to `partition` is the name of the set, and subsequent arguments are disjoint sets of elements, as on line 18 of the EVENT-B code block in Figure 5.1. Declaring a subtype in EVENT-B is also done using the `partition` operator by providing the name of the parent set as the first argument, and the names of the children as subsequent arguments, as on line 16. To declare a snapshot variable to be a scalar from a given set, we add a type signature using the '`:`' set element operator and the name of the set, as on line 12. To declare a snapshot variable whose value is a set of elements from a given set, we use the **POW** operator (for powerset) in the type signature, as on line 11.

The primitives of ALLOY consist only of sets, and there are no scalars in ALLOY. A "scalar" in ALLOY is represented using a singleton set. For instance, `5` is really syntactic sugar for the singleton set `{5}`. Sets are declared using the **`sig`** keyword (for signature). If formulas are added as 'signature facts' immediately following a signature, they are implicitly quantified over the elements of the signature. Subset signatures can be declared using the **`in`** keyword. Note that multiple subsets declared using **`in`** may not necessarily be disjoint. To declare disjoint subsets of a set, the **`extends`** keyword can be used, as on line 3 of Figure 5.1. Enumerated sets can be declared using the **`enum`** keyword, as on line 4, which

```
// textual description
sets and constants:
  A, B: set
  C: subset of A
  c1: a constant of type C
  c2: integer constant with value 3
variables and events:
  v1: powerset of A
  v2: integer
  ev1, ev2: events
```

```
// AsmetaL
signature:
  enum domain Event = {EV1 | EV2}
  abstract domain A
  abstract domain B
  domain C subsetof A
  monitored evs: Powerset(Event)
  controlled v1: Powerset(A)
  controlled v2: Integer
  static c1: C
  static c2: Integer
definitions:
  function c2 = 3
```

```
// B
SETS
  A; B;
  Event = {ev1, ev2}
VARIABLES
  v1, v2, evs
CONSTANTS
  isC, c1, c2
PROPERTIES
  // subtype predicate
  isC : A --> BOOL &
  c1 : A &
  c2 : INTEGER & c2 = 3
INVARIANT
  v1 : POW(A) &
  v2 : INTEGER &
  evs : POW(Event)
```

```
// Event-B
CONTEXT
  c0
SETS
  A
  B
  Event
CONSTANTS
  C
  nC
  ev1
  ev2
  c1
  c2
AXIOMS
  partition(B, C, nC)
  card(C) > 0
  partition(Event, {ev1}, {ev2})
  c1 : C
  c2 : INT & c2 = 3
```

```
// Event-B (continued)
MACHINE
  m0
SEES
  c0
VARIABLES
  v1
  v2
  evs
INVARIANTS
  v1 : POW(A)
  v2 : INT
  evs : POW(Event)
```

Figure 5.1: Primitives in snapshot declarations across languages (part one)

```
// Alloy
sig A, B {}
sig C extends A
enum Event { ev1, ev2 }
sig Const {
  c1: one C,
  c2: one Int
} { c2 = 3 }
sig Example {
  v1: set A,
  v2: one Int,
  evs: set Event
}
```

```
// Dash
open util/integer
sig A, B {}
sig C extends A {}
sig Const {
  c1: one C,
  c2: one Int
}
state Example {
  v1: set A,
  v2: one Int,
  event ev1 {}
  event ev2 {}
  invariant { Const.c2 = 3 }
}
```

```
\* TLA+
EXTENDS Integers, FiniteSets
CONSTANTS A, B, C, c1, c2
VARIABLES v1, v2, evs

Event == {"ev1", "ev2"}

TypeOK ==
  /\ v1 \in SUBSET A
  /\ v2 \in Singleton(Int)
  /\ evs \in SUBSET Event
  /\ c1 \in C
```

```
\* PlusCal
EXTENDS Integers, FiniteSets
CONSTANTS A, B, C, c1

(* --algorithm example
variables v1, v2, evs;

define
c2 == 3
\* helper predicate
Singleton(S) == {{i} : i \in S}
Event == {"ev1", "ev2"}

TypeOK ==
  /\ v1 \in SUBSET A
  /\ v2 \in Singleton(Int)
  /\ evs \in SUBSET Event
  /\ c1 \in C
  /\ c2 \in Int
end define
end algorithm *)
```

Figure 5.1: Primitives in snapshot declarations across languages (part two)

is syntactic sugar for declaring a parent `abstract sig`, and declaring its elements as `one sig` singleton sets, each of which `extends` the parent set. To declare a snapshot variable whose value is an element of a given set, we use the `one` multiplicity keyword in the type signature, as on line 11. To declare a snapshot variable whose value should be a set of elements from a given set, we use the `set` multiplicity, as on lines 10 and 12. DASH's `state` syntax is translated to an ALLOY signature, and the above statements about ALLOY apply to DASH as well.

The absence of real scalars in ALLOY has various implications. On the one hand, everything being sets implies having a simpler, non-overloaded set of operators that can readily be applied to various objects in the language. On the other hand, it means there are some unusual characteristics of the language. For instance, in ALLOY Boolean values are not scalars, and even though formulas have Boolean values, variables never do [43, p. 137]. The reason for the absence of Boolean-valued expressions is that their presence would render an expression like `not p` ambiguous when `p` contains zero or more than one Booleans. Instead, Boolean-like valuations can be mimicked using the `Boolean` set from the `util/Boolean` module of ALLOY's standard library.

The primitives of TLA$^+$ and PLUSCAL consist of scalars and sets. Sets can be declared either as constants in the `CONSTANTS` section of a module, or as a predicate definition whose right-hand side is a set-valued constant expression. To declare a set to be a subset of another set, we can use the `SUBSET` keyword. To declare a snapshot variable whose value is an element from a given set, we add a type signature using the '`\in`' set membership operator and the name of the set, as on line 10 of the TLA$^+$ code block in Figure 5.1. To declare a snapshot variable to be a scalar from a given set, we can either use the powerset keyword `SUBSET` along with the set membership operator `\in`, or just use `\subseteq`. Line 11 shows an example of the former.

In ASMETAL, the primitives are scalars and sets. Sets are declared using the `abstract domain` keywords, as on lines 4 and 5 of the ASMETAL code block in Figure 5.1. To declare elements for a set, one can either change the set declaration from `abstract domain` to `enum domain` and list the enumeration elements as on line 3, or declare the elements using the `static` keyword, as on lines 10 and 11. In ASMETAL, a subtype is declared using the `subsetof` keyword, as on line 6. To declare a snapshot variable to be a scalar from a given set, we add a type signature with the name of the set, as on line 9. To declare a snapshot variable whose value is a set of elements from a given set, we use the `Powerset` keyword, as on lines 7 and 8.

Each language consists of some built-in sets and scalars that are elements of those sets. In B and EVENT-B, the built-ins are Booleans, naturals, and integers. B additionally has strings. TLA$^+$ and PLUSCAL have Booleans, naturals, integers, reals, and strings. In ASMETAL, the built-ins are Booleans, naturals, integers, reals, complex numbers, characters, and strings. ALLOY and DASH have integers and Booleans, but the names typically used for scalars in these sets are actually singleton sets. ALLOY also has strings, but currently the only elements in the set of strings are the string literals used in the current model, and the analyzer cannot find arbitrary instances for strings like it can do for other sets.

5.2 Constructors and Multiplicities

In this section, we examine constructors and multiplicities across the languages. Constructors are operators that create composite units of data from primitives or other composite data units. Multiplicities in a constructor impose constraints that limit the values in the composite data. Constructors appear in type signatures. We also discuss how formulas can be used to constrain composite units of data in many languages, but we do not consider these formulas to be constructors because they are not part of a type signature, and a formula can do much more than constrain the form of one kind of data. Figure 5.2 helps compare the languages with respect to constructors and multiplicities using an equivalent snapshot declaration across the languages.

The composite units of data in B are functions, relations, and records, each with various pre-defined operations available to the modeller. Functions can be created using the "arrow" operators built into the B language. These arrow operators are: ⇸ (partial functions), ⟶ (total functions), ⤔ (partial injections), ↣ (total injections), ⤀ (partial surjections), ↠ (total surjections), and ⤖ (total bijections). Relations can be created using the ↔ arrow operator. The B code block in Figure 5.2 shows examples of using some of these operators. Line 5 shows a total function `f1`, and line 6 shows a surjective function `f2`. `R1` on line 7 is a binary relation, with the additional 1-to-2 constraint imposed in a formula on lines 9 – 11. `s1` is declared to be a sequence of `A`'s, as on line 8. Since functions and relations in B are both represented as sets of pairs, they can also be created using the set literal notation along with the ↦ ("maplet") pair constructor. In addition to functions and relations, B also has records with named fields, as shown in the B code block of Figure 5.3. The **`struct`** operator denotes the set of records matching the given field signatures, and is used in the type signature when declaring a new record. A new instance of a record can be created using the **`rec`** operator. Built-in constructors in B include (zero-indexed) sequences and trees.

EVENT-B's composite units of data are functions and relations. EVENT-B, similar to B, has various arrow operators for constructing functions and relations. In addition to all of B's functions and relations arrows mentioned above, EVENT-B has three additional arrows for constructing specific kinds of relations. Namely, $\trel$ (total relations), $\srel$ (surjective relations), and $\strel$ (total surjective relations). In Figure 5.2, line 12 shows a total function `f1`, and line 13 shows a surjective function `f2`. `R1` on line 14

```
// textual description
uninterpreted sets from Figure 6.1
snapshot variables:
  f1: total function from A to B
  f2: monitored surjective function from
     A to B
  R1: 1-to-2 binary relation from C to B
  s1: sequence of A's
```

```
// B
VARIABLES
  f1, f2, R1, s1
INVARIANT
  f1 : A --> B  &
  f2 : A -->> B &
  R1 : A * B    &
  s1 : seq(A)   &
  !(v).(v: A & isC(v) <=>
    card(R1(v)) = 2 &
    not(isC(v)) => card(R1(v)) = 0)
```

```
// Event-B
MACHINE
  m0
SEES
  c0
VARIABLES
  f1
  f2
  R1
  s1
INVARIANTS
  f1 : A --> B
  f2 : A -->> B
  R1 : C <-> B
  s1 : NAT +-> A
  !v . v : C =>
    (card({v} <| R1) = 1 &
     card(R1[ {v} ]) = 2)
```

```
// AsmetaL
signature:
  controlled f1: Powerset(Prod(A, B))
  monitored  f2: Powerset(Prod(A, B))
  controlled r1: Powerset(Prod(C, B))
  controlled s1: Seq(A)
  derived isfun: Powerset(Prod(D1, D2)) -> Boolean
  derived issurfun: Powerset(Prod(D1, D2)) -> Boolean
  derived is1to2: Powerset(Prod(D1, D2)) -> Boolean
  derived ran: Powerset(Prod(D1, D2)) -> Powerset(D2)
definitions:
  function dom($f in Powerset(Prod(D1, D2))) =
    {$d1 in D1, $d2 in D2 |
      contains($f, ($d1, $d2)): $d1}
  function isfun($f in Powerset(Prod(D1, D2))) =
    isEmpty({$d1 in D1, $d21 in D2, $d22 in D2 |
      contains($f, ($d1, $d21))
      and contains($f, ($d1, $d22))
      and neq($d21, $d22): $d21})
  function issurfun($f in Powerset(Prod(D1, D2))) =
    isfun($f) and eq(D2, {$d1 in D1, $d2 in D2 |
      contains($f, ($d1, $d2)): $d2})
  function is1to2($r in Powerset(Prod(D1, D2))) =
    eq(D1, {$d1 in dom($r), $d21 in D2, $d22 in D2 |
      contains($r, ($d1, $d21))
      and contains($r, ($d1, $d22))
      and neq($d21, $d22): $d1})
  invariant inv_f1_fun over f1: isfun(f1)
  invariant inv_f2_surfun over f2: issurfun(f2)
  invariant inv_r1_1to2 over r1: is1to2(r1)
```

Figure 5.2: Composite units of data in snapshot declarations across languages (part one)

```
// Alloy
sig Example {
  f1: A -> one B,
  f2: A some -> one B,
  R1: C -> B,
  s1: seq A
}

fact R1_multiplicity {
  all a: A | #(a.R1) = 2
}
```

```
// Dash
state Example {
  f1: A -> one B,
  env f2: A some -> one B,
  R1: C -> B,
  s1: seq A,

  fact R1_multiplicity {
    all a: A | #(a.R1) = 2
  }
}
```

```
\* TLA+
EXTENDS Integers, FiniteSets, Sequences
VARIABLES f1, f2, R1, s1

\* helper operators
Range(f) == {f[x] : x \in DOMAIN f}
SurFun(a, b) == {f \in [a -> b] : b = Range(f)}
RelRange(R) == {x[2] : x \in R}
RelDomRes(S, R) == {x \in R : x[1] \in S}
OneToN(R, n) ==
  \A x \in RelDom(R) :
    Cardinality(RelRan(RelDomRes(R, {x}))) = n

TypeOK ==
  /\ f1 \in [A -> B]
  /\ f2 \in SurFun(A, B)
  /\ R1 \in C \X B  \* \X is cross product
  /\ OneToN(R1, 2)
  /\ s1 \in Seq(A)
```

```
\* PlusCal
EXTENDS Integers, FiniteSets, Sequences

(* --algorithm example
variables f1, f2, R1, s1;

define
\* helper operators
Range(f) == {f[x] : x \in DOMAIN f}
\* ... all helpers identical to the TLA+ ones

TypeOK ==
  /\ f1 \in [A -> B]
  /\ f2 \in SurFun(A, B)
  /\ R1 \in C \X B  \* \X is cross product
  /\ OneToN(R1, 2)
  /\ s1 \in Seq(A)
end define
end algorithm *)
```

Figure 5.2: Composite units of data in snapshot declarations across languages (part two)

```
// B
MACHINE rectest
VARIABLES vrec, vbar
INVARIANT
  vrec : struct(foo : INTEGER, bar : BOOL) &
  vbar : BOOL
INITIALISATION
  vrec := rec(foo : 9, bar : FALSE) ||
  vbar := TRUE
OPERATIONS
  setvbar = vbar := vrec'bar
END
```

```
\* TLA+
------------ MODULE rectest ------------
EXTENDS Integers
VARIABLES vrec, vbar

Init ==
  /\ vrec = [vrec |-> 9, bar |-> FALSE]
  /\ vbar = TRUE

setvbar ==
  /\ vbar' = vrec.bar
  /\ UNCHANGED vrec

TypeOK ==
  /\ vrec \in [foo : Int, bar : BOOLEAN]
  /\ vbar \in BOOLEAN

Spec ==
  /\ Init
  /\ [][setvbar]_<<vrec, vbar>>
  /\ []TypeOK
========================================
```

Figure 5.3: Records in B and TLA$^+$

is a binary relation, with the additional 1-to-2 constraint imposed on lines 16 – 18. Since EVENT-B does not have a built-in sequence constructor, we can use a partial function from `NAT` to `A` to represent `s1` as a sequence of `A`'s, as on line 15.

