B Annotations in Critical Control Systems Development

Wilson Ifill

Submitted in fulfilment of the degree of Doctor of Philosophy
Department of Computing
July 6, 2008
Annotations in Critical Control Systems Development

Wilson Ifill

Submitted in fulfilment of the degree of Doctor of Philosophy

University of Surrey
Department of Computing

July 6, 2008
To
Kim,
Luke, Aaron, Ross, Alice and
Dexter.
B Annotations in Critical Control Systems Development

Wilson Ifill
Department of Computing University of Surrey Guildford England
w.ifill@surrey.ac.uk

July 6, 2008

Abstract

The design and implementation of critical controllers benefit from development in a formal method such as B. However, B does not support execution specification directly, which is a requirement in controller design. The aim here is to develop a set of annotations so that they can be used by a B design engineer to capture execution requirements, while creating the B model. The annotations, once shown to be machine-annotation consistent with the B machine, can be used independently from the machine to assess the correctness of CSP controllers developed to detail the control behaviour. CSP||B is a formal method integration that can be used to develop critical controller with both state and event behaviour. The advantages of using annotations is that the execution requirements can be captured and shown to be machine-annotation consistent during state operation development, and that a control loop invariant does not have to be independently developed.

Handel-C is used on route to hardware synthesis as it supports the implementation of concurrency and the manipulation of state. Annotations are utilised again to guide the translation of the B and control annotations into Handel-C.

This PhD. work has three main aims. Firstly, to introduce a set of annotations to describe control directives to permit controller development in B. The annotations capture execution requirements. They give rise to proof obligations that when discharged prove that the annotations are machine-annotation consistent with the machine they are written in, and therefore will not cause the machine to diverge. Secondly, we have proven that CSP controllers that are consistent with the annotations will preserve the non-divergence property established between the machine and the annotations. Thirdly, we show how annotation refinement is possible, and show a range of mappings from the annotated B and the consistent controller to Handel-C. The development of mappings demonstrates the feasibility of automatic translation of annotated B to Handel-C.

Keywords
B, CSP, B annotation, critical state machine development, Handel-C, and BVHDL
Acknowledgements

Thank you to my wife Kim and my children Luke, Aaron, Ross, Alice, and Dexter for supporting this endeavour. Thanks to Steve Schneider, my supervisor, for constantly testing my ideas and for his patience. Thanks to my managers along the way who supported my case to fund this part-time PhD.: Dave Thomas, Digby Knight and John Eves. Thank you to Helen Trehanne and Roger Peel for advice and support. Thank you to Ib Sorensen and Dave Neilson for their help and guidance on the use of the BToolkit.

Published Papers, Book Chapters and Posters

Published works:

- The use of B to specify, design and verify hardware, book chapter [ISS01]
- Augmenting B with Control Annotations, conference paper at B2007 [IST07]
- A Step Towards Refining and Translating B Control Annotations to Handel – C, conference paper at CPA2007 [IST07]
- Towards a Demonstrably Correct Ada Compiler, conference paper at SIGAda 2007 [NIM07]
- Achieving B state Machine Designs with Annotations, prize winning poster at ZB2005

Unpublished works:

- B specifications and Proof of the AEP Pipelined Processor [Iff02]
- Combining B, OSP and Handel – C to Specify, Design and Verify Hardware [Iff03]
Contents

Table of Contents

1. Background Motivation and Notation 1
   1.1. Introduction to Chapters 2
   1.2. Technical Background 4
   1.3. typographic convention 19

2. The NEXT Annotation 21
   2.1. Introduction 21
   2.2. The General Framework 21
   2.3. The Approach 23
   2.4. The NEXT Annotation of a Consistent B Machine 25
   2.5. Controller Syntax and Auxiliary Functions for NEXT Annotations 26
   2.6. Annotation Consistency with Action-Prefix 28
   2.7. Proving Termination of Controlled Machines with NEXT Annotations 30
   2.8. Picturing of NEXT Annotation 36
   2.9. Examples Utilising NEXT 37

3. The FROM and QUERY Annotations 41
   3.1. The FROM annotations 41
   3.2. The QUERY Annotations 64

4. Adding I/O and NEXT INVARIANT to Annotations 75
   4.1. Communication Between B Operations, Controller and Environment 75
   4.2. I/O NEXT Annotations and the NEXT INVARIANT 78
   4.3. Examples of the Use of I/O NEXT 81

JULY 6, 2008
4.4. I/O NEXT Annotation Listing Functions .............................................................. 84
4.5. I/O FROM Annotations ............................................................................................ 86
4.6. I/O NEXT and FROM Annotation Consistency ...................................................... 92
4.7. Proving Termination of Controlled Machines with I/O Annotations .................. 94
4.8. Picturing I/O Annotations ...................................................................................... 97
4.9. I/O NEXT Examples ............................................................................................... 97
4.10. NEXT INARIANT Annotations Example ............................................................... 100

5. Extending the NEXT Annotation ........................................................................ 107
5.1. The NEXT Extensions ............................................................................................ 107
5.2. The SELECT NEXT Annotations ........................................................................ 107
5.3. The CONDITION NEXT Annotations ................................................................ 110
5.4. Controller Syntax and Auxiliary Functions with CONDITION NEXT Annotations 111
5.5. SELECT NEXT and CONDITION NEXT Annotation Consistency ............. 113
5.6. Proving Termination of Controlled Machines with CONDITION NEXT Annotations 115
5.7. Examples Utilising SELECT NEXT and CONDITION NEXT ....................... 121

6. Translating and Refining Annotations ................................................................ 129
6.1. The INTERLEAVED NEXT Annotation .............................................................. 132
6.2. Annotation Consistency with Interleaving ......................................................... 140
6.3. Refinement and Translation to Handel-C .......................................................... 157
6.4. Example: Safe Control System ........................................................................... 163