TLA$^+$ does not have type signatures, and thus no constructors; but composite units of data can be specified using TLA$^+$ formulas. The composite data units in TLA$^+$ are functions and relations. TLA$^+$ also supports records, which are built on top of functions. The TLA$^+$ code block of Figure 5.3 shows an example of declaring and using TLA$^+$ records. Lines 6 – 12 showcase formulas that constrain data to be of a certain composite form. For instance, we define `SurFun(a, b)` to return the set of all surjective functions from `a` to `b`, `f1` is required to be a total function from `A` to `B` (line 15), `f2` a surjective function from `A` to `B` (line 16), `R1` a binary relation between `C` and `B` (line 17) and constrained to be a 1-to-2 relation (line 18), and `s1` a sequence of `A`'s (line 19).

ASMETAL's primary composite data unit is relations, created using the `Powerset` and `Prod` constructors for creating sets and tuples respectively. Although ASMETAL has a dedicated function constructor arrow that can be used in type signatures, it is not always be the best representation for a function, since the possible operations on a function are limited. Lines 3 – 5 show the declaration of three relations, which are constrained by the invariants on lines 28 – 30. The `f1` total function from `A` to `B` is defined as a set of pairs, where the set elements satisfy the constraint laid out in `isfun`: the set of elements from `f1`'s domain that map to two different elements in `f1`'s range must be empty. The surjective function `f2` from `A` to `B` is defined as a set of pairs, where the set elements satisfy the constraint laid out in `issurfun`: first, `f2` must already be a function; and additionally, `f2`'s codomain must match the set of elements mapped to by `f2` (*i.e.* every element of `f2`'s codomain must be mapped to by an element in `f2`'s domain). The 1-to-2 binary relation `r1` from `C` to `B` is defined as a set of pairs, where the set elements satisfy the constraint laid out in `is1to2`: each element of `r1`'s domain must map to two different elements from its range. Lastly, `s1` is declared to be a sequence of `A`'s.

ALLOY's composite unit of data is relations, created using the `->` arrow constructor. As relations are ubiquitously used in ALLOY models, ALLOY provides convenient dot join (`.`) and box join (`[]`) operators [43, p. 57-62], which resemble record field access and array indexing respectively. Somewhat hidden to users of ALLOY is that ALLOY does not actually have record objects; and though signatures and their fields mimic records and record fields, signatures are actually sets (*i.e.* unary relations) and signature fields are top-level identifiers themselves, each a relation mapping their parent signature to the field's value from the set declared in the field's type signature. This can be confusing to new users trying to understand the output of the analysis (*e.g.* counterexamples for model checking), when a variable name is actually a relation.

ALLOY provides a range of **multiplicity constraints** within its type signatures. A multiplicity constraint is a constraint limiting the number of elements in the range associated with elements in the domain. ALLOY has four multiplicity keywords, **set** (any number), **lone** (zero or one), **one** (exactly one), and **some** (one or more). When the type signature is a set (*i.e.* unary relation), it can be prefixed with a multiplicity keyword as in `x: m e`, where `x` is the set being constrained, `m` is the multiplicity keyword, and `e` is the set-valued bounding expression. The default multiplicity for a set-valued type signature is *one*. Thus, if the multiplicity keyword `m` is omitted, `x: e` makes `x` a 'scalar' (*i.e.* a singleton set). Here is the meaning of the other multiplicities on sets:

`x: set A`	`x` becomes a subset of the set `A`
`y: lone B`	`y` can either be empty or a singleton subset of the set `B`
`z: one C`	`z` becomes a singleton subset of the set `C`; equivalent to `z: C`
`x: some D`	`x` becomes a nonempty subset of the set `D`

When the type signature is a relation (of arity greater than one), it *cannot* be prefixed with a multiplicity keyword. Multiplicity keywords may appear around the `->` arrow constructor, as in `x: e1 m->n e2`, where `m` and `n` are multiplicity keywords, and `e1` and `e2` are sets. The expression `x: e1 m->n e2` means the relation `x` is constrained to map each member of `e1` to `n` members of `e2`, and to map `m` members of `e1` to each member of `e2`. `e1` and `e2` need not be sets (unary relations) or even named relations; and they may be any arbitrary expressions. The above description is generalized by replacing 'member' with 'tuple'. Below are examples of some commonly-used multiplicities on relations:

`x: A ->one B`	`x` becomes a total function from `A` to `B`
`y: A ->lone B`	`y` becomes a partial function from `A` to `B`
`z: A one->one C`	`z` becomes a total bijective function from `A` to `C`
`x: A some->one C`	`x` becomes a total surjective function from `A` to `C`
`y: A some->lone D`	`y` becomes a partial surjective function from `A` to `D`
`z: A some->some D`	`z` becomes a nonempty relation from `A` to `D`

DASH has the same constructors and multiplicities as ALLOY.

Having considered constructors across the languages, we now briefly touch on the names of built-ins in each language. B's built-ins include sequences and trees. ALLOY's built-ins include a constructor that imposes linear total order on other sets, (zero-indexed) sequences, strings, and graphs. DASH shares ALLOY's built-ins. TLA$^+$'s built-in's include composite data units like records, bags, and (one-indexed) sequences, from TLA$^+$'s standard modules. Recent TLC versions override the definitions from most of these standard modules in the host language (Java) for improved efficiency. PLUSCAL shares TLA$^+$'s built-ins. ASMETAL's `StandardLibrary.asm` [11] contains a large number of built-ins such as bags, maps, products (tuples or pairs), and (zero-indexed) sequences.

5.3 Expressions

In this section, we discuss briefly the syntax of each language for writing formulas, and the operators and constructs provided in each language for writing expressions. Since all of the languages are based on first-order logic and set theory, their expressions are reasonably similar in succinctness. They all provide operations to create and modify primitives and composite data units, such as sets, relations, and functions. In the following, we discuss only the interesting differences in the expression syntax of each language.

The expressions of B and EVENT-B have a more imperative style than declarative, in that initialization and changing of variables are primarily done using the := assignment operator. Additionally, both languages have a nondeterministic assignment operator :∈ for setting the value of a variable to a nondeterministically-chosen element from a set. However, as declarative modelling languages, both B and EVENT-B support changing a variable in a declarative style using the : ( ) and : | operators respectively, for when an a regular assignment is not flexible enough. In B, assignments statements are combined using the `||` and `;` operators, for parallel and sequential assignments respectively. Since both relations and functions in B and EVENT-B are represented as sets of pairs, in addition to the arrow operators described earlier they can also be created using the set literal notation along with the $\mapsto$ ("maplet") pair constructor.

ALLOY and DASH have expression languages that include set operators mixed with first-order logic. Models typically make extensive use of the join operator because every unit of data is a relation.

TLA$^+$ has a small number of essential operators for conveniently working with sets and functions, TLA$^+$ also has the a `CHOOSE` operator, `IF..THEN..ELSE` expressions, `CASE` expressions, `LET..IN` expressions, and `LAMBDA`. Even though TLA$^+$ is a very small language, one can easily define new operators . For instance, on line 6 of Figure 5.2 we define the `Range` operator which is absent in TLA$^+$, to return the range of a function. TLA$^+$ uniquely supports recursive and higher-order operators, which take other operators as arguments.

The ASMETAL language rarely uses symbols and most operators have long-form textual names.

ASMETAL's built-in units of data are opaque, in that operations defined on primitives cannot be applied to composite data units. This means that the operations on the built-in composite data units are limited by what is currently implemented in the tool support for the language. As of now, in ASMETAL sets are the most flexible/featureful construct in the language, and the modeller may need to fall back to using them as the underlying representation of a data unit if the built-in data unit provided by the language does not include the operations they need. For instance, few operations are available for the built-in function constructor in ASMETAL. Since an ASMETAL function cannot be treated as a set of tuples, the set operations are not defined on functions. As such, when more flexibility is needed, *e.g.* for writing formulas for dealing with a function as a whole, the built-in function constructor cannot be used, and instead the more flexible 'set of pairs' representation must be used.

5.4 Events

An event denotes an occurrence at a moment in time, such as a button being pressed or a card swiped. An event may be used as a precondition of a transition. Among the declarative modelling languages studied in this work, events are a feature unique to DASH. A statechart event in DASH can be added to a snapshot using the `event` keyword, as on lines 12 and 13. The semantics of events in DASH is that they persist as long as transitions are enabled, allowing multiple transitions to be taken in response to an input from the environment.

To model events in languages other than DASH which do not have a notion of events built into them, we use regular snapshot variables to model them. There are several possible approaches for keeping track of the triggered state of the events of a model. One approach is to use one snapshot variable for each event, with the value of each variable drawn from a two-element set, for whether or not the event is currently triggered. A language's built-in Boolean set or another two-element set with more descriptively-named elements are good candidates for the type signature of this snapshot variable. Another approach is to declare an enumerated set with one element per event. Then, declare one set-valued snapshot variable whose value is a subset of the enumerated set of events, with each currently triggered event being a member of this set. Constraints must be added to specify the desired semantics of events, such as whether the event persists. Figure 5.1 shows exemplars using this approach across the languages (besides DASH, which has built-in support for events).

5.5 Constants

A constant is a mapping from a name to a value, either a primitive or a composite data unit. Constants retain their value throughout all of the transitions of the transition system. Figure 5.1 shows examples of declaration of constants across the languages. In B, constants are declared under the `CONSTANTS` section of a machine, and are further constrained in the `PROPERTIES` section. In EVENT-B constants are declared under the `CONSTANTS` sections of a context, and are further constrained in the `AXIOMS` section of the context. The B and EVENT-B code blocks in Figure 5.1 show examples of this. In ASMETAL, constants are declared under the `signature` section after snapshot variables, using the keyword `static`, as shown on lines 10 and 11 of the ASMETAL code block in Figure 5.1. A constant may be assigned a specific value under the `definitions` section, as on line 13.

ALLOY and DASH do not have a keyword or construct for constants. A constant is declared like a regular variable as a field of a signature. To avoid confusing constants with snapshot variables, we declare them in a separate signature, conventionally named `Const`. In ALLOY, further constraints, such as having the constant be a certain literal value, can be imposed using either a fact or a predicate. In DASH, the constraints are added in an `invariant` block inside a `state`. Lines 5 – 8 and 5 – 8 of the ALLOY and DASH code blocks of Figure 5.1 show examples of constant declarations in ALLOY and DASH; and lines 8 and 14 of the ALLOY and DASH code blocks show an example of a constraint being imposed on the constant `c2` using a signature fact and an invariant in ALLOY and DASH respectively.

In TLA$^+$ and PLUSCAL, constants are declared under the `CONSTANTS` section of the module. For model checking with TLC, every constant must be constrained in the “Model Overview” page of the model in the TLA$^+$ toolbox. A constant can be constrained using either an “ordinary assignment” (any valid TLA$^+$ expression that can be assigned to a variable), or can be constrained to be a “model value” or a “set of model values”. In TLC terminology, a “model value” is a unique value equal only to itself. Alternatively, if we use a predicate definition like `c2 == 3`, we will not have to declare `c2` as a constant under `CONSTANTS`, and every mention of `c2` is replaced with `3` during analysis. This works in both TLA$^+$ and PLUSCAL, and is similar to the practice of defining constants and magic values in C programs using preprocessor macros. Note that this approach only yields a constant when the right-hand side expression is a constant expression, such as an integer literal, and for instance `c2 == CHOOSE cc \in Integer: TRUE` would *not* result in `c2` being a constant, since `CHOOSE` is not guaranteed to always return the same value each time it is evaluated. The TLA$^+$ and PLUSCAL code blocks of Figure 5.1 show an example of the former and latter method respectively.

5.6 Well-formedness and Typechecking

Well-formedness is a condition under which a formula or expression is valid in a formal language, and has a well-defined meaning. In this section, we investigate the notions of well-formedness across the seven declarative modelling languages, and the constructs in each language that help make it easier for modellers to find or avoid mistakes in their models, such as use of type signatures and typechecking. In this section, we focus on static well-formedness checking techniques, which include parsing and typechecking. All the languages studied in this work do basic forms of syntax checking when parsing input models.

Typechecking is the process of checking whether the use of data in the formulas conforms to the constraints expressed in the type signatures. With respect to typechecking, the following two questions arise: *First*, does a language have type signatures? The answer is that B, EVENT-B, ALLOY, DASH, and ASMETAL do have type signatures, and TLA$^+$ and PLUSCAL do not. *Second*, does a language with type signatures have a separate typechecking pass from other forms of analysis? The answer to this question is yes; all of the languages we studied that have type signatures also have a typechecking pass separate from the main analysis (*e.g.* model checking).

In TLA$^+$, there are few well-formedness checks beyond the basic syntax checking done by the parser. TLA$^+$ does not have type signatures, so variable and constant declarations only consist of names. Any typing constraints must be stated with other invariants and are properties checked to hold in every snapshot of the transition system during the main analysis. An example of such typing constraints is the `TypeOK` predicate defined on lines 15 – 19 of Figure 5.2. In our experience, having to manually add typing invariants in the absence of a typechecker can be error-prone and can hinder the debugging experience, as the errors arising from the violation of these typing constraints can result in cryptic error messages.