7. Comparisons with Other Work ............................................................................ 177
7.1. Associated Technologies ..................................................................................... 177
7.2. Hardware Verification and Hardware Description Languages ......................... 179
7.3. Hardware Development in B ............................................................................. 183
7.4. State Machine Modelling .................................................................................... 197
7.5. CSP and Z ............................................................................................................ 198
7.6. Event-B ............................................................................................................... 201
7.7. Dynamic Constraints in B .................................................................................. 210
7.8. ProB ..................................................................................................................... 212
7.9. Translation of CSP to Handel-C ......................................................... 215

B. Future Work and Conclusion ................................................................. 217
  8.1. Future Work ................................................................................ 217
  8.2. Conclusion .................................................................................. 220

A. The Comparison Between CSP||B's Translations and B Annotations  225

B. Glossary ............................................................................................. 227
  B.1. Annotations Syntax ........................................................................ 227
# List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1.</td>
<td>BVIDL Development Methodology</td>
</tr>
<tr>
<td>1.2.</td>
<td>Future Annotated B Handel-C Development Methodology</td>
</tr>
<tr>
<td>1.3.</td>
<td>Illustration of a Handel-C Function</td>
</tr>
<tr>
<td>2.1.</td>
<td>The Process Flow in the Approach</td>
</tr>
<tr>
<td>2.2.</td>
<td>A B Machine</td>
</tr>
<tr>
<td>2.3.</td>
<td>Picturing the ({op_j, \ldots, op_k}) \textsc{next} Annotation of (op_i)</td>
</tr>
<tr>
<td>2.4.</td>
<td>Traffic Machine</td>
</tr>
<tr>
<td>2.5.</td>
<td>Traffic Controller</td>
</tr>
<tr>
<td>2.6.</td>
<td>Traffic Machine POs</td>
</tr>
<tr>
<td>2.7.</td>
<td>Traffic Controller Consistency</td>
</tr>
<tr>
<td>3.1.</td>
<td>Picturing the \textsc{from}-\textsc{any} Annotation of (op_j)</td>
</tr>
<tr>
<td>3.2.</td>
<td>Picturing the \textsc{from}-\textsc{set}({op_i, \ldots, op_k}) Annotation of (op_j)</td>
</tr>
<tr>
<td>3.3.</td>
<td>Picturing the \textsc{from}-\textsc{set-init} Annotation of (op_j)</td>
</tr>
<tr>
<td>3.4.</td>
<td>Global Interrupting Traffic Machine - Part 1</td>
</tr>
<tr>
<td>3.5.</td>
<td>Global Interrupting Traffic Machine - Part 2</td>
</tr>
<tr>
<td>3.6.</td>
<td>Global Interrupting Traffic Controller</td>
</tr>
<tr>
<td>3.7.</td>
<td>Defined Traffic Machine</td>
</tr>
<tr>
<td>3.8.</td>
<td>Defined Traffic Controller</td>
</tr>
<tr>
<td>3.9.</td>
<td>Carrying Through \textsc{next} Annotation Expectations Through Query Operation</td>
</tr>
<tr>
<td>3.10.</td>
<td>Picturing the (op_i) \textsc{query} Annotation</td>
</tr>
<tr>
<td>3.11.</td>
<td>Querying Traffic Machine</td>
</tr>
<tr>
<td>3.12.</td>
<td>Querying Traffic Controller</td>
</tr>
<tr>
<td>4.1.</td>
<td>A B machine with Operations with I/O</td>
</tr>
</tbody>
</table>
List of Figures

4.2. The Controller Initiates Action that Reads Environment Status .................. 77
4.3. The B Operation Outputs to the Environment and Controller .................. 77
4.4. Evens Machine .................................................................................. 82
4.5. Example of a NEXT Annotation with I/O .............................................. 83
4.6. Controller B Operation Interaction ...................................................... 84
4.7. SetDifferent Machine ........................................................................ 98
4.8. SetDifferent Controller ...................................................................... 98
4.9. Heater Machine Header ...................................................................... 100
4.11. Heater Controller Operations ........................................................... 102

5.1. Safety Switch Machine - Header ........................................................... 123
5.2. Safety Switch Machine - Environmental Operations ............................ 124
5.3. Safety Switch Machine - Controller Operations ..................................... 126
5.4. Safety Switch Machine Controller Operations ....................................... 126
5.5. Annotated B to CSP Synthesis Guide .................................................. 127

6.1. Different Views of the Same Action ....................................................... 130
6.2. A B Refinement with Operations and I/O ............................................ 132
6.3. NEXT Refinements - Reduction of Non-determinism .............................. 158
6.4. Picturing SEQUENCE NEXT Refined Operations ................................. 159
6.5. NEXT Refinements - Structural Refinements ......................................... 160
6.6. NEXT Annotation Translation Guide Part 1 .......................................... 161
6.7. NEXT Annotation Translation Guide Part 2 .......................................... 162
6.8. Safe Machine - Part 1 ......................................................................... 167
6.9. Safe Machine - Part 2 ......................................................................... 168
6.10. Safe Machine Controller .................................................................... 169
6.11. Safe Refinement - Part 1 .................................................................... 170
6.13. Refined Safe Controller ..................................................................... 172
6.15. SafeR Translation Part 2 ................................................................. 173

x

JULY 6, 2008
List of Figures

6.16. SafeR Translation Part 3 ............................... 174
6.18. CSP to Handel-C Translation Guide ............... 176

7.1. AEP Example Essentials ............................... 184
7.2. AEP Entity Architecture Simple Machine Plus Packages ................................................. 185
7.3. AEP Sequencer Process Machine .................. 187
7.4. AEP Read Process Machine ......................... 187
7.5. AEP Write Process Machine ......................... 188
7.6. AEP Read Process Machine ......................... 188
7.7. AEP VHDL - Part 1: Declarations .................. 189
7.8. AEP VHDL - Part 2: Processes ..................... 190
7.9. BHDL Description of a Multiplexer ................. 192
7.10. The Multiplexer Implementation .................. 192
7.11. Nine Valued Logic BHDL Model .................. 193
7.13. Distributed Model Check Synchronisation ........ 197
7.14. Example of a CSP Controller for CSP-Z specification ......................................................... 199
7.15. BVHDL-CSP Preliminary Functions ............... 202
7.16. BVHDL-CSP Main Section (Effect) ................. 203
7.17. BVHDL-CSP: Semantic Function .................. 204
7.18. PULSER1 Circuit with Clocking ................... 206
7.19. Legal Signal Transitions ............................... 208
7.20. Track and Signal Arrangement ..................... 208
7.21. Distributed Model Check Synchronisation ......... 209
7.22. Traffic Controller Machine for ProB ............... 214

8.1. Interrupting Traffic Controller Machine with the Variant Annotation ......................... 219
8.2. Interrupting Traffic Controller CSP with Variant .......................................................... 220
8.3. Future Annotated B Handel-C development Methodology .............................. 221
List of Tables

1.1. B Mappings to CSP .................................................. 19
1.2. Typographical conventions ........................................... 20

6.1. Existing CSP to Handel-C Translation Guide ................. 163

7.1. The Temporal Logic Operators Used in ProB .................. 215
7.2. CTL Temporal Logic Connectives ................................. 216

8.1. Annotations and Related Control Operations .................. 223

A.1. The Relationship of B Translations of CSP Fragments to Annotations ... 226
List of Tables
1. Background Motivation and Notation

The contribution of this thesis proposes the use of annotations to allow B designers to capture control aspects of the design during the development of the B operations for critical control systems. The annotations, when entered to capture control aspects, represent a specification of control flow, which have to be met by the final control system implementation. The annotations can be refined along with the machines and they are also used to record the design decisions about implementation choices in hardware. An annotation to support interrupts is included. The contribution of this thesis also includes translating the annotated B and CSP controllers to Handel-C (Handel-C is the choice of implementations language).

The thesis aims were set by the context surrounding the work. Critical control systems need to be developed for analysability. Concurrency and multi-processor based implementations are only resorted to if necessary. Control systems implemented with a single thread on a single processor (back up processors may be necessary) are preferred and control systems implementation in hardware is preferred where possible to avoid operating systems use. Development methodology mandated for the most critical systems include simple analysable source code subsets, formal specification and support of proof, static analysis, configuration management of code and verification records. The B methodology and associated toolkits support some of these aspects. Some of the more challenging requirements of the standards such as formal verification of the toolkits may never be met by any vendor. This thesis shows that control systems development, refinement, proof and code generation to a simple language subset can all supported by annotations. The BToolkit provides the documentation and configuration support for this methodology. Code generation is used to remove coding errors and reduce coding costs in development. Code generation supported by libraries in B allows reusability and permits refinement to specified library components. In the B Methodology all source code is developed from configurable library components. In this thesis translation to a source language is utilised. Unproven translation is not as good as refinement to proven source language libraries, but it is a step towards such rigour.

In light of the above context the goals of the research were as follows:

- accommodate controller state machine design
- opt for simplicity to accommodate analysis
- provide multiple views of a development for checking from different point of views
- work with existing tools
- support code generation

JULY 6, 2008
• have one design entry tool and language - accommodated B centred development process
• permit hardware or software implementation
• abstract away from the development of explicit clocks in single clock systems where hardware implementation is selected
• take advantage of parallel/interleaved processing capabilities if hardware target selected
• avoid operating system in implementation

1.1. Introduction to Chapters

Chapter 1.2 is the technical background to the work. It introduces the notation of B, VHDL, CSP, Handel-C and CSP||B. It reviews how specifications are captured in B and translated or refined to implementations. The introduction of CSP allows concurrency and event behaviour to be specified. Handel-C is an implementation route for CSP. CSP||B is an existing integrated formalism.

The technical chapters' (chapter 2 to 6.3) layout reflects the framework by which new annotations are introduced. Other annotations developed in future work would follow the same framework. An overview of the general framework is given in Section 2.2. What that leads to in terms of sections in each chapters is itemised as follows:

• Discussion of the purpose of the annotation
• Definition of syntax of annotation and possible abbreviations
• Definition of the Proof Obligations (POs) that guarantee machine-annotation consistency
• Definition of the controller syntax that the annotations are associated with
• Definition of functions to extract information from CSP controller fragments
• Definition of annotation consistency
• Proving correctness of machine-controller pairs
• Picturing the annotations using diagrams
• Examples utilising the introduced annotation
• Demonstrations of machine-annotation consistency
• Demonstration of annotation-controller consistency

JULY 6, 2008
In chapter 2 the annotation approach and the basic NEXT annotation is introduced, which is applicable to both classical B and Event-B. The annotation framework is about defining annotations, showing them consistent with the machine they are attached to and the controller that implements them. The approach to establishing that annotations are consistent with the machine is termed machine-annotation consistency (the various uses of the term consistency are defined in Section 2.3.2). The approach to showing that annotations are consistent with a controller is termed annotation-consistency. A proof showing that given machine-annotation consistency and annotation-controller consistency then machine-controller consistency follows, is given for each annotation that is introduced in the technical section where appropriate. This proof guarantees non-divergence: a machine and controller that are consistent will not diverge when they run together. The NEXT annotation is associated with the prefix and choice controller fragments. A short example using the NEXT annotation is given.

In chapter 3 two more annotations are introduced: FROM and QUERY. The FROM annotation is like the NEXT annotation, but it introduces interrupt behaviour. The FROM annotation comes in three different varieties. The operational definition of the associated controller CSP interrupt definition is defined. A definition of the traces model for the interrupt is given. It differs from the standard trace model of CSP interrupt operators, because the concept of an interrupt set has been added. The QUERY annotation is used to mark operations that do not cause a state change, but just return information to the environment. The guarantees that the NEXT and FROM annotations set up must be carried across QUERY annotated operations. The introduction of the QUERY annotation leads to the associated hiding CSP operator. As in chapter 2 the machine-annotations POs are introduced. Then the consistency rules for annotation-controller consistency are stated, followed by proofs of non-divergence and examples. There are no worked examples of consistency in this case as was the case in the last chapter. This waits until the next chapter where I/O is introduced. Chapter 4 introduces I/O into the NEXT and FROM annotations. The model of communications is introduced, which has aspects of interactions between the environment, the controller and the B operations. This chapter and proceeding chapters follow the same chapter framework as the previous ones, apart from there being no section on the pictorial representation of the annotations. A NEXT INVARIANT is introduced in Chapter 1. They are unlike normal annotations as they are static predicates associated with operations. Chapter 5 follows the standard framework. It introduces the SELECT NEXT and CONDITIONAL NEXT annotations. These annotations allow branching. A table is included that details the systematic translations of the annotations in the chapter example to the CSP of the controller fragments. This demonstrates the possibility of automatic synthesis of controllers from annotations. Chapter 6 introduces the INTERLEAVED NEXT and SELECT NEXT annotations. These annotations are helpful for annotation refinement and structuring specifications for implementations in hardware. Although this chapter follows the normal framework additional information is given in the form of tables outlining translations of annotations to Handel-C and the refinement of annotations. This chapter highlights some areas for future work.