Compared to TLA$^+$, PLUSCAL has a number of safeguards in place that make certain classes of bugs syntactically invalid, thus allowing the parser to catch them when the modeller asks for the TLA$^+$ translation of their PLUSCAL model [49]. Examples include mistakes such as multiple assignments to the same variable in a transition, and having unreachable statements. These are eliminated by PLUSCAL's notion of *labels*, and where they may or must appear in the transition definition. For the above two examples, the former is prevented due to the rule that a variable may only be assigned to once between any two labels in a PLUSCAL process, and the latter is eliminated by the requirement that a label must be added for any statement that follows a `goto` or `return` statement. For PLUSCAL models, typing constraints can be added using the same approach as TLA$^+$ described earlier.

In ALLOY, the only well-formedness checking is a kind of typechecking. Type signatures constrain the model's reachable snapshot space. There are two kinds of type errors in ALLOY [43, 31]:

1. since ALLOY's logic assumes each relation has one fixed arity, an expression resulting in relations of mixed arity is illegal; and
2. an expression that can be shown to be redundant itself or contain a redundant sub-expression, based on the declarations alone, is ill-typed. A common example of this is an expression that is redundant due to being equal to the empty relation.

The current implementation of DASH performs several well-formedness checks in addition to ALLOY's. Currently, typechecking in DASH is not as thorough as in ALLOY, and some type errors will only be caught in the generated ALLOY model. DASH's well-formedness checks are as follows [70]:

- Every top level **`state`** must be declared as **`conc`** (error).
- If a model has state hierarchy then there must be one default child state defined (error).
- Either all children states at the same level of the hierarchy are concurrent or none at all (error).
- Only snapshot variable declarations can be primed (error).
- Monitored (**`env`**ironmental) events cannot be generated in a transition (warning).
- Monitored (**`env`**ironmental) variables cannot be primed (error).
- A transition action must constrain the next value of a variable (error).
- Elements in a state should have different name (not currently enforced).

Besides well-definedness conditions, B and EVENT-B also use typechecking to statically catch type errors like assigning a value from a set to a variable with a type signature declaring a different/incompatible set, and applying a function to an argument not matching its type signature. To ensure that models only contain well-defined formulas, B and EVENT-B rely on well-definedness proof obligations. These are formulas corresponding to conditions that are expected to hold when performing certain actions, such as integer division and function application, in order for the action to be well-defined. For example, division by zero is not well-defined, and neither is applying a function to an element outside its domain. In a theorem proving setting, these well-definedness conditions have to be proven, along with other proof obligations. In a model checking or animation setting, ProB performs well-definedness checking during constraint solving, model checking, or animating. In B, type signatures are declared in the **`INVARIANT`** section along with other invariants. Similarly, type signatures in EVENT-B are declared in the **`INVARIANTS`** section along with the other invariants of the machine.

Type signatures in ASMETAL are included along with each variable declaration. Further, the ASMETAL grammar uses various syntactic rules to distinguish between language elements:

- name of (local) variables must start with a dollar sign (`$`);

- enumerated set elements must start with two upper-case letters, followed by zero or more upper-case letters, underscores, or digits;
- domain (set) names must start with an upper-case letter;
- rule names must start with a lower-case ‘`r`’, followed by an underscore (*i.e.* “`r_`”);
- function names (including snapshot variables) must start with a lower-case letter, but *not* start with the string “`r_`” (to avoid confusion with rule names);
- invariant names must start with the letters “`inv`” followed by an underscore (*i.e.* “`inv_`”); and
- natural numbers must have a ‘`n`’ suffix (*e.g.* `14n` is the natural literal for the number fourteen), and integers must not (*e.g.* `13` is the integer literal for the number thirteen).

These syntactic rules allow the ASMETA parser to detect a variety of errors statically during parsing, including some type errors that would normally be caught by a typechecker rather than a parser. For example, (mis)using an integer where a natural number is expected, or confusing an enumerated set element with the parent set or a variable. The ASMETA typechecker in turn helps catch other ill-typed formulas and errors like trying to calculate the sum of one and the empty set (`{} + 1`), or assigning a set to a variable previously declared to be a string.

5.7 Scopes

In this section, we look at the constructs in each of the declarative modelling languages and/or their tool support provide for setting the scope (size) of each of the sets used in the model. None of the languages studied in this work require specifying the scopes as part of the model. However, some of the languages allow the definition of scopes to be part of the model.

In ASMETAL, there is no default set size, and user-defined domains and their elements must be explicitly stated if needed. For instance, a signature like `abstract domain A` is sufficient for declaring a new set `A` and using it in the model, including with a `choose` rule. However, the Asmeta Animator and Simulator currently enter an infinite loop if we try to animate or simulate a model without scopes for all sets. To use the abstract domain, we have to declare its elements. For example, `static a1: A` declares an element `a1` in `A`. Alternatively, we could either declare a new concrete domain that is a `subsetof` the abstract domain and specify elements for it and use it instead of using `A` directly; or, change `A` into an enumerated domain like `enum domain A = {AA1 | AA2}.`

For B, the ProB animator and model checker requires that all deferred sets to be given a finite cardinality. This is done by adding to the **`PROPERTIES`** section a **`card`**`(SETNAME) = n` predicate for each deferred set SETNAME, or by adding to the **DEFINITIONS** section a definition scope_SETNAME == m..n ; or alternatively scope_SETNAME == n, equivalent to 1..n. For sets with no specified cardinality, a DEFAULT_SETSIZE=2 will be used. To limit the scope of implementable integers (INT) and naturals (NAT and NAT1), we can add the definitions SET_PREF_MININT == x and SET_PREF_MAXINT == y in the **DEFINITIONS** section to set MININT to x and MAXINT to y, which default to -1 and 3 respectively.

As ProB integrates into Rodin (the tool support for EVENT-B), the above descriptions about adding scopes to B models mostly apply to EVENT-B models as well. The difference is that EVENT-B does not have a **DEFINITIONS** section like B does; so the only method for setting specific sizes for carrier sets is by adding a **card**(SETNAME) = n axiom for each carrier set SETNAME. The default set size, the default values of MININT and MAXINT, and other ProB preferences may be changed from the Rodin tool by going to the Window menu, selecting Preferences, and clicking on ProB in the left pane.

In TLA$^+$, the size of sets may be specified under the ASSUME section of a model using the Card operator. The cardinality of the largest set that the TLC model checker can handle is 1000000 by default. This can be changed using the -maxSetSize x command-line option to set the value to x, if using TLC from the command-line. When using the TLA$^+$ toolbox, the value can be changed by opening the Model Overview, clicking on Additional TLC Options, expanding the Parameters section, and changing the option there. The upper bound for this option is Java's Integer.MAX_VALUE. The largest value for TLA$^+$'s Int and Nat sets is Integer.MAX_VALUE, and the smallest value for Int is Integer.MIN_VALUE.

In ALLOY and DASH, scopes are upper bounds or exact sizes of sets, and are specified for each **run** or **check** command. All top-level signatures have a default scope of 3. The two exceptions to this rule are the two special **Int** and **seq** signatures, both of which have a default scope of 4. **Int**'s scope is the maximum bit-width for integers. For example, a scope of 6 allows **Int** to range from -32 to +31. For **seq**, the assigned scope is the length of the largest allowed sequence. For example, a scope of 5 on **seq** allows sequences of up to 5 elements. Given a predicate x and an assertion y, in its simplest but still useful form, a command may look like **run** x or **check** y, where no scope is specified and thus the above default scopes will be used. Note that this does not apply to the special signatures **Int** and **seq**, and their scopes only change when set explicitly, as explained below. The following explanations apply to both **run** and **check** commands, but we will use **run** in our examples. Given the signatures A, B, and C, **run** x **for** 5 sets the upper bound scope for A, B, and C to 5. To use a different scope for some signatures, we use the **but** keyword: **run** x **for** 5 **but** 4 B sets the upper bound for every signature to 5, except for B, which is given an upper bound of 4. To force an exact scope for a specific signature, we can use the **exactly** keyword: **run** x **for** 5 **but** 4 B, **exactly** 6 A sets an upper bound of 5 for all signatures, except for B which is given an upper bound of 4, and A which is forced to have an exact size of `6`. We do not have to set a default scope for all signatures in a command, as long as we explicitly specify the scope for each signature or if ALLOY can infer it implicitly. For details about the cases where ALLOY can calculate the scopes implicitly, see [43, p. 283].

5.8 Missing Features

None of the languages explicitly support operators for dynamic allocation of parts of the state (*i.e.* "`new`" as in Spin [40]). Further, none of the languages have a "message passing channel" feature like Spin does, and such a construct must be modelled using snapshot variables and techniques ensuring correct synchronization. Also, none of the languages have a built-in construct for time; but it can be modelled using techniques such as the "explicit-time" approach explained in [48, 78], wherein the current time is represented as the current value of a snapshot variable, and the passage of time is modelled using a transition that increments the value of that variable.

Chapter 6

Modularity

To create large models, some form of modularity is needed. In this chapter, we discuss how each language provides support for modularity in a description of a transition system. Modularity can be evaluated from two points of view:

(1) How can a description of a single transition relation be decomposed into multiple parts? This decomposition may take the form of

   (a) subtransition systems that are composed to create the single top-level transition relation implicitly or explicitly, or

   (b) subformulas relevant to other aspects of the model, such as a grouping information (axioms, *etc.*) regarding a unit of data.

   A subtransition system is a full description of a transition system. We call (1)(b) a data decomposition.

(2) How can a description of a single transition system be decomposed into multiple files? Typically a decomposition into multiple files involves interfaces that can be parameterized and have private and public parts.

It may be possible for the decompositions of (1) to be realized in multiple files. In the following, we describe the different means within a language of decomposing the transition relation (*i.e.*, (1) above) and within this discussion address when/how different parts of the transition relation can be realized in multiple files. Both (1) and (2) may involve differing namespaces.

Table 6.1: Summary of comparison of modularity across the languages

Criteria \ Language	B	EVENT-B	ALLOY	DASH	TLA$^+$	PLUSCAL	ASMETAL
sub*TR*s	✓	—	✓	—	✓	—	✓
sub*TR* namespaces	—	—	—	✓	—	—	—
Data decomposition into multiple files	✓	✓	✓	✓	✓	✓	✓
File import	**SEES**, ...	**SEES**	**open**	**open**	EXTENDS	EXTENDS	import
File export	varies	entire context	non-**private**	—	non-LOCAL	non-LOCAL	export
File parameterization	✓	—	✓	✓	✓	✓	—
File namespaces	—	—	✓	—	✓	✓	—
Syntax overloading	—	—	✓	✓	—	—	✓

In B, the definition of the transition relation is implicit, and it is possible to decompose a transition system into subtransition systems. Each B machine resides in one file, under one namespace. The B language has extensive constructs for creating relationships between machines, using the keywords **SEES**, **INCLUDES**, **PROMOTES**, **EXTENDS**, **USES**, and **IMPORTS** [29]. A B machine can reference the name of another machine along with these keywords to establish the desired kind of relationship with that machine. The **SEES** keyword grants a module read-only access to *see* the data components (*i.e.* sets, constants, and variables) of another module, and the **INCLUDES** keyword grants read-write access to the data components of the included machine, allowing instantiation if the machine is parameterized (over a scalar or set parameter). More specifically, **INCLUDES** allows establishing a subtransition system relationship where the **SETS**, **CONSTANTS**, **PROPERTIES**, **VARIABLES**, **INVARIANT**, **ASSERTIONS**, and **INITIALISATION** sections of the included machine are in effect prepended to those of the including machine. The details of the remaining keywords and the relationships they establish are beyond the scope of this work. There is no support for syntax overloading in B.

In EVENT-B, the definition of the transition relation cannot be further decomposed into subtransition systems. In EVENT-B, there are two kinds of files: contexts and machines. Contexts describe non-changing parts of the system, namely the sets, constants, and axioms; and a machine contains the snapshot and transitions (the transitions are implicitly combined to build the transition relation). A machine can *see* any number of contexts by adding their names under its **SEES** section. The only possible relationship between two machines in a project is where one machine **REFINES** another. There is no support for syntax overloading in EVENT-B.

An orthogonal perspective on the notion of modularity in B and EVENT-B is the concept of *refinement* [14, 15, 29, 44]. The idea behind refinement is starting the description of a transition system at a very abstract level and with minimal details, and gradually refining that description over several refinement steps to arrive at a more concrete and complete description. Both B and EVENT-B have tool support dedicated to assisting the modeller with performing refinement steps and doing them correctly, by presenting various proof obligations that need to be proven correct at each step. Details on refinement is beyond the scope of this work.

In ALLOY, the definition of the transition relation is explicit as a formula. Therefore any means in ALLOY of decomposing formulas can be used to decompose the description of a transition system. In Chapter 4, we discussed some typical styles of decomposing transition systems into different formulas (*e.g.*, with pre and postconditions). Within one ALLOY file, the namespace includes all of the identifiers declared/defined in the current file.

In ALLOY, formula decomposition can be split across multiple files. This decomposition could be for subformulas of the transition system or for a grouping of axioms regarding a unit of data. ALLOY refers to each file as a module, which may be given a name using the `module` keyword at the beginning of the file. By default, all of the identifiers declared or defined in a file are exported, along with the exported identifiers of all of the other files imported into the current file using the `open` keyword. However, signatures, their fields, functions, and predicates can be marked as `private`, making them private to the current module and not appear in the namespaces of other modules that import the current module. To assign a dedicated namespace to the identifiers of an imported module, we can use `open..as`. For example, `open utils as u` will bring the definitions of the `utils` module into scope under the `u` namespace, and an identifier named `test` can be referred to using `u/test`. In ALLOY, a module can be parameterized over an argument, similar to the notion of genericity in programming languages. For instance, ALLOY's ordering module is parameterized over a signature on which the module will impose a linear total order. In ALLOY, for syntax overloading, fields in different signatures can use the same name, as long as the two signatures do not overlap (by one being a subset of the other). In other words, the type signatures for two fields with the same name must different at least in the first column. Further, two predicates or functions may share the same name, so long as the type signatures for their arguments are not identical (*i.e.* an unambiguous name resolution possible).