A comparison with other related work is given in chapter 7. In the development of hardware and software there is much diversity and the approaches to development range from formal through semi-formal to informal. Semi-formal approaches involve property or assertion checking, or utilise a formal notation somewhere in the development cycle. No fully formal commercial hardware or software development route that harnesses full refinement exists. A sample of relevant methodologies, formalisms and tools are reviewed that compare with the
Chapter 1. Background Motivation and Notation

1.2. Technical Background

In early work at MSc level investigated the use of a B Hardware Description Language (HDL) (which was reported on in the MSc dissertation [El99]). The BVHDL language was used to capture the specification of a pipeline processor, Another Example Processor (AEP), and that specification is discussed in the related work chapter 7. The work on the AEP highlighted two issues associated with obtaining a translation: the need to specify a clock and the requirement for specialised hardware data types at the VHDL Register Transfer Level (RTL). The need for the detail is reflected in the need to develop implementations that are synthesisable.

Research then began into a higher level approach from specification to Field Programmable Gate Array (FPGA) implementation for critical control systems. The technical background section reviews the other essential elements necessary to embark on the research, such as an understanding of VHDL, Handel-C, a simple subset of CSP, and CSP||B [Tre00]. The review of VHDL highlights the differences between B and VHDL. Besides the clock there is the use of sensitivity lists and time delays in the VHDL language that distinguishes it from B. However, such language constructs are not essential for the development of critical controller circuits. In this thesis the interrupt operator is defined as it differs from the CSP operator in some respects. The Handel-C language is introduced which has many similarities to CSP. One notable difference is the principles of communication detailed in the review.

Annotations research is closely related to the research into CSP||B. Both approaches use CSP controllers to sequence the actions of B machines. Annotations are given a B Proof Obligations (POs) semantics and CSP consistency to allow them to be viewed as, (1) part of the B; or, (2) as a CSP controller. The CSP and B in CSP||B are brought together by giving the B a CSP action systems semantics. Action systems have an initialisation and set of guarded actions. Actions systems like B can be represented in CSP by considering their weakest precondition (wp) semantics. Treatman [TS00] defines the wps of B operations guarded with true. The wp of a sequence of actions can be calculated thereby giving meaning to a B execution in terms of CSP traces. Controllers can therefore be translated into B specifications, and tested to see if execution loops preserve Control Loop Invariants (CLI). CLI are used to show the correctness of loops within the controller. The use of annotations prevents the need to convert the CSP of the controller to B or develop a CLI.

Research translating between formal notations and Hardware Description Languages (HDL) is well established and is discussed in detail in section 1.2.1. Research exists on B [Abr96] to VHDL [VAS05] translation [ABD*03] [ISS01]. VHDL is a widely used HDL and is reviewed in section 1.2.3. It is a simulation language that has a subset that when written with a synchronous reset and synthesisable RTL constructs can be directly synthesised into a netlist of basic gates. A netlist can be laid out on a logic device such as a FPGA. The VHDL must be at a sufficiently low level to be able to be synthesised into a netlist representation. In this thesis designs are refined to a level above RTL. This is possible because a different HDL language is utilised for synthesis. The higher level of abstraction possible from which netlist can be synthesised means less detailed hardware oriented refinement, but means less efficient circuits. In this thesis less efficient circuits are acceptable. The paramount concern is to
maintain safety and security properties proven during development. Handel-C is an example of a higher level HDL that can support synthesis into hardware, but abstracts away from details of the hardware such as clocks.

This related work chapter firstly discusses, in section 1.2.1, the methodological approach taken previously by the author using BVHDL [ISS01] (B hardware description language based on VHDL), and the new proposed approach in the technical section of this thesis. The technical background of the B method utilised by the BToolkit are reviewed in section 1.2.2. A detailed discussion of BVHDL that reveals the unnecessary detail that must be reflected in the B specifications, and understand how previously implicit control was coded into the guards or preconditions of the models is given in section 7.3.1. The interesting aspects of VHDL are reviewed in section 1.2.3. The aspects of CSP that are utilised in this thesis are reviewed in section 1.2.4. Handel-C the target language used in this thesis is reviewed in section 1.2.5. Lastly, we review the work on CSP||B (an approach to controlling B machines with CSP). The ability to express control flow in the CSP||B formalism has its advantages, certainly BVHDL does not have this capability. However, a criticism is that the control flow specification is not developed at the same time as the state models are, and there is no established hardware/software implementation route from CSP||B.

1.2.1. Hardware Development and Analysis in B

B has been used to capture and verify static properties of a hardware design at RTL. By keeping to a prescriptive entity-architecture structure familiar to hardware designers, it is possible to translate directly to VHDL from B. By assuming a single clock and being more abstract about data structures, we can save the effort otherwise spent detailing clock and data refinements in B and still achieve a translation to hardware, but this time via Handel-C.

This section outlines background work concerning trusted formal development techniques in the production of digital hardware. The B Toolkit has been used to develop formal software [NS01]. Hardware development traditionally has involved coding VHDL [Asl96] from English requirements, simulation and synthesis to FPGAs or custom ASICs. Validation is undertaken in the later stages. One of the problems with this approach is that requirements, particularly high level requirements, are not tested early enough and the lack of a design other than the VHDL model makes maintenance difficult.

The approach to the development of hardware using BVHDL is more flexible. A hardware engineer more accustomed to developing low-level RTL, can develop B in a style closely resembling VHDL using B Abstract Machine Notation (AMN) [Abr96] and generate synthesizable VHDL from within the B Toolkit. This approach to hardware design using BVHDL can be viewed as a substitute methodology to VHDL, which does allow proofs to be obtained in the BVHDL model. However, the methodology could be extended to do much more. The B Toolkit provides an environment in which higher level specifications may be written in advance of low level VHDL code generation. The environment allows the animation of abstract specifications to investigate requirements and generate validation test cases. It generates and assists in the discharging of formal POs thereby formally linking abstract designs and lower level refinements. A VHDL generator has been developed by B-CORE (UK) Ltd., so VHDL may be obtained from the proven AMN refinements, but the constraints on the form of the
BVHDL means unnecessary design detail must be included in the model right from the start. This restricts the extent that abstraction can be utilised. The BVHDL methodology involving design and VHDL generation is given in Figure 1.1.

There is a wider uptake of C and C++ hardware design languages and hardware designers are becoming more accustomed to developing behavioural hardware descriptions of circuits and compiling to logic using Handel-C like languages. This thesis has a more advanced development framework which proposes incorporating control directives in B specifications to guide the development of control in CSP and translation to Handel-C. This allows hardware to be developed from a higher-level of abstraction. The resultant methodology is given in Figure 1.2.

Breuer and Kloos [BKLM97] proposed a formal semantics and a refinement calculus for VHDL. They have demonstrated that "the programming logic and the associated refinement calculus can be shown to be complete". Breuer et. al. make no excuses for the "difficulties of the working of the formalism". However, the difficulty of a formalism will have an impact on its use in the wider community. Ideally, the semantics and refinement calculus should be intuitive. B Guarded Substitution Language (GSL) is intuitive to programmers, when presented as AMN (sugared GSL). The proof obligation of AMN’s machine and refinements are given in [Abr96] and in section 2.3.1 and section 6.0.1. The B interface for hardware development is given in detail in [ISS01]. It provides an excellent interface for the development of hardware models and hardware component libraries.

An unproven translation process may introduce errors into the design model. Handel-C uses OCCAM (OCCAM is an implementation of CSP) for its underlining semantics. Handel-C and OCCAM lie comfortably together, because Handel-C was built from OCCAM in the first instance. This thesis does not provide a proof for the translation from B to CSP or from B to Handel-C. However, the task of verifying a translation from B to a subset of CSP or Handel-C is a much easier task that of verifying a translation from B to VHDL. The work on CSP||B suggests a relationship between the two languages. There are fundamental differences between B and VHDL. VHDL includes the following that have no natural equivalent in B:
1.2. Technical Background

1. the concept of a clock that synchronises register updates
2. sensitivity lists
3. entity architectures
4. processes

To overcome the difference between VHDL and B the style of BVHDL designs are carefully managed, and re-usable libraries are utilised. So instead of giving meaning to the whole of VHDL, meaning is given just to the subset of VHDL that is needed using AMN.

1.2.2. The B Method

A B specification is partitioned hierarchically into MACHINES, each of which is a module encapsulating both constant and variable data, and operations on the data. However, machines cannot be used as data types. A machine describes a number of services that it provides to the system called operations. Operations are introduced in a B machine after the key word OPERATIONS. They specify behaviour using Abstract Machine Notation (AMN) constructs. AMN provides a description of dynamic behaviour within the operations in the form of set theoretic notation and substitutions. The substitutions capture state transformation, and resemble assignments in programming languages. The set theoretic notation describes predicates. The AMN syntax, more commonly known as B, is introduced in the following sections. The focus is on the specification of preconditioned operations and their associated predicate transformers. In section 7.3.1 more technical background of the B method is given with respect to the construction of hierarchical specifications and the specification of hardware.
components. In section 2.3.1 the syntax of a B machine and consistency proof obligations are introduced. In section 6.0.1 details of the refinement proof obligations are given. The ASCII syntax of B is introduced in this section. In subsequent sections (section 1.2.6, page 15) and the following chapters the latex version of the notation is introduced.

As mentioned, operations specify the state transformations and therefore capture the dynamic behaviour of the system. The AMN substitutions capture the effect of carrying out an operation on a particular variable. Within an operation only one assignment may be made to any particular variable of a machine, but several substitutions can be made within one operation. In classical B the operations are preconditioned in the following way:

\[ o_p \triangleq \text{PRE} \ P_{op} \ \text{THEN} \ B_{op} \ \text{END}; \]

The precondition \( P_{op} \) is the predicate that must hold in order to establish that the substitutions in \( B_{op} \) will terminate.

The invariant is introduced in the B machine after the keyword INVARIANT. It is a predicate that captures important properties that the machine must maintain. It characterises the permitted states of the system: the safe states. The expectation is that every possible execution of each operation must take the system from a valid state to a valid state: every operation execution must re-establish the invariant. This is an important aspect of machine-consistency. Predicate transformers are introduced to reason about machine-consistency. On one hand we have the specification of operations in terms of substitutions that change the state, and on the other invariant predicates that must be maintained. Machine-consistency is calculated by transforming the invariant with the operation's substitutions to create a new predicate, that characterises the final states that the operations can produce assuming the invariant held before the operation was invoked. A predicate transformer is written as follows:

\[ [B_{op}]I \]

In the above \( B_{op} \) is the substitutions that transform the predicate \( I \).

The preconditioned operation in Generalised Substitution Language (GSL) is written \( P_{op} | B_{op} \) (\( P_{op} \preceq B_{op} \)).

In this case the predicate, if true, guarantees that the operations re-establishes the machine invariant, \( I \):

\[ I \Rightarrow [P_{op} | B_{op}]I \]

Abrial [Abr96] introduced rules to reduce the predicate transformers. The rule for the reduction of preconditioned GSL is used in the proofs in this thesis and is defined as follows:

\[ [P_{op} | B_{op}]I = P_{op} \land [B_{op}]I \]

The consistency condition reduces to:

\[ \text{JULY 6, 2008} \]
\[ I \Rightarrow P_{op} \land [B_{op}]I \]

which reduces to:

\[ I \land P_{op} [B_{op}]I \]

The above states that given that the invariant and the precondition hold then the predicate formed by transforming the invariant with the body of the operation holds.