In DASH, the definition of the state hierarchy for the transition relation must all reside together in one file because the transition relation is defined implicitly. This includes DASH invariants (declared using `invariant` or `fact`), which must be in the same file as the transition system. In DASH, each `state` has a dedicated namespace. Separate namespaces create interfaces between states, while still allowing global communication. A reference to an identifier in another state must be given by its fully qualified name. A qualified name is formed by following the state hierarchy separating state names with '/' and then adding the element name. In DASH, the data modelling aspects of a transition system are described separately from the state hierarchy and therefore can be separated into multiple files as in ALLOY. ALLOY's file-level module system and namespaces are available to DASH models, with the exception of per-file dedicated namespaces (using `open..as`), which are not yet implemented in DASH. The details of ALLOY's syntax overloading apply to DASH as well.

In TLA$^+$, the definition of the transition relation is explicit as a formula. Therefore any of the means of decomposing formulas in TLA$^+$ can be used to decompose the description of a transition system. Within one TLA$^+$ file, the namespace includes all of the identifiers declared/defined in the current file.

In TLA$^+$, formula decomposition can be split across multiple files. This decomposition could be for subformulas of the transition system or for a grouping of axioms regarding a unit of data. TLA$^+$ refers to each file as a module, which must be given a name using the `MODULE` keyword at the beginning of the file. By default, the identifiers of all of the predicates defined in a module are exported, along with those of the other modules imported into the current module. To import another module into the current module, we use the `EXTENDS` keyword. To import the definitions of a module `M` into a dedicated namespace, instead of importing it directly, we add a definition `M == INSTANCE M`. We can then use `M!def` to refer to a definition `def` from the `M` module. In TLA$^+$, a module is parameterized over its `CONSTANTS`. As such, if `M` has declared `CONSTANTS A, B` (which may potentially be higher-order operators), they must be instantiated using `M == INSTANCE M WITH A <- e1, B <- e2`, where `e1` and `e2` are TLA$^+$ expressions such as literals or identifiers of sets or constants from the current namespace. To mark a definition or an instantiation as local to the current module, we use the `LOCAL` keyword, preventing it from appearing in the namespaces of the other modules importing the module. TLA$^+$ supports a number of user-definable symbols, but as with other definitions, they must not clash with any existing definitions for those symbols, as TLA$^+$ does not support syntax overloading.

In PLUSCAL, a transition system is defined in an `algorithm` block. The definitions of the transitions represented using PLUSCAL `process`es must all reside together in that `algorithm` all in one file, because the transition relation is defined implicitly. Each `process` can optionally have local variables only available to that process, declared and initialized using the `variables` keyword before the beginning of the body of the transition. Locally-scoped bindings using `LET..IN` TLA$^+$ expressions are possible as well. In PLUSCAL, the data modelling aspects of a transition system can be described separately from the transition definitions, and therefore can be separated into multiple files as in TLA$^+$. TLA$^+$'s file-level module system is available to PLUSCAL models as well. The details of TLA$^+$'s user-definable symbols and lack of syntax overloading apply to PLUSCAL as well.

In ASMETAL, the definition of the transition relation is explicit as an ASMETAL `rule`. Therefore any of the means of decomposing rules in ASMETAL can be used to decompose the description of a transition system. Within one ASMETAL file, the namespace includes all of the identifiers declared or defined in the current file.

In ASMETAL, a model can be split across multiple files for subtransition systems and data decomposition. ASMETAL refers to each file as a module, and there are two kinds of modules. A module containing a transition system — *i.e.* a `main rule` (transition relation) and an `init` (initial snapshot) section — is given a name using the `asm` keyword. Modules not including the above two constructs can be created using the `module` keyword, and may contain type signatures and specifications for data units. A module can selectively export a subset of its definitions using a comma-separated list of identifiers with the `export` keyword, or export everything at once using `export *`. To import a module, we use the `import` keyword along with the name of the module, and optionally a parenthesized comma-separated list of identifiers if only some of the module's definitions are desired. ASMETAL does not support prefixed/namespaced imports, and there is no support for parameterized modules. In ASMETAL, functions and transition definitions may be overloaded, so that functions with different type signatures can share the same name.

In summary, the B, ALLOY, and TLA$^+$ languages allow decomposition of the model into subtransition systems across multiple files, while EVENT-B, DASH, PLUSCAL, and ASMETAL only allow data decomposition across multiple files.

Chapter 7

Case Studies

In this chapter we present several case studies across the data- vs. control-oriented spectrum that we carried out to help us compare the languages with respect to our developed comparison criteria. We dedicate a section to each case study, in which we examine the models of that case study in each language, mentioning the noteworthy characteristics and the differences between them based on the criteria in the control and data modelling chapters. When discussing a particular criterion, we put the criterion's name in **bold** in the sentence. Our case studies were not large enough to exercise the differences presented in the modularity chapter. We conclude each section with our recommendations as to which language(s) are better suited for modelling that particular case study.

Table 7.1 shows the sizes of each model of each system across the languages. The entry in row *i* column *j* is the number of lines of code for the model of the case study *i* in the language *j*.

Table 7.1: Lines of code for each case study across the languages

Case study \ Language	B	EVENT-B	ALLOY	DASH	TLA$^+$	PLUSCAL	ASMETAL
EHealth	62	111	135	95	120	101	87
Digital Watch	135	210	295	112	197	160	142
Musical Chairs	68	84	130	65	101	106	97
Library Management	180	164	317	120	146	151	207
Railway	82	86	387	280	79	84	78

7.1 EHealth

The first case study tackled in this thesis is the EHealth (Electronic Health) system, originally done in TLA$^+$ by Professor Jonathan S. Ostroff [64]. The goal of the EHealth system is to make sure that medications prescribed to patients are safe, by keeping track of dangerous interactions between the medications recorded in the system. The system allows a transition adding a medication to a patient's prescription if and only if it does not interact dangerously with any of the other medications previously prescribed to that patient.

The EHealth system falls on the very data-oriented end of the data- vs. control-oriented characterization spectrum due to its use of rich **primitives** (such as sets and tuples) and **constructors** (such as relations and functions), and having no need for fine control of when a transition is relevant beyond the preconditions of each action. It consists of the following components:

- two sets representing all patients and medications in the universe;
- four snapshot variables representing sets of patients, medications, interactions, and prescriptions registered in the system; where the variable for interactions maps each medication to a set of medications with which it has undesired interactions, and the variable for prescriptions maps each patient to a set of medications prescribed to that patient; and
- transitions for adding patients and medications, and for adding and removing interactions and prescriptions.

We consider the following three safety properties for the EHealth system:

- Symmetry of interactions: medication m_1 has an undesired interaction with a medication m_2 if and only if m_2 has an undesired interaction with m_1.
- Irreflexivity of interactions: no medication has an undesired interaction with itself.
- Safety of prescriptions: for all pairs of medications m_1 and m_2 and patient p, if m_1 and m_2 are prescribed to p then m_1 and m_2 do not have an undesired interaction.

The transitions for adding an interaction and adding a prescription must have appropriate preconditions to ensure that the above safety properties are always satisfied. Namely, prescription of a mediation for a patient must only be allowed if the newly prescribed medication does not have an undesired interaction with any of the patient's existing prescribed medications. Further, addition of a pair of medications as having undesired interaction must only be allowed if the two medications are different, and that they are not both prescribed to any patient already.

The TLA+ model of the EHealth system is used as reference for modelling the system in the other languages studied in this work. I had written the TLA+ version, as a completed extension of the snippets in Prof. Ostroff's technical report [64], based on the descriptions there. As TLA+ does not have **constructors** for partial functions, the model uses relations for representing the prescriptions and interactions data. Although we wrote the transitions such that the relations represent partial functions, we did not impose any additional constraints enforcing this.

The B and DASH models of the EHealth system were written by Ali Abbassi and Jose Serna respectively. Compared to TLA+ where there are no **type signatures**, B has type signatures with automatic **typechecking**. The B model of the EHealth system was incorrectly written to use the constructor of a total function representation for both prescriptions and interactions data over the set of all patients and all medications in the universe. This results in the invariants being violated from the very beginning in the initial snapshot.

Similar to the B EHealth model, the EVENT-B model also has **type signatures** with automatic **typechecking**. In contrast to the B model, the EVENT-B version correctly uses type signatures that are consistent with the use of data in the model. The EVENT-B model uses the **constructor** a partial function representation for prescriptions and interactions data over only the set of people and medications that have been added to the system. This representation establishes the **invariants** in the initial snapshot and preserves them in future snapshots.

The notable difference between the DASH EHealth model and the reference TLA+ version is the use of DASH's **monitored variables** for the input variable declarations, by marking the input patient and input medications snapshot variables with the `env` keyword.

The first notable difference between the ALLOY and TLA+ models of the EHealth system is that in the ALLOY model, we represent the input variables as snapshot variables prefixed with "`in_`" that are never constrained in the destination snapshot. Due to the nature of the **frame problem** in ALLOY, this effectively makes these variables **monitored variables**. Although one could use a similar approach in TLA+, it makes little sense to do so, because in TLA+ every snapshot variable must be constrained in every transition, and making the input variables part of the snapshot would make the transitions needlessly more verbose. In fact, this is the reason why all of our ALLOY models for the case studies are the largest, because in ALLOY all (non-monitored) snapshot variables that are to remain unchanged by a transition must be constrained by that transition. The second difference between the two models is that in the ALLOY model, the predicate for every transition requires the source and destination snapshot of the transition as arguments. However, in the TLA$^+$ model, the arguments to each snapshot are only the input variables, as TLA$^+$ has built-in support for source and destination snapshots using unprimed and primed variable names, and the packaging of the **snapshot** is implicit; whereas in ALLOY, unprimed and primed variables do not have any significant meaning in the language, and the packaging of the snapshot is an explicit signature.

The PLUSCAL model of the EHealth system closely resembles the TLA$^+$ version, with many common/overlapping TLA$^+$ expressions between the two models. Where the TLA$^+$ model used a TLA$^+$ action (definition) directly for defining each transition, in the PLUSCAL version each `process` corresponds to one transition. Since the `process`es each contain a single PLUSCAL label, the translation of the PLUSCAL model back to TLA$^+$ results in one TLA$^+$ transition per each PLUSCAL `process`.

The ASMETAL EHealth model, like the reference TLA$^+$ model, uses a relation representation for the prescriptions and interactions data because of ASMETAL's lack of a partial function **constructor**. This model is one of two ASMETAL models among our case studies for which we were able to use the AsmetaSMV model checker, because as previously mentioned, AsmetaSMV currently only supports a limited subset of the ASMETAL language, and in this model we were able to confine ourselves to using the limited subset of the language. For instance, use of `Powerset` in **type signatures** for declaring set-valued snapshot variables is not supported by the model checker, so we did not use it in this model. To work around the limitation, we used membership predicates for representing membership in the sets and functions in the type signatures of the snapshot variables, as shown in Figure 7.1.

```
controlled patients: Patients -> Boolean
controlled medications: Medications -> Boolean
controlled interactions: Prod(Medications, Medications) -> Boolean
controlled prescriptions: Prod(Patients, Medications) -> Boolean
```

Figure 7.1: Membership predicate representation for sets and functions in ASMETAL

For the EHealth case study, we found all languages and their tooling to be adequate and useful for modelling and debugging the system, as there is little structure in the control aspects of the transition system, and the **primitives** and **constructors** used in the example are fairly standard across declarative modelling languages. For the ASMETAL EHealth model, though the need to use a membership predicate representation for sets just so we could use the AsmetaSMV model checker did not pose a serious issue in our modelling and verification process, we did find it to be an annoyance and deviation from the reference model nonetheless. More importantly, ASMETAL does not have a **constructor** for partial functions. Further, while writing the ALLOY model, we found ourselves running into strange behaviours with our model, which we eventually noticed was due to under-specification in some of our transition definitions, relating to the **frame problem**, where in ALLOY any variable not constrained in a transition may freely change from the source to the destination snapshot. This may be a common mistake by modellers of varying levels of experience, and may be hard to debug due to it causing unexpected behaviour and/or **inconsistency** in various parts of the model. As such, we would be less inclined to recommend ALLOY for modelling this example.

7.2 Digital Watch

The second case study in this thesis is the digital watch example, an adaptation of Harel's model [38]. The digital watch system consists of a digital display and four buttons *a*, *b*, *c*, and *d*. The watch has several modes/states of operation that may be accessed using specific sequences of button presses. The digital watch system has a large number of **control states** and many **events** triggering transitions between them, and depends on control state hierarchy to decompose these into related parts. Considering this, and the model's limited use of **primitives** and **constructors**, the digital watch example falls on the very control-oriented end of the data- vs. control-oriented characterization spectrum. For the digital watch system, we consider the following reachability property: from any snapshot, when *a* is pressed, it is possible to get to a snapshot in which in the next snapshot the watch will be in the Time state.

The reference model for the digital watch system is written in DASH by Jose Serna [12]. The model makes extensive use of DASH's `state` construct to hierarchically decompose the watch's behaviour. The model's **control state hierarchy** consists of two concurrent regions, one representing the state of the watch's display light, and the other representing the current active display. It also uses DASH's environmental (**monitored**) **events** for representing the button presses of the watch and passage of time, to enable transitions. The above reachability property does not always hold in the DASH model and is not an invariant of the system, as expected.

The B and TLA$^+$ models of the digital watch system were written by Ali Abbassi and myself respectively. Since **control state hierarchy** and **events** are not supported by any of the modelling languages studied in this work besides DASH, we model these features using other constructs available in each language. For control state hierarchy, we model all transitions flatly (*i.e.* no hierarchy), which required extra care to ensure that the preconditions of the transitions of the flat model are written correctly, so that each transition in the flat model is only enabled if and only if its corresponding transition in the hierarchical model is enabled. This involves making sure that the hierarchical and concurrent states of the reference model are faithfully represented in the non-hierarchical versions. We use one variable for representing each concurrent region of the reference model, ensuring that the concurrent regions can change independently of each other. For **events**, we model each using a combination of a variable keeping the triggered state of the event, and a transition (or pair of transitions) to change that variable. We allow events to persist arbitrarily long into the future. The advantages of these approaches include the fact that the flat model closely resembles the original hierarchical model. The disadvantages include that extra care needs to be taken to make sure that the preconditions for each transition are set up correctly to faithfully match the behaviour of the transition in the hierarchical model as closely as possible. In the absence of DASH's per-`state` **namespaces**, we use a naming convention for the names of transitions to highlight the state of the reference model each transition belongs to. The above descriptions apply to the remainder of the digital watch models in all of the languages (besides DASH).

The behaviour of the B model by Abbassi does not fully match the earlier descriptions, particularly with respect to **events**. In Abbassi's model, the transitions modifying the display state also change the triggered state of the events. However, these two belong in separate concurrent regions, and doing so effectively ties together the two concurrent regions of the reference model. This results in a special-case behaviour not necessarily guaranteed by the original model.

One difference between the TLA$^+$ and EVENT-B models of the digital watch system and the rest of the models is that for the pressed/released state of the four buttons, we used a total function that maps each button to a Boolean value, instead of four separate Boolean variables. This allows for a more compact representation of **events**, with only one or two transitions that could change the state of any of the events, as opposed to requiring one or two transitions per each event.

In the ASMETAL digital watch model, we represent **events** as **monitored variables** (using the `monitored` keyword) which may change freely from one snapshot to the next. This eliminates the need to include transitions for toggling the variable(s) representing the triggered state of the events. As discussed in Chapter 4, ASMETAL does not support defining the **transition relation** as a disjunction of multiple transitions. To write a transition relation that would take a randomly-chosen transition each time, we have to declare an enumerated set with each element corresponding to one transition, use the `choose` rule in the definition of the transition relation to choose an element from that set, and use a `switch` with a `case` for each transition, executing the transition corresponding to the chosen element of the enumerated set. This is especially cumbersome and error-prone in a model such as the digital watch system with many transitions, as the modeller may forget to update the `switch` cases when adding or removing transitions. The ASMETAL digital watch model is the second of the two ASMETAL models among our case studies for which we was able to use the AsmetaSMV model checker, finding a satisfying instance for the reachability CTL property.

The PLUSCAL digital watch model closely resembles the TLA$^+$ version, with many common TLA$^+$ expressions between the two models. These include the reachability property written as a TLA$^+$ expression. Where the TLA$^+$ model used a TLA$^+$ action (definition) directly for defining each transition, in the PLUSCAL version each `process` corresponds to one transition. Since the `process`es each contain a single PLUSCAL label, the translation of the PLUSCAL model back to TLA$^+$ results in one TLA$^+$ transition per each PLUSCAL `process`.

For the digital watch case study, we found DASH's unique **control state hierarchy** and **events** constructs to be powerful and convenient abstractions for modelling the various states/modes of the control-oriented digital watch system. We thus recommend DASH as the most suitable language for this example.

7.3 Musical Chairs

The musical chairs case study (originally in [62]) resides near the middle of the data- vs. control-oriented characterization spectrum. The musical chairs example is a game consisting of a number of players and chairs. At any time during the game, there is always exactly one less chair than there are players. In each round, the players circle the chairs while music plays. The music is then stopped abruptly, and the players have to scramble to sit on a chair. The one player who did not manage to sit down is eliminated. The music then resumes, and the next round begins. This process is repeated until only one player remains, dubbed the winner.

The snapshot variables in a musical chairs model consist of two sets for active players and active chairs, and a mapping from chairs to players for keeping track of the occupied chairs. Further, the current mode of the game (players are sitting, walking, *etc.*) is represented either as an enumerated set or using control states. The status of music (playing or paused) is represented either using events for control states, or as a snapshot variable and transition(s) changing it.

For the musical chairs case study, we consider the following safety and liveness properties:

- The number of players is always one greater than the number of chairs.
- It is possible that a specific player named Alice wins.

- The game will always eventually be in the “sitting” state.
- The number of active players always eventually reaches one and remains at one.

The reference model for the musical chairs example is written in DASH by Jose Serna. The model uses DASH’s **control state hierarchy** for representing the different modes of the game. Further, two environmental (**monitored**) **events** `MusicStops` and `MusicStarts` are used to model the abrupt stopping and starting of music. The above four properties were verified to hold for the DASH musical chairs model.

The DASH model of the musical chairs example used as the reference for the other models in this work was itself modelled after an ALLOY musical chairs model [32] by Sabria Farheen. A notable difference between this ALLOY model and the DASH model of this system is the absence of **control states** and **events** in ALLOY. Thus, the current mode of the game is represented using a **snapshot variable** with its value a member of an enumerated set, and the starting and stopping of music are modelled as transitions that directly change the snapshot variables, including the mode of the game. The above four properties were included and verified to hold for the ALLOY musical chairs model as well.

Similar to the ALLOY musical chairs model, the B musical chairs model [12] by Ali Abbassi uses a snapshot variable for representing the mode of the game along with two snapshot variables keeping the sets of active players and active chairs, and an occupancy mapping between the two. There are some notable differences between the two models. Namely, the B version uses the **constructor** of a total function in the **type signature** of the occupancy mapping, and has two extra variables keeping the player and chair to be eliminated at the end of the current round. Further, an `Assign` **constant** predicate is defined and used to constrain the occupancy mapping according to the given sets of chairs and players. Also, there is no snapshot variable keeping the status of music explicitly, and there are no separate transitions dedicated to starting and stopping of music. Instead, the effects of starting and stopping the music are carried out in the main transitions of the system. This choice of representation makes the B musical chairs model more different than the models of this system in the other languages, since starting and stopping of music is not a standalone **event** in this model, and is *always* accompanied by a change/progress in the game. These changes are *not* inherently necessary, and it is possible to model the system in B without them.

The differences between the TLA$^+$ and DASH musical chairs models stem from the absence of **control states** and **events** in TLA$^+$. As with the previous TLA$^+$ models, we used a snapshot variable for tracking the current state of the game, corresponding to DASH’s use of **control state hierarchy**.

This makes the TLA$^+$ model more similar to the ALLOY version. However, in contrast to the ALLOY model, for the TLA$^+$ model we used a Boolean variable for modelling the status of the music, and used a separate transition `ChangeMusicPlaying` for changing the status of music. For this reason, along with the fact that we are checking liveness properties for the system, we impose fairness conditions on the transitions [54, 20, 46]. As such, we added weak fairness (justice) on the `ChangeMusicPlaying` transition and strong fairness (compassion) on the rest of the transitions, so that the firing of the music change transition would not starve the main transitions.

The **expressions** of the ASMETAL musical chairs model are less declarative and more imperative than our other musical chairs models, because in ASMETAL we cannot constrain the destination snapshot value of the occupancy function in a declarative manner, as we would in the other languages. We could use ASMETAL's set comprehension notation if the occupancy mapping was a relation; but we cannot use a formula or constructor to further constrain that set of pairs to be a function in a declarative way. Instead, for filling up the occupancy function when the music stops, we have to use a helper transition to update the occupancy mapping for each chair one by one using a `forall` rule over the active chairs and a nested `choose` rule over the active players (excluding the loser of the round). A reasonable idea may be to keep the **type signature** of the occupancy function as `Powerset(Prod(Chair, Player))`, but this will not work because gradually updating the occupancy mapping using the `union` function is seen as assigning multiple different values to the same snapshot variable, and thus an inconsistent update. Therefore we have to change the **type signature** to `Chair -> Player`. We will not be able to easily extract all the information we need from a variable with this type signature; for instance we need to introduce a separate variable to keep track of the number of occupied chairs. Due to these limitations, the ASMETAL musical chairs model has tedious imperative-style operations for updating the function-valued snapshot variables.

The PLUSCAL musical chairs model for the most part resembles the TLA$^+$ version, with many common TLA$^+$ expressions between the two models. Aside from the inherent differences between PLUSCAL and TLA$^+$ in terms of defining transitions, the two models also differ in how they constrain the occupancy function: Assignment to **snapshot variables** in PLUSCAL is done only using the `:=` assignment operator and the constraints need to be described all at once, whereas in TLA$^+$'s declarative approach, the modeller describes how each snapshot variable changes, using one or more conjuncts. Also related, in regular TLA$^+$ **expressions** we can always refer to the source snapshot value of a variable in a transition using its unprimed name, whereas we cannot do so in PLUSCAL if the variable is assigned to in the current transition. In PLUSCAL, we can declare a variable local to the **namespace** of that `process` and initialize it with the source snapshot value of that variable, and use the local variable where needed in the process (transition) body.

The EVENT-B musical chairs model uses the language's rich arrow **constructors** for various functional and relational units of data in **type signatures** as well as snapshot variable assignments. These arrows, when combined with the :| nondeterministic assignment operator, are a powerful way of writing terse and concise descriptions of functions. We declare the occupancy mapping to be a partial injection from chairs to players, and in the `Sit` transition we assign to the occupancy function a total injection from the sets of active players to active chairs, enforcing the two criteria that each chair can have only one player sitting on it, and each player may only sit on one chair.

For the musical chairs case study, we found DASH's **control state hierarchy** and **events** to lend themselves well to modelling this system. The control states and their hierarchy are used to concisely and precisely represent the current state/mode of the game, and DASH's environmental (monitored) events are a natural candidate for modelling the starting and stopping of music. Since these features are unique to DASH, we believe they give DASH an advantage over the other languages. After DASH, we found B and EVENT-B's extensive set of arrow **constructors** to be convenient and powerful tools for writing concise **type signatures** and constrains on functional and relational data units.

7.4 Library Management

The library management case study (originally by Frappier *et al.* in [33, 34]) resides at the very data-oriented end of our data- vs. control-oriented characterization spectrum. The library management information system has a set of members and a set of books available in the library. Members can join and leave the library, and books may be acquired or discarded by the library. Each member can borrow a certain maximum number of books and return any of them when they wish. Members can also reserve a book if it has already been lent to another member. Doing so will add the member to a wait list for the book, and they can pick up the book when it becomes available to them. They can also cancel their reservation at any time if they wish to. Members may renew any of their borrowed books if no other member has entered the reservation waiting list for that book. A member may only leave the library (relinquish their membership) when they are not borrowing or reserving any of the library's book, and a book may only be discarded by the library if it is not loaned to or reserved by any members.

The snapshot variables in the library management system consist of two sets for acquired books and registered members, and two mappings, one for keeping track of the books loaned to members, and the other for tracking the reservations placed for the currently-unavailable books. This mapping should store the order in which reservations were placed, so as to allow a pickup only by the member who placed the current oldest reservation for that book.

The reference model for the library management case study is in B by Frappier *et al.* [34]. The model uses the **constructor** of a partial function from books to members for keeping track of the lent books, and a total function from books to an injective sequence of members for tracking the reservations. A number of safety and liveness properties are included in the reference B model, such as:

- A book may be reserved only if it has been lent or already reserved by another member.
- A book cannot be lent to a member if it is reserved.
- A member is allowed to pick up a reservation only if their reservation is the current oldest.
- Ultimately, a member can leave the library. (CTL property, not expressible in LTL)

In addition to the B library management model used as reference for our subsequent models, the above paper by Frappier *et al.* also contains an ALLOY model of the system. This model, though written well before Farheen wrote her style guidelines [32], adheres to several of Farheen's recommendations promoting structure, including decomposing each transition definition into two separate predicates, for preconditions over the variables in the source snapshot and postconditions over the variables in the source and destination snapshots, respectively. With respect to the **frame problem** in ALLOY, the modellers also took the extra step to put the no-change formula for each snapshot variable into a separate predicate, and use those predicates in the transition definitions instead of repeating the constraints. We believe this may be excessive, given the simplicity and triviality of such constraints.

The EVENT-B library management model is in many ways similar to the reference B model, considering the closeness of the two languages. A notable difference between the two is the absence of a sequence **constructor** in EVENT-B. As such, we had to use a partial injective function from the set of natural numbers to members to model the injective sequence of members from the reference B model. EVENT-B's lack of built-in sequences and operations on them was particularly inconvenient when writing the `Reserve` transition, where we need to update the reservation mapping for the given book and insert the new member with the correct index in the book's reservation function.

The TLA$^+$ and PLUSCAL models of the library management system are similar to the reference B model in their use of a partial function for representing the loans mapping from books to members. They differ from the B model in that the TLA$^+$ and PLUSCAL models do not have **type signatures** and **constructors**, and use a formula definition in the typing constraint to express that `loans` should be a partial function. However, since all TLA$^+$ functions are by default total, it may not be obvious to a modeller how to specify a partial function. To declare a partial function, we first add a helper definition `PFun(S, T) == UNION {[s -> T] : s \in SUBSET S}` describing all partial functions from `S` to `T`, where the domain could be any subset of `S`. We can then use this predicate to write a constraint such as `loans \in PFun(books, members)`, making `loans` a partial function from `books` to `members`. There are other possible representations and constraints for `loans`, such as `[books -> SUBSET members]` and `[books -> members \union NULL]`, but aside from diverging from the reference specification, the other disadvantages of using these include adding unnecessary verbosity to the model and requiring special treatment of the `NULL` constant throughout the model, respectively.

Modelling the library management system in PLUSCAL and checking it using the TLC model checker helped expose a bug in one of the properties from the reference B model, due to incorrect assumptions about operator precedence and associativity in B along with using $=$ instead of $\neq$.

With respect to the criterion of syntax of **expressions**, the ASMETAL library management model is more verbose compared to our other models of this system, due to ASMETAL's rare use of symbols and most operators having long-form textual names. However, in terms of transitions and **type signatures** it is very similar to our other library management models. Since ASMETAL transitions are always enabled and there is no keyword for checking whether a transition is enabled, we factored out the conditions of the top-level `if` rule for each transition into separate `derived` functions. We then used these functions in the properties to check when each transition can be taken.

The DASH model of the library management system is fairly similar to the reference B model. We were able to translate and check the properties from the original B model to DASH, as they all happened to lie in the set of properties expressible in both LTL and CTL. Additionally, as DASH's tool support supports CTL model checking, we were able to express in DASH one of the CTL properties that was commented out in the original model and not expressible in LTL.