The machine has to be brought into a state that is consistent with the invariant initially. The initialisation substitutions follows the INITIALISATION key word in the machine. They represent the initial assignments that the machine makes before the operations are invoked. The predicate that must hold to initially establish the invariant is as follows:

\[ [T]I \]

In the above \(T\) represents the initialisation substitutions and \(I\) the invariant. The above predicate must be true to ensure that the machine state is initially established. Section 1.2.6 gives more technical background on AMN, GSL and associated predicate transformers.

### 1.2.3. VHDL Hardware Description Language

The core of the VHDL language has the imperative flavour of Pascal or C. It supports procedures, functions and concurrency. Hardware is modelled as a set of concurrently executing process. Signals and variables provide local state in the process. Variables are updated at the point an assignment is made within the sequence of events (\(\text{var}_a := 1\)). Signals are updated at the end of a process description in accordance with a specified delay (\(\text{delta}\)). Communication between processes is achieved by signals. Processes do not contain other processes, so parallelism is only one layer deep. Processes can contain while loops, if branches, case statements, etc. The code within a process loops continuously. VHDL introduces two conceptually new statements not available in B:

1. Delayed Signal assignment: \(\text{signal}_a <= \text{signal}_b \text{ after delay}_x\) This statement schedules an assignment in delay time \(\text{delay}_x\) of \(\text{signal}_a\) updated with \(\text{signal}_b\). If the delay time were missing then a default delay \(\text{delta}_x\) would be assumed, which is a non zero infinitesimal delay time. Delay are taken into account during simulation.

2. In the implied wait statement: \(\text{process}_n(\text{sig}_a) = \ldots\) The signal \(\text{sig}_a\) is the only signal in the sensitivity list of process \(\text{process}_n\). This process is blocked until a level transition occurs on the \(\text{sig}_a\). (A transition is a change in voltage at a point in a circuit from logic level 1 to logic level 0 or vice-versa.) Other signals could be added to the sensitivity list.

A standard VHDL simulation executes each process until it becomes blocked by a wait statement (\(\text{wait delay}_x ; \)). Each process must contain a wait. Some waits are implicit. A
wait state is maintained until the next scheduled assignment time or until the wait delay has expired, at which time the process is re-executed from the point of the last wait. Other signals could be added to the sensitivity list.

VHDL is a wide spectrum hardware description language used for modelling, simulation and synthesis of digital logic circuits. It is wise to focus on a synthesisable subset of VHDL and avoid the use of unregistered feedback (logic inputs that are fed directly from the circuit outputs without a register holding that logic state to a value obtained during the previous clock cycle). A stable state may not occur in circuits with unregistered feedback.

The VHDL generated from the B specification of a pipelined processor in section 7.3.1 is given in Figure 7.7, which illustrates the language constructs discussed in this section.

1.2.4. CSP Notation

CSP is a language for describing concurrently executing processes that interact through communication. A detailed definition of CSP can be found in [Ros98] or [Sch00]. CSP is a process algebra defined by Hoare [Hoa85] who introduced CSP denotational semantics in terms of traces, failures, and failures and divergence. Refinement in these denotational semantics is an important notion, and we write $P \vdash T = Q$ to denote $P$ is trace refined by $Q$. Trace refinement can be thought of as language containment, and means that all traces of $Q$ must be included in the traces of $P$. The subset of CSP used in this thesis is described in the following section. CSP models behaviour with events and processes. Events may be associated with a channel of communication. In the following brief BNF of CSP we use the convention that events and channels are denoted with lower case letters and processes with upper case (in the examples both processes and events are lowercase). Events are drawn from the universal alphabet $\Sigma$.

Process $P$ is defined in machine readable ASCII form (CSPM) as follows (the accompanying text below which explains the BNF incorporates the LaTeX version of the CSP operators):

$$P ::=$$

1. $\text{SKIP}$ (successful termination) |
2. $a \rightarrow P$ (prefix) |
3. $P [] P$ (external choice) |
4. $P \mid \mid P$ (internal choice) |
5. $P \mid \mid P$ (interleaving) |
7. $c ! v \rightarrow P$ (output) |
8. $c ? x \rightarrow P (x)$ (input) |
9. $P \{ x \}$ (hide $x$ events) |
10. $f (P)$ (renaming) |

July 6, 2008
11. DIV (divergence) |
12. $\%$ i : T @ (index external choice) |
13. $\cdash\%$ i : T @ (index internal choice) |
14. $\backslash$ (interrupt) |
15. $P \equiv Q$ (recursive definition) |
16. STOP (deadlock)

**SKIP** ($SKIP$) is the process that terminates successfully. The action prefix (2) ($a \rightarrow P$) is the process that performs action $a$ then acts like process $P$. Processes can be combined using choice. External choice (3) ($P \square P$) is a combinator that lets the environment choose between the processes on offer, whereas with internal choice (4) ($P \sqcap P$) the decision is made by the process itself. Composition of processes can be done using interleaving ($P || P$), which interleaves the actions of each process arbitrarily. The interface parallel (6) ($P ||| P$) process can make progress when the processes combined agree on the actions in set $S$. When $S = \{\}$ (the empty set) the process traces (visible behaviour) are equivalent to that of an interleaved process. When $S$ is non-empty the processes $P$ and $Q$ are forced to agree on events that are in the set $S$ (a subset of the alphabet). Events are channels and can either be read (8) or written to (7). (8) illustrates also parameterised process $P(x)$. Several processes may synchronise on a single communication channel. The channels in the Handel-C are point-to-point. Hiding (9) obscures events from observable behaviour. Hiding, as a programming structure, is used to abstract away unwanted channels. However, hiding is only used in this thesis in proof. The renaming ($f(P)$) function renames the actions of $P$ (10). The divergent process (DIV) immediately divergence and therefore provides no guarantee about its behaviour (11). The internal and external choice operators are generalised in (12) and (13). An interrupt operator is shown in (14). The interrupt operator can be used to pass control from a process $P$ to a processes $Q$. $P \triangledown Q$ states that the process $Q$ can interrupt the process $P$ at any time during the execution of $P$ or not at all. To support the annotations alternate interrupt operators are introduced in chapter 3, which are operationally different from the normal CSP interrupt operator. The recursive definition allows a process to be defined in terms of itself. In (15) the process $P$ is defined by the process $Q$ where $Q$ is itself defined in terms of $P$. The STOP (16) process acts like deadlock. It will not engage in any subsequent events.