For the library management case study, we found the `Cancel` and `Reserve` operations to be the most challenging to model correctly across the languages. This is because these operations require correctly updating or overriding the entry corresponding to a book in the reservations mapping, where each entry itself is an ordered collection of members. This has to be done while preserving the structural correctness of the underlying data unit. This was hardest to do in EVENT-B, TLA$^+$/PLUSCAL, and ASMETAL. For EVENT-B, the absence of a built-in sequence **constructor** and supporting operations was what made this challenging. For TLA$^+$/PLUSCAL, even though TLA$^+$ has built-in sequences, we found the available operations for correctly modifying sequences to be lacking for our purposes in the **expressions**. Thus, the challenge was writing correct custom definitions for operations on functions and sequences. We found that ASMETAL's more imperative-style **expressions** make modelling this system more challenging. Namely, the lack of certain declarative operations and the difficulty of writing a **transition relation** that would take a randomly-chosen transition, described earlier. As such, we recommend B, DASH, or ALLOY for modelling this system. We would recommend exercising caution while using ALLOY, due to the dangers of under-specification relating to the **frame problem**.

7.5 Railway Scheduling Deadlock Freedom

The railway scheduling deadlock freedom (railway for short) case study originally by Mazzanti *et al.* in [59] resides near the middle of the data- vs. control-oriented characterization spectrum. The railway system is a model of a yard consisting of eight trains and their missions. The yard contains a number of critical sections of railway tracks that the scheduling algorithm must ensure are never saturated, which would cause a deadlock in the system where no train could proceed. Briefly, this is done by keeping track of the number of trains currently present in each critical section, and making sure that number is always below a certain maximum that would not cause the system to reach a deadlock. For the railway case study, the property we are interested in is whether all of the trains always eventually reach their final destination, *i.e.* the last station on their mission. For a model to satisfy this property, it must be accompanied with appropriate fairness constraints.

The railway system was originally modelled in UMC, Promela/SPIN, NuSMV, mCRL2, CPN Tools, FDR4, and CADP. Thus, for our first model of this system that we wrote (in B), we based our work mostly on the descriptions in the original paper, occasionally comparing our model against the paper's accompanying NuSMV and Promela models for clarification. For the subsequent models of the system, we used our B model as the reference.

The B and EVENT-B railway models use **type signatures** to specify what the precise ranges of the valid natural numbers of the model's **primitives** are. Compared to the models accompanying the original paper, our B and EVENT-B models use a more compact representation for the train missions and critical section constraints. Instead of declaring eight separate snapshot variables one for each train, we use the **constructor** for a total function with the domain `0..7`. This allows accessing the missions and constraints using the index for each train, alleviating the need for writing eight separate `move_train` transitions, one for moving each train. Instead, the B and EVENT-B models each have only one transition for moving all of the trains, which indexes into the relevant snapshot variables using the currently chosen train `ct` as the index. The chosen train index, represented by the `ct` snapshot variable, is updated in a round-robin fashion each time the `choose_train` transition is executed.

The ASMETAL model of the railway system is similar to the reference B version. The notable difference is ASMETAL's distinction between **controlled** and **monitored** variables. We designate the snapshot variable for the currently chosen train as a `monitored` variable, eliminating the need for having a `choose_train` transition.

The ALLOY railway model differs from the reference B model in a number of ways. These differences stem from the fact that ALLOY's **built-in** integers are highly inefficient in analysis (it is not a modelling language problem; see Section 5.1 and Section 5.7), and increases in **scope** of integers results in them quickly becoming a performance bottleneck during the analysis. As such, instead of using integers for modelling trains and stations, we use an enumerated set for each, allowing us to avoid increasing the scope of integers. The downside of this approach is that it makes writing the model more tedious, and drastically increases the line count for the ALLOY model, as we can no longer use only one transition capable of moving any of the trains, and have to write one transition per each train.

The PLUSCAL and TLA$^+$ models of the railway system differ from our reference B model due to the absence of **constructors** and **type signatures** in TLA$^+$ and PLUSCAL. The PLUSCAL and TLA$^+$ models are very similar to each other. Both models have two transitions, one for moving any train, and another for choosing the train to be moved, with strong fairness imposed on the former and weak fairness on the latter. This helps make sure the transition for changing trains does not starve the transition for moving a train.

The DASH railway model is similar to the ALLOY model, differing from the reference B model in their use of an enumerated set representation instead of integers for the train stations. Aside from the inherent differences between DASH and ALLOY, the two models differ in their use of ALLOY's more recent features in its **expressions** language. Namely, DASH does not yet support ALLOY's **`let`** expressions and the **`enum`** short-hand for declaring an enumerated set. The other main difference between the two models is that in the DASH model, we use the **`env`** keyword for the chosen train snapshot variable to mark it as a **monitored** variable, allowing the environment to choose which train to move next. This lets us eliminate the `choose_train` transition in favour of a more general train-choosing mechanism. However, similar to the ALLOY model, the DASH railway model is noticeably larger than the other models of the railway system, as we have to write one transition per each train, for the same reasons as the ALLOY model, explained earlier. While writing the DASH model, we ran into a **Contradictory** *TR* issue stemming from DASH's use of ALLOY's ordering module by default and the scope we set on the `Snapshot` signature not matching exactly the number of snapshots needed for the snapshot trace of the property we were checking. We resolved this problem by using a scope exactly matching the number of needed snapshots.

For the railway scheduling deadlock freedom case study, the limitations of integers in ALLOY really shows during analysis, and needs to be worked around if possible. For the ALLOY and DASH railway models, we kept the **scope** of integers at a minimum by using enumerated sets instead of integers for representing the 27 train stations. We believe that B, EVENT-B, TLA$^+$, PLUSCAL, and ASMETAL are better equipped to handle this model's use of integers. DASH and PLUSCAL in particular distinguish between controlled and **monitored** variables, allowing the modeller to denote the snapshot variable for the currently chosen train as a monitored variable changed by the environment. This eliminates the need for a transition for changing the variable.

Chapter 8

Related Work

In this chapter, we discuss related work on comparing declarative modelling languages.

The Software Abstractions book [43] includes an appendix that discusses alternative modelling languages to Alloy. Jackson compares Alloy to four other formal frameworks (B, OCL, VDM, and Z) by using a simplified version of one of the running examples used earlier in the book and modelling the example in the other four languages. Jackson's comparison consists largely of discussing the historical background as well as tool support for the languages, and goes into little detail about the constructs of the language or a comparison criteria for comparing the languages.

Newcombe [61] compares TLA$^+$ and ALLOY, and concludes that TLA$^+$'s data operations, together with its higher-order and recursive operators, make descriptions simpler than ALLOY's for engineers, particularly with respect to nested record structures. Our comparison looks in detail at the constructs and semantics of the transition system representations across the languages, whereas Newcombe focuses on the usage of the languages and their tool support, as well as more on non-functional criteria such as each language's minimization of cognitive burden on the modeller, high return on investment, and the handling of subtle or complex problems.

Zave [77] compares ALLOY and Spin by using them to model a complex system, the Chord network protocol for a distributed hash table, with the goal of recommending a lightweight formal method for modelling and analyzing complex network protocols. Zave's comparison includes a few common criteria with ours, such as implicit vs. explicit snapshot representation, building blocks for control and data aspects of the model, scope (size bound) of sets, and the frame problem. Through modelling and analyzing Chord in ALLOY, Zave arrives at the surprising result that Chord is not correct, as it was previously thought to be. Zave notes that although several previous work had reported on the application of model checking to implementations of Chord, due to implementations being immensely more complex than an abstract model, the analyses were necessarily incomplete. Although those previous works were able to find bugs in the implementations they had analyzed, none of them had found the specification-level bugs that Zave found. In terms of differences between ALLOY and Spin, Zave notes that because the Chord specification did not include a strong global invariant, she had to resort to using Spin to find a provably correct global invariant. Otherwise, Zave notes, the presence of a sufficiently strong global invariant would have given ALLOY a performance boost sufficient to make it the 'clear winner' of the comparison (by using the invariant to restrict the model's reachable snapshot space).

Sullivan *et al.* [74] compare three snapshot modelling techniques for transition systems in ALLOY for performance, using four examples. They compare two commonly-used snapshot representation techniques with a new "parameterization" approach, in which all signature fields are removed from signature declarations, and are instead added as arguments to all predicates, and fact blocks are converted to a predicate, referenced in other predicates where needed. Their results suggest that the Alloy Analyzer exhibits improved performance on examples using their new parameterization technique. In contrast to Sullivan *et al.*'s work, the focus of our comparison is more generally on the techniques for modelling transition systems across modelling languages, and we do not consider the performance of tool support for languages as part of our comparison criteria.

Frappier *et al.* [34] compare six model checking tools (ALLOY, CADP, FDR2, NuSMV, ProB, and Spin) for validation of specification of information systems. Their comparison focuses on the ease of specifying behaviours, properties, and the number of instances that can be checked by each tool for information systems. They present the requirements for a library management system to be modelled and checked with the above tools and languages. Their paper is the original source for our library management case study (Section 7.4). Frappier *et al.* conclude that a suitable language for validating information systems using model checking should support a notion of snapshots and transitions, and that process algebraic operators would be useful for easily expressing information system scenarios. Further, they conclude that CTL is powerful enough for expressing the kinds of properties they are interested in, whereas LTL falls short of expressing some of the properties.

Deutsch *et al.* [30] study the specification and verification of data-driven information systems, particularly an interactive web application that receives input from the user, potentially interacts with a database, and displays output to the user. They verify various CTL and CTL* properties about their model in an ASM-like specification language based on an earlier work [72].

Fraikin *et al.* [33] compare the two specification languages B and EB^3, representative of the snapshot-based and event-based modelling paradigms respectively, for modelling information systems. They compare the two languages by modelling a library information system in each language, and comparing the models on the four criteria of ease of specifying functional behaviour, validation, verification, and evolution of a model to meet new requirements. The authors conclude that the two languages are complimentary. While B seems better at expressing complex ordering and invariants, EB^3 provides a simpler, modular, explicit representation of temporal properties. The downside of EB^3's process algebra is the difficulty of proving preservation of invariants, since it does not include an explicit representation of pre and postconditions for transitions, while B's drawback is the difficulty of understanding the orderings of input events, due to the strong data coupling between transitions.

Mazzanti *et al.* [59] study and report on the use and diversity of formal methods in railway systems. They do so in part by modelling and verifying a train scheduling deadlock avoidance algorithm in seven formal environments (UMC, Promela/SPIN, NuSMV, mCRL2, FDR4, CPN Tools, and CADP) from three families of languages (state-machine-oriented, process algebras, and Petri Nets) and comparing their findings. The authors observe that even small decisions in the design and verification process can result in drastic changes in tooling performance, especially for the studied process algebraic approaches, but also notice that a model can be ported to the other environments with limited effort. The authors believe this helps propel the new idea of formal methods diversity, in which modellers port their model to several potentially non-certified formal tools to increase their confidence in the correctness of the results of their analysis and verification. Their paper is the original source for our railway scheduling deadlock freedom case study (Section 7.5).

Aydal *et al.* [22] compare four tools (USE, Alloy Analyzer, ZLive, and ProZ) for three modelling languages (UML, ALLOY, and Z respectively) with respect to strengths and weaknesses of the tools for analyzing and verifying the models written in each language. In particular, Aydal *et al.* study and compare each tool against one another on a course assignment system example, with four comparison criteria of animation of models, generation of pre and postconditions for transitions, information reported about the analysis, and the required expertise. They also report on the performance and efficiency of the tools on their models.

Ardis *et al.* [21] compare seven formal languages (Modechart, VFSM, Esterel, Basic LOTOS, Z, SDL, and C) on the same example, a telephone switching system. None of the languages they compare are in the set of languages we compare in this work. The criteria that Ardis *et al.* used to compare the languages are much higher-level concerns, such as testability and language maturity, than our comparison, which focuses on how a transition system model is modelled in a declarative modelling language.

Huynh *et al.* [41] formalize SCAG, a new healthcare access control model with conflict resolution for managing each patient's wishes as to who can access their Electronic Health Records (EHR). The access control model takes into account regional laws and regulations applicable in Québec and Canada, where under certain strictly defined scenarios, for safety reasons patient consent can be overridden to protect the patient's life. The authors formalize the access control policies in B and ALLOY, and use ProB and the Alloy Analyzer to verify properties about their models. The results show that without any optimizations, the Alloy Analyzer performs better than ProB. However, when using ProB's capability for controlling the order of constraint solving, as well as storing frequently-used results in snapshot variables of the B model, ProB performs significantly better than Alloy for all of the checked properties.

Bruel *et al.* [26] survey the role of formalism in system requirements, discussing and comparing more than twenty approaches to requirements. They classify the studied approaches into fives categories of natural language, semi-formal, automata and graphs, mathematical, and seamless, based on how they express requirements. The authors then use a running example of the Landing Gear System [24] to compare the approaches with respect to nine criteria including level of abstraction, tool support, separation of the external environment and the system, and verifiability of requirements. Bruel *et al.* present a detailed and lengthy discussion of their results and conclusions; but in short, they conclude that formal methods complement other (potentially informal) requirement approaches, and that they should be seen as powerful tools available to every requirements engineer or business analyst.

Leuschel *et al.* [56] demonstrate the use of B as a high-level constraint modelling language, with the ProB tool as the constraint solver. They compare B and ProB with ALLOY and the Alloy Analyzer on several examples, showing the strengths and weaknesses of each tool on the examples. The authors note the ProB solver's weaknesses compared to the Alloy Analyzer on certain relational operators such as relational image and transitive closure, citing improvements to these as future work. Leuschel *et al.* conclude that using B with ProB can be a nice trade-off between the high performance of low-level constraints solvers and the high difficulty of encoding problems in them, and the increased computational power needed for solving problems encoded in higher levels of abstraction.

Samia *et al.* [66] compare the use of a high-level specification language and tooling such as B and ProB with a low-level language and tooling such as Promela and Spin. They do so by modelling ten examples in each language and comparing the models on three criteria of model length, modelling time, and model checking performance.

Hatcliff *et al.* [39] survey designing and using behavioural interface specification languages. They compare several specification languages — including JML, SPARK, Spec#, and Dafny — on several examples. Hatcliff *et al.* focus on program verification and languages that allow verifying a program satisfies certain properties, while our comparison focuses specifically on modelling transition systems in declarative modelling languages, at a higher level of abstraction than typical programming languages.

Maraee and Sturm [58] examine the usage of the OCL declarative language with that of the Java imperative programming language for understanding and developing constraints in a controlled experiment among a group of undergraduate students. The obtained results suggest that using a declarative language such as OCL has many advantages for understanding and developing constraints over an imperative language such as Java, with the differences increasing as the constraints grow more complex. Writing constraints in a declarative language enables verification and validation of models, whereas writing them in an imperative language inevitably shifts the focus to low-level implementation details that are irrelevant to the essence of the constraints.

Lamport and Paulson [52] compare specification with and without a type system, stating advantages and disadvantages for both typed and untyped specification languages. They compare set theory and typed formalisms across four general criteria of flexibility, convenience, pitfalls, and abstractness; and conclude that an untyped set theory could serve as a solid, common foundation upon which different tool-specific type systems can be overlaid.

López *et al.* [57] propose a formal framework for assessing the expressivity of formal specification languages, based on the mutation testing technique. The idea is to create mutated versions of a specification that differ from the original specification in some aspect, with some of the mutants still exhibiting correct behaviour and some behaving incorrectly. Then, model the mutants in the specification language under study. The authors' measure of expressivity and suitability of languages for modelling a certain class of systems is based on the ratio of the correctness of mutants and the correctness of the models of those mutants in the given modelling language. In other words, whether the models of the correct and incorrect mutants are distinguishable in the modelling language.

Some textbooks [62, 17, 65, 23, 53, 23, 18, 63] introduce various formal and informal modelling languages, each with their own examples or occasionally the same example, to illustrate certain aspects of the language. However, they generally do not compare the languages against one another.

In comparison to these related works, we focus on models of transition systems. We develop a set of categorized comparison criteria and examine in depth each language with respect to each of the criterion. Further, we use these criteria to compare a diverse range of examples on the data- vs. control-oriented characterization spectrum, modelling each of the five examples in all of the seven languages, producing a total of thirty-five models. We use these models and our observations from carrying out the case studies to make recommendations as to which language(s) we think would be the best fit for modelling various kinds of transition systems.

Chapter 9

Conclusion

This thesis presents

- a set of criteria to compare declarative modelling languages;
- the comparison of the selected declarative modelling languages (B, EVENT-B, ALLOY, DASH, TLA$^+$, PLUSCAL, and ASMETAL) based on these criteria; and
- our recommendations for the choice of modelling language based on the characteristics of the transition system under description, rooted in our observations of the differences and similarities between the languages with respect to our comparison criteria from the several case studies we carried out.

We categorize our comparison criteria into three main categories of control modelling, data modelling, and modularity; discussed in detail in Chapter 4, Chapter 5, and Chapter 6 respectively.

Based on our experience modelling each case study presented across the languages, we now present more general recommendations for the choice of declarative modelling language depending on the characteristics of the system under description.

For models where fine control over the **transition relation** of the transition system by the modeller is desired, ALLOY, TLA$^+$, and ASMETAL are the most suitable languages, as they use an explicit representation for the transition relation.

For modelling control-oriented systems where the relevance of the **transitions** and their being enabled can be captured using (potentially hierarchical) **control states**, we believe DASH is a great choice for the modelling language. While our case studies showed that these systems can be modelled in any of the languages we studied, we found DASH's **control state hierarchy** and **events** to enable the modeller to model control-oriented systems in an abstract, concise, and convenient way. Also, DASH's per-`state` **namespaces** allow separation of names, while still allowing global communication between different states through a clear interface of fully-qualified names.

A common source of mistakes and **inconsistency** bugs in ALLOY models is the under-specification of behaviours, relating to the **frame problem**. In ALLOY, any variable not constrained in a transition may freely change from the source to the destination snapshot. This can result in the model exhibiting strange behaviours that could be especially hard to debug. TLA$^+$, also a highly declarative language, addresses this problem by requiring that every transition constrain every snapshot variable, either explicitly using primed and unprimed names of variables, or denoting the unchanged variables using the `UNCHANGED` keyword. For this reason, we believe TLA$^+$ is a better choice than ALLOY for modelling larger data-oriented models, as it eliminates an entire common, hard-to-debug class of problems using static syntactic checks.

For models where a clear distinction between the system and its surrounding environment is desired, DASH and ASMETAL both distinguish between **controlled** and **monitored** variables. DASH additionally supports environmental (monitored) **events** as well, which are those that are fired exclusively by the environment and not by any of the system's transitions.

For highly data-oriented systems, we found B and EVENT-B's extensive set of arrow **constructors** for various relational and functional units of data to be powerful tools for writing terse and concise descriptions of highly-constrained composite units of data, both in **type signatures** and in formulas. Although ALLOY/DASH using **multiplicities** and TLA$^+$/PLUSCAL through use of flexible and expressive TLA$^+$ **expressions** could describe the same set of constraints on data, B/EVENT-B allow this in a more convenient way. We found ASMETAL to be the least declarative and least flexible in this regard.

As B and EVENT-B are similar languages with the same roots, and EVENT-B is a successor of B, the choice between the two languages may not be immediately obvious. Compared to B, EVENT-B is a smaller and much simpler language. While B supports several kinds of relationships between machines, in EVENT-B the only kinds of relationships are a machine importing one or more contexts, and a machine refining another machine. Further, B has **built-in** constructors for sequences and trees while EVENT-B does not. On the other hand, B only has one arrow **constructor** for declaring a relation, whereas EVENT-B has three additional arrows for constructing specific kinds of relations; namely total, surjective, and total surjective relations. Further, EVENT-B has support for declaring **subtypes** and partitioning a set into multiple disjoint subsets, whereas B does not.

PLUSCAL brings to the table the power and expressiveness of TLA$^+$'s **expressions**, wrapped in a semantics geared more towards modelling multi-process concurrent and parallel algorithms, with additional **well-formedness** safeguards that make certain classes of bugs unrepresentable in a valid model in the language. ASMETAL, too, has semantics for modelling multi-threaded systems.

With respect to modularity, EVENT-B, DASH, PLUSCAL, and ASMETAL allow **data decomposition** of a model across multiple files. B, ALLOY, and TLA$^+$ additionally allow **control decomposition** of a model into subtransition systems across multiple files.

Although we observed subtle and noteworthy differences across the languages with respect to the **stuttering** criterion, it did not seem to affect the modelling of our case studies noticeably. Similarly, the representation of **snapshot variables**, **initialization**, **deadlock**, **contradictory transition postcondition**, **constants**, **scopes**, and **syntax overloading** did not have a significant effect on our process of modelling the case studies. However, these criteria are useful to acknowledge, helping the modeller see the differences between the languages with respect to these criteria, when moving from one modelling language to the next.

If we were creating an *ideal* declarative modelling language, it would include the following features and characteristics:

- a more declarative rather than imperative **expression** style, so as to stay true to the declarative behavioural modelling paradigm;
- a wide range of **constructors** for composite units of data, as well as **multiplicities** for the constructors, with constructors for commonly used data units such as total, partial, and surjective functions built as syntactic sugar on top of multiplicities; as well as support for declaring **subtypes**;
- implicit representation for the **transition relation** by default for convenience, along with options to supplement or tweak the transition relation by more knowledgeable modellers;
- improved tool support for helping the modeller find the sources of **inconsistency** in their model;
- notion of **control state hierarchy** or similar constructs to decompose a transition system from a control modelling point of view into **subtransition systems**, each having their own (uniquely addressable) **namespace**;

- **typechecking** done as a separate pass, with typing constraints restricting the reachable snapshot space for improving performance, as opposed to treating typing constraints as properties checked with other invariants during the main analysis;
- dealing with the **frame problem** in a way that is both convenient and avoids the issue of under-specification as seen in ALLOY models, by either adding implicit constraints that keep unconstrained variables in a transition unchanged (as DASH does), or by requiring that every variable either be constrained or marked as unchanged using a special keyword (as TLA$^+$ does); and
- a flexible module system for data decomposition of models across multiple files, with each file hosting one or more **parameterized** modules that support selective **import** and **export** of identifiers.

DASH already has several of the above characteristics and features, and we believe it has potential to become even closer to our described ideal language by implementing the missing features and enhancements from the above list.

In future work, we would like to extend our comparison to include other similar languages and also extend our comparison criteria to include an examination of tool support, which would address issues such as performance, robustness, and ease of understanding counterexamples.

References

[1] ALDB - A debugger for transition systems modelled in ALLOY. https://github.com/WatForm/aldb. [Online; accessed May 25, 2020].

[2] Asmeta - License. http://asmeta.sourceforge.net/download/license.html. [Online; accessed May 22, 2020].

[3] GNU General Public License, version 2. https://www.gnu.org/licenses/gpl-2.0.html. [Online; accessed May 22, 2020].

[4] ProB Licence. https://www3.hhu.de/stups/prob/index.php/ProBLicence. [Online; accessed May 22, 2020].

[5] Rodin Licence - Event-B.org. http://www.event-b.org/install.html. [Online; accessed May 22, 2020].

[6] The B-Toolkit - License. https://github.com/edwardcrichton/BToolkit#license. [Online; accessed May 22, 2020].

[7] What is free software? https://www.gnu.org/philosophy/free-sw.html. [Online; accessed May 22, 2020].

[8] OMG object constraint specification (OCL) specification. http://www.omg.org/spec/OCL/2.4/PDF, 2014. [Online; accessed June 15, 2018].

[9] OMG unified modeling language. http://www.omg.org/spec/UML/2.5/PDF/, 2015. [Online; accessed June 15, 2018].

[10] Xtext. https://eclipse.org/Xtext/, 2017. [Online; accessed June 15, 2018].

[11] StandardLibrary.asm. https://sourceforge.net/p/asmeta/code/HEAD/tree/asm_examples/STDL/StandardLibrary.asm, 2020. [Online; accessed May 18, 2020].

[12] Ali Abbassi, Amin Bandali, Nancy A. Day, and José Serna. A comparison of the declarative modelling languages b, dash, and TLA+. In Ana Moreira, Gunter Mussbacher, João Araújo, and Pablo Sánchez, editors, *8th IEEE International Model-Driven Requirements Engineering Workshop, MoDRE@RE 2018, Banff, AB, Canada, August 20, 2018*, pages 11–20. IEEE Computer Society, 2018.

[13] Abbassi, Ali. Astra: Evaluating Translations from Alloy to SMT-LIB. Master's thesis, David R. Cheriton School of Computer Science, 2018.

[14] Jean-Raymond Abrial. *The B-book - assigning programs to meanings*. Cambridge University Press, 1996.

[15] Jean-Raymond Abrial. *Modeling in Event-B - System and Software Engineering*. Cambridge University Press, 2010.

[16] Jean-Raymond Abrial, Michael J. Butler, Stefan Hallerstede, Thai Son Hoang, Farhad Mehta, and Laurent Voisin. Rodin: an open toolset for modelling and reasoning in event-b. *Int. J. Softw. Tools Technol. Transf.*, 12(6):447–466, 2010.

[17] Vangalur S. Alagar and Kasilingam Periyasamy. *Specification of Software Systems*. Graduate Texts in Computer Science. Springer, 1998.

[18] Vangalur S. Alagar and Kasilingam Periyasamy. *Specification of Software Systems, Second Edition*. Texts in Computer Science. Springer, 2011.

[19] Yamine Aït Ameur and Klaus-Dieter Schewe, editors. *Abstract State Machines, Alloy, B, TLA, VDM, and Z - 4th International Conference, ABZ 2014, Toulouse, France, June 2-6, 2014. Proceedings*, volume 8477 of *Lecture Notes in Computer Science*. Springer, 2014.

[20] Krzysztof R. Apt and Ernst-Rüdiger Olderog. Proof rules and transformations dealing with fairness. *Sci. Comput. Program.*, 3(1):65–100, 1983.

[21] Mark A. Ardis, John A. Chaves, Lalita Jategaonkar Jagadeesan, Peter Mataga, Carlos Puchol, Mark G. Staskauskas, and James Von Olnhausen. A framework for evaluating specification methods for reactive systems experience report. *IEEE Trans. Software Eng.*, 22(6):378–389, 1996.

[22] Emine Gokce Aydal, Mark Utting, and Jim Woodcock. A comparison of state-based modelling tools for model validation. In Richard F. Paige and Bertrand Meyer, editors, *Objects, Components, Models and Patterns, 46th International Conference, TOOLS EUROPE 2008, Zurich, Switzerland, June 30 - July 4, 2008. Proceedings*, volume 11 of *Lecture Notes in Business Information Processing*, pages 278–296. Springer, 2008.

[23] Dines Bjørner and Martin C. Henson. Logics of specification languages. *Monographs in Theoretical Computer Science. An EATCS Series*, Dec 2007.

[24] Frédéric Boniol and Virginie Wiels. The landing gear system case study. In Frédéric Boniol, Virginie Wiels, Yamine Aït Ameur, and Klaus-Dieter Schewe, editors, *ABZ 2014: The Landing Gear Case Study - Case Study Track, Held at the 4th International Conference on Abstract State Machines, Alloy, B, TLA, VDM, and Z, Toulouse, France, June 2-6, 2014. Proceedings*, volume 433 of *Communications in Computer and Information Science*, pages 1–18. Springer, 2014.

[25] Egon Börger and Robert F. Stärk. *Abstract State Machines. A Method for High-Level System Design and Analysis*. Springer, 2003.

[26] Jean-Michel Bruel, Sophie Ebersold, Florian Galinier, Alexandr Naumchev, Manuel Mazzara, and Bertrand Meyer. The role of formalism in system requirements (full version), 2019.

[27] 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 - 26th International Conference, CAV 2014, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 18-22, 2014. Proceedings*, volume 8559 of *Lecture Notes in Computer Science*, pages 334–342. Springer, 2014.