The above BNF details the CSP used in this thesis. What follows is the subset of CSP used in the controllers associated with the annotations. We present them this time in their latex form:

$$P ::=$$

1. *SKIP* (successful termination) |
2. $c?y/vz \rightarrow P$ (input/output) |
3. $P \square P$ (external choice) |
4. $P ||| Q$ (interleaving) |
Chapter 1. Background Motivation and Notation

```c
chan struct DATA aa_chan ;
chan struct DATA ww_chan ;
void aa_port(){
  struct DATA aa_var ;
aa_var.o0 = 0 ;
  while(1){
    seq{
      par{aa_chan!aa_var ; ww_chan!aa_var ; /* other comms */}
      par{ /* Code */}
    }
  }
}
```

Figure 1.3.: Illustration of a Handel-C Function

5. $P(x)$
6. $P \setminus x$ (hide x events)
7. $\Delta$ (interrupt)
8. $P(y) = Q(y)$ (recursive definition with a process variable)

1.2.5. Handel-C Hardware Description Language

Handel-C is a hardware description language\(^1\) that has evolved from parallel software languages. Support for parallelism comes from OCCAM, which is a programming language that implements CSP. Only the subset of Handel-C used is described. The `par` and `pri alt` operators provide parallel composition (``prial t has a pre-determined service order). Sequential composition is provided by the `seq` operator. Communication in Handel-C is point-to-point and occurs on a rising edge of the clock. External communication on special channels (`chanin`) may take an indeterminate number of clock cycles to become ready. As with C assignment, for loop, while loop, conditional and switch statements are allowed. As with CSP communication channels are annotated with the `?` and the `!` symbols. To permit a direct mapping to hardware variables are declared with a size in bits. Variables may be arrays. The example in Figure 1.3 illustrates a number of Handel-C language features. In it channels are declared to pass structures (defined elsewhere) of type DATA. A function executes a loop. The loop executes a sequence repeatedly. Firstly, it carries out a number of communications simultaneously, and then runs a parallel block of code (code not defined).

1.2.6. Previous CSP\|B Research

Combining model based formalisms like B [Abr96] and Z [Spi92] with event based formalisms like CSP [Sch00] is an active research area. This thesis is concerned with annotated B,

\(^1\)www.celoxica.com
CSP controllers and the Handel-C implementation language. Some space is devoted to CSP||B in this section as way of motivation. Some space has been devoted to the review of CSP/Z [Fis97] in this thesis, in chapter 7. This is as much CSP and Z consideration that is given. Advances are being made developing the theory for tools to support the Circus approach [Fre05] [SWC02] [CSP03]. Circus is another unification of CSP and Z. Although it is an important development in formal methods it does not have a direct impact on this thesis. Comparisons between the annotated B approach, and CSP combined with Z approaches are made when considering CSP/Z.

The motivation for a closer integration is to obtain a formalism that is as powerful at describing state predicates as it is at describing event behaviour. The CSP||B [Tre00] approach combines CSP and B so that CSP captures, primarily, the event aspect of the design, whereas the B captures the state evolution. Each CSP controller directs a single B machine via communication channels. Controllers may interact with other controllers.

In CSP||B consistency between the pre-conditioned B machine and the CSP controllers is established in two ways. Firstly, by showing that operations are always called within their preconditions, which establishes divergence freedom. Secondly, consistency condition establishes that controllers are deadlock free. Consistency is investigated using the wps of guarded actions [Dij76]. An action system is a pair \(( \text{init} , A ) \), where init is a system initialisation and \( A \) is a set of actions.

Morgan’s failure divergence semantics captures both state based and event based properties of Action Systems [Mor90], and it is ideal for capturing the evolving state of action systems. Using wp semantics action system can be represented in CSP. A guarded command, \( G \rightarrow S \), consist of a guard \( G \) which controls the execution of a statement \( S \). When the guard does not hold the statement is blocked. Morgan describes the wp of the semantics for the guard command in the following way:

**Definition 1 (Morgan’s wp Semantics for Guarded Commands).**

\[
wp(G \rightarrow S , Q) = G \Rightarrow wp(S, Q)
\]

If \( G \) is false in definition 1 then the wp is trivially true, otherwise it is \( wp(S, Q) \) (the weakest precondition such that \( Q \) is satisfied when \( S \) is executed).

The wp of a pre-conditioned abstract system operation \(( \text{PRE} \ P \ \text{THEN} \ T \ \text{END} )\) is given in Treharne [Tre00] and restated here:

**Definition 2 (wp of Operation Guarded with true).**

\[
wp(\text{true} \rightarrow \text{PRE} \ P \ \text{THEN} \ T \ \text{END} , Q) = P \land wp(T, Q)
\]

Preconditioned abstract system operations can be considered to have a guard of true. The wp of the B pre-conditioned statement is the precondition \( P \) conjoined with the wp of the statement \( T \) as described in definition 2. Treharne [Tre00] and this thesis assume that systems are developed under the assumption that operations are always called within their preconditions. The controller for the system must be carefully designed to ensure this. The generation of a consistent controller from annotations is one of the subjects of this thesis. In Treharne’s initial
work, B SELECT statements are avoided to prevent the introduction of blocking statements introduced by them. To understand this consider the wp of a statement $S$ that establishes false ($wp(S, false)$). In B that wp is written $[S]false$. B statements are monotonic hence it can be deduced from false $\Rightarrow P$ that:

$$[S]false \Rightarrow [S]P$$

The initial condition of $S$ that establishes false establishes everything. The B select statement in Generalised Substitution Language (GSL) is $G \Rightarrow S$, “$G$ guards $S$”. The wp of the select statement that establishes $P$ is written $[G \Rightarrow S]P$, and is equivalent to $G \Rightarrow [S]P$. The B select statement that has a false guard establishes false, i.e. $false \Rightarrow S]false$. When the guard is false the select statement miraculously establishes false. The guard is defined as follows:

$$grd(S) = \neg [[S]false]$$

The guards for a select statement are defined differently to that of the preconditioned statement [Abr96] ($Q|S$).