[28] Kaustuv Chaudhuri, Damien Doligez, Leslie Lamport, and Stephan Merz. Verifying safety properties with the TLA+ proof system. In Jürgen Giesl and Reiner Hähnle, editors, *Automated Reasoning, 5th International Joint Conference, IJCAR 2010, Edinburgh, UK, July 16-19, 2010. Proceedings*, volume 6173 of *Lecture Notes in Computer Science*, pages 142–148. Springer, 2010.

[29] ClearSy. *B Language Reference Manual, version 1.8.6.* 2007.

[30] Alin Deutsch, Liying Sui, and Victor Vianu. Specification and verification of data-driven web applications. *J. Comput. Syst. Sci.*, 73(3):442–474, 2007.

[31] Jonathan Edwards, Daniel Jackson, and Emina Torlak. A type system for object models. In Richard N. Taylor and Matthew B. Dwyer, editors, *Proceedings of the 12th ACM SIGSOFT International Symposium on Foundations of Software Engineering, 2004, Newport Beach, CA, USA, October 31 - November 6, 2004*, pages 189–199. ACM, 2004.

[32] Farheen, Sabria. Improvements to Transitive-Closure-based Model Checking in Alloy. Master's thesis, David R. Cheriton School of Computer Science, 2018.

[33] Benoît Fraikin, Marc Frappier, and Régine Laleau. State-based versus event-based specifications for information systems: a comparison of B and EB^3. *Software and Systems Modeling*, 4(3):236–257, 2005.

[34] Marc Frappier, Benoît Fraikin, Romain Chossart, Raphaël Chane-Yack-Fa, and Mohammed Ouenzar. Comparison of model checking tools for information systems. In Jin Song Dong and Huibiao Zhu, editors, *Formal Methods and Software Engineering - 12th International Conference on Formal Engineering Methods, ICFEM 2010, Shanghai, China, November 17-19, 2010. Proceedings*, volume 6447 of *Lecture Notes in Computer Science*, pages 581–596. Springer, 2010.

[35] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. Deriving a textual notation from a metamodel: an experience on bridging modelware and grammarware. In João Andrade Almeida, Luis Ferreira Pires, and Marten J. van Sinderen, editors, *Milestones, Models and Mappings for Model-Driven Architecture: European Workshop on Milestones, Models and Mappings for Model-Driven Architecture (3M4MDA), Bilbao, Spain, July 11, 2006. Proceedings*, number 02 in CTIT Workshop Proceedings Series, pages 33–47, Netherlands, 2006. Centre for Telematics and Information Technology (CTIT).

[36] Angelo Gargantini, Elvinia Riccobene, and Patrizia Scandurra. A metamodel-based language and a simulation engine for abstract state machines. *J. UCS*, 14(12):1949–1983, 2008.

[37] Yuri Gurevich. Reconsidering turing's thesis (toward more realistic semantics of programs). Technical Report CRL-TR-36-84, University of Michigan, September 1984.

[38] David Harel. Statecharts: A visual formalism for complex systems. *Sci. Comput. Program.*, 8(3):231–274, 1987.

[39] John Hatcliff, Gary T. Leavens, K. Rustan M. Leino, Peter Müller, and Matthew J. Parkinson. Behavioral interface specification languages. *ACM Comput. Surv.*, 44(3):16:1–16:58, 2012.

[40] Gerard J. Holzmann. *The SPIN Model Checker - primer and reference manual*. Addison-Wesley, 2004.

[41] Nghi Huynh, Marc Frappier, Herman Pooda, Amel Mammar, and Régine Laleau. SGAC: A multi-layered access control model with conflict resolution strategy. *Comput. J.*, 62(12):1707–1733, 2019.

[42] Daniel Jackson. Alloy: a lightweight object modelling notation. *ACM Trans. Softw. Eng. Methodol.*, 11(2):256–290, 2002.

[43] Daniel Jackson. *Software Abstractions: Logic, Language, and Analysis*. The MIT Press, 2012.

[44] Michael Jastram and Prof Michael Butler. *Rodin User's Handbook: Covers Rodin v.2.8*. CreateSpace Independent Publishing Platform, USA, 2014.

[45] Clifford B. Jones. *Systematic software development using VDM (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1991.

[46] Leslie Lamport. The temporal logic of actions. *ACM Trans. Program. Lang. Syst.*, 16(3):872–923, 1994.

[47] Leslie Lamport. *Specifying Systems, The TLA+ Language and Tools for Hardware and Software Engineers*. Addison-Wesley, 2002.

[48] Leslie Lamport. Real-time model checking is really simple. In Dominique Borrione and Wolfgang J. Paul, editors, *Correct Hardware Design and Verification Methods, 13th IFIP WG 10.5 Advanced Research Working Conference, CHARME 2005, Saarbrücken, Germany, October 3-6, 2005, Proceedings*, volume 3725 of *Lecture Notes in Computer Science*, pages 162–175. Springer, 2005.

[49] Leslie Lamport. The pluscal algorithm language. In Martin Leucker and Carroll Morgan, editors, *Theoretical Aspects of Computing - ICTAC 2009, 6th International Colloquium, Kuala Lumpur, Malaysia, August 16-20, 2009. Proceedings*, volume 5684 of *Lecture Notes in Computer Science*, pages 36–60. Springer, 2009.

[50] Leslie Lamport. *A PlusCal User's Manual (C-Syntax)*, version 1.8 edition, August 2018. https://lamport.azurewebsites.net/tla/c-manual.pdf.

[51] Leslie Lamport. *A PlusCal User's Manual (P-Syntax)*, version 1.8 edition, August 2018. https://lamport.azurewebsites.net/tla/p-manual.pdf.

[52] Leslie Lamport and Lawrence C. Paulson. Should your specification language be typed. *ACM Trans. Program. Lang. Syst.*, 21(3):502–526, 1999.

[53] Soren Lauesen. *Software Requirements: Styles and Techniques*. Pearson Education, 1 edition, 2001.

[54] Leslie Lamport. Safety, Liveness, and Fairness. https://lamport.azurewebsites.net/tla/safety-liveness.pdf, May 2019.

[55] Michael Leuschel and Michael J. Butler. ProB: A model checker for B. In Keijiro Araki, Stefania Gnesi, and Dino Mandrioli, editors, *FME 2003: Formal Methods, International Symposium of Formal Methods Europe, Pisa, Italy, September 8-14, 2003, Proceedings*, volume 2805 of *Lecture Notes in Computer Science*, pages 855–874. Springer, 2003.

[56] Michael Leuschel and David Schneider. Towards B as a high-level constraint modelling language - solving the jobs puzzle challenge. In Ameur and Schewe [19], pages 101–116.

[57] Natalia López, Manuel Núñez, and Ismael Rodríguez. Assessing the expressivity of formal specification languages. In Michael Johnson and Varmo Vene, editors, *Algebraic Methodology and Software Technology, 11th International Conference, AMAST 2006, Kuressaare, Estonia, July 5-8, 2006, Proceedings*, volume 4019 of *Lecture Notes in Computer Science*, pages 220–234. Springer, 2006.

[58] Azzam Maraee and Arnon Sturm. Imperative versus declarative constraint specification languages: a controlled experiment. *Software and Systems Modeling*, May 2020.

[59] Franco Mazzanti, Alessio Ferrari, and Giorgio Oronzo Spagnolo. Towards formal methods diversity in railways: an experience report with seven frameworks. *STTT*, 20(3):263–288, 2018.

[60] Kenneth L. McMillan. The smv system. *Symbolic Model Checking*, page 61–85, 1993.

[61] Chris Newcombe. Why amazon chose TLA$^+$. In Ameur and Schewe [19], pages 25–39.

[62] Nimal Nissanke. *Formal Specification: Techniques and Applications*. Springer London, 1999.

[63] Gerard O'Regan. *Concise Guide to Formal Methods - Theory, Fundamentals and Industry Applications*. Undergraduate Topics in Computer Science. Springer, 2017.

[64] Jonathan S. Ostroff. Validating software via abstract state specifications. Technical Report EECS-2017-02, York University, 2017.

[65] Shari Lawrence Pfleeger and Joanne M. Atlee. *Software engineering - theory and practice (3. ed.)*. Ellis Horwood, 2006.

[66] Mireille Samia, Harald Wiegard, Jens Bendisposto, and Michael Leuschel. High-level versus low-level specifications: Comparing B with Promela and ProB with Spin. In Attiogbe and Mery, editors, *Proceedings TFM-B 2009*, pages 49–61. APCB, June 2009.

[67] Patrizia Scandurra, Angelo Gargantini, Claudia Genovese, Tiziana Genovese, and Elvinia Riccobene. A concrete syntax derived from the abstract state machine metamodel. In *Proceedings of the 12th International Workshop on Abstract State Machines, ASM 2005, March 8-11, 2005, Paris, France*, pages 345–368, 2005.

[68] Jose Serna, Nancy A. Day, , and Shahram Esmaeilsabzali. Dash: Declarative modelling with control state hierarchy (preliminary version). Technical Report CS-2018-04, David R. Cheriton School of Computer Science, 2018.

[69] José Serna, Nancy A. Day, and Sabria Farheen. DASH: A new language for declarative behavioural requirements with control state hierarchy. In *IEEE 25th International Requirements Engineering Conference Workshops, RE 2017 Workshops, Lisbon, Portugal, September 4-8, 2017*, pages 64–68. IEEE Computer Society, 2017.

[70] Serna, Jose. Dash: Declarative Behavioural Modelling in Alloy. Master's thesis, David R. Cheriton School of Computer Science, 2019.

[71] Edel Sherratt. Relativity and abstract state machines. In Øystein Haugen, Rick Reed, and Reinhard Gotzhein, editors, *System Analysis and Modeling: Theory and Practice - 7th International Workshop, SAM 2012, Innsbruck, Austria, October 1-2, 2012. Revised Selected Papers*, volume 7744 of *Lecture Notes in Computer Science*, pages 105–120. Springer, 2012.

[72] Marc Spielmann. *Abstract state machines: verification problems and complexity*. PhD thesis, RWTH Aachen University, Germany, 2000.

[73] J. Michael Spivey. *Z Notation - a reference manual (2. ed.)*. Prentice Hall International Series in Computer Science. Prentice Hall, 1992.

[74] Allison Sullivan, Kaiyuan Wang, Sarfraz Khurshid, and Darko Marinov. Evaluating state modeling techniques in alloy. In Zoran Budimac, editor, *Proceedings of the Sixth Workshop on Software Quality Analysis, Monitoring, Improvement, and Applications, Belgrade, Serbia, September 11-13, 2017*, volume 1938 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2017.

[75] Emina Torlak and Daniel Jackson. Kodkod: A relational model finder. In Orna Grumberg and Michael Huth, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 13th International Conference, TACAS 2007, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2007 Braga, Portugal, March 24 - April 1, 2007, Proceedings*, volume 4424 of *Lecture Notes in Computer Science*, pages 632–647. Springer, 2007.

[76] Yuan Yu, Panagiotis Manolios, and Leslie Lamport. Model checking tla^{+} specifications. In Laurence Pierre and Thomas Kropf, editors, *Correct Hardware Design and Verification Methods, 10th IFIP WG 10.5 Advanced Research Working Conference, CHARME '99, Bad Herrenalb, Germany, September 27-29, 1999, Proceedings*, volume 1703 of *Lecture Notes in Computer Science*, pages 54–66. Springer, 1999.

[77] Pamela Zave. A practical comparison of alloy and spin. *Formal Asp. Comput.*, 27(2):239–253, 2015.

[78] Hehua Zhang, Ming Gu, and Xiaoyu Song. Specifying time-sensitive systems with TLA+. In Sheikh Iqbal Ahamed, Doo-Hwan Bae, Sung Deok Cha, Carl K. Chang, Rajesh Subramanyan, Eric Wong, and Hen-I Yang, editors, *Proceedings of the 34th Annual IEEE International Computer Software and Applications Conference, COMPSAC 2010, Seoul, Korea, 19-23 July 2010*, pages 425–430. IEEE Computer Society, 2010.

APPENDICES

Appendix A

Tool versions

This appendix lists the versions of the tool support for each language we used when carrying out our case studies. The tools are all free software [7] and publicly available for use by anyone.

- For B, we used ProB:

  ```
  ProB 1.9.0-release
  718e254497d921bb4f82945fefdb73774780d007
  Wed Jul 17 16:07:40 2019 +0200
   TclTk 8.5.19
   SICStus 4.5.1 (x86_64-linux-glibc2.17): Tue Apr  2 06:27:49 PDT 2019
  ```

- for EVENT-B, we used the Rodin Platform `Version: 3.4.0.201802230927-6980ca1` along with the `ProB for Rodin3 3.0.10.201909041430 de.prob2.feature.feature.group HHU Düsseldorf STUPS Group` plugin.

- For ALLOY, we used the Alloy Analyzer `5.1.0 built 2019-08-14T18:53:58.297Z`.

- For DASH, we used commit `725cd790497cf561443033d2af666877df6d58f5` of `https://git.uwaterloo.ca/jserna/dash` built using

  ```
  Eclipse DSL Tools

  Version: 2019-12 (4.14.0)
  Build id: 20191212-1212
  ```

- For TLA$^+$ and PLUSCAL, we used the TLA$^+$ Toolbox and its accompanying tools:

  ```
  TLA+ Toolbox provides a user interface for TLA+ Tools.

  This is Version 1.6.0 of 10 July 2019 and includes:
    - SANY Version 2.1 of 23 July 2017
    - TLC Version 2.14 of 10 July 2019
    - PlusCal Version 1.9 of 10 July 2019
    - TLATeX Version 1.0 of 20 September 2017
  ```

- For ASMETAL, we used

  ```
  Eclipse IDE for Java Developers
  Version: 2019-09 R (4.13.0)
  Build id: 20190917-1200
  ```

  with the following plugins:

  ```
  Asmeta Animator 0.0.10
  Asmeta editor and simulator 0.9.10
  Asmeta model advisor 0.0.14
  Asmeta model checker 1.0.6
  Asmeta test generator 0.0.3
  Asmeta visualizer 1.0.8
  ```