$$grd(G \Rightarrow S) = G \land grd(S)$$

$$grd(Q | S) = Q \Rightarrow grd(S)$$

The select statement blocks execution if the guard is false, whereas the precondition statement aborts (it is not guaranteed to terminate). Select statements are not implementable and are avoided in Treharne's [Tre09] approach. The approach in this thesis is to guarantee execution paths provided proof obligations associated with the annotated path are discharged. The annotations ensure that operations are called within their precondition or guard if that replaces the outer most precondition. Even when there is a choice between to alternative paths the annotations ensure that both are available. This approach ensure the absence of divergence and deadlock regardless of whether precondition or guarded operations are used. However, there is no example of the use of annotations with select in this thesis. The use of annotations in Event-B is considered in chapter 7.

Understanding how to calculate the wp of a statement is an important step in calculating the wp of a sequence of statements. The wp semantics of a sequence of actions is developed inductively. Firstly, the empty sequence of actions does not change the state of the system, as it has no commands. To establish $Q$ with an empty sequence of commands $Q$ must be true before the execution of the empty sequence: $wp(\cdot, Q) = Q$. This constitutes the base case. For a sequence of guarded commands $tr$, where $tr \in A^*$ and $tr$ is not empty, the wp of the sequence is developed inductively as follows:

$$wp(a; tr_a, Q) = wp(a, wp(tr_a, Q))$$

Working back from the base case each action must establish the wp of the following action. Treharne uses Morgan [Mor90] approach that involves giving the B action systems a CSP semantics, which means defining the traces, failures and divergences of the action system.
In our approach the annotations will explicitly state the allowable execution steps. Then two consistency checks must be performed. It must be shown that the annotations are consistent with the operations of the machine (they allow non-divergent next steps) and that the controller operates within the annotation guidelines. It is then not necessary to translate the controller into B and prove that the Control Loop Invariant (CLI) holds. The annotations create proof obligation (PO) requirements that have to be met to guarantee a non-divergence execution. Discharging annotation POs guarantees that the associated preconditions of operations next in the sequence of execution control will be satisfied allowing it to executed without diverging. The NEXT annotation is core to the avoidance of divergence. Discharging a NEXT annotation ensures that an associated execution sequence will always have its precondition satisfied.

Control Loop Invariants

In Treharne [Tre00] a CLI is explicitly defined to reason about non-divergence of looping paths within the controller. B operations are represented as atomic action prefixes in CSP processes. The operations communicate with the outside world and their controlling CSP processes via the input and output of CSP channels. The prefixed action $f \times w \rightarrow R$, reads $x$ into the operation $f$ and outputs $w$ then acts like $R$. Output of one operation can be tunneled into another operation, by fixing the input of one process or action with the output of another in the CSP. Treharne uses $wp$ to investigate safety constraints captured in the CLI. Cyclic behaviour is represented using mutually recursive processes that pass specific control information as process parameters. The loops may be terminating or non-terminating. The communication between the controller and the B machine is handled using channels. The CSP controller can accept inputs from either the B machine or the environment, whereas the B machine accepts inputs from the CSP and output to the CSP. The I/O set up is different in this thesis and defined in chapters 4 and chapter 6. The controllers are captured in a subset of CSP that translates to B. An early B subset and associated translation to CSP is given in [TS00]. The translation mappings are restated in appendix A. The relationship to the annotations has been stated showing how the CSP||B translation framework and the annotation framework compare.

The select statement is avoided because it is impossible to implement its miraculous behaviour. The CSP process that models blocking behaviour is given as follows:

$$p = block \rightarrow STOP$$

The absences of process $p$ demonstrates the absence of deadlock. The above process $p$ issues a $block$ then halts.

The B operation:

$$op = PRE false THEN skip END$$
is divergent if called because if the precondition of an operation is false any behaviour may occur. The process that models this behaviour in the CSP is:

$$DIV$$

As mentioned previously, mutually recursive LOOPS are used to control the sequencing of operations. It is essential that control LOOPS call operators within their preconditions. The CLI establishes that the LOOP will not call any operation outside its precondition and hence will not diverge. Treharne states non-divergence in the following way:

$$traces(LOOP) \cap divergences(M) = \{\}$$

In order to establish the CLI the LOOP has to be translated to B. Treharne's translation rules include ways of representing LOOP and the loop control variables in B. The translation rules are given in table 1.1. In the table the B translation of the equivalent fragment of CSP is given. We write $$\{E_n \rightarrow P\}_P$$ to mean the translation of the CSP fragment $$E_n \rightarrow P$$ in the context of the mapping environment $$\rho$$. In this case the translation is $$n; \{P\}_\rho$$, which means the translation of the process $$P$$ in the environment $$\rho$$ is composed sequentially after operation $$n$$. The action $$E_n$$ in CSP is mapped to the operation $$n$$ in B.

The translation environment $$\rho$$ contains the mappings from B variables to the controller variables. The translation environment is a collection of the mappings between the variables of the B and the variables of the CSP (in the following example $$x_c \mapsto x_3$$). The control loop variable ($$c_b$$) is used in the CLI and is key in the determination of consistency. The control loop variable is equated to one of the CSP variable (ie. $$x_c = c_b$$).

$$\rho(s(p')) = c_b := \rho[p']$$ \text{ where } c_b \text{ is a control variable.}$$

The CSP process parameter $$p'$$ in the LOOP $$s$$ is represented as a variable $$c_b$$ in the B. The value of the parameter is derived from the translation environment $$\rho$$.

We note that in addition to non-divergences the traces of the LOOP must not block. Treharne state that the traces of a correct controller will not contain a block:

$$\neg \exists t \in traces(LOOP) \cdot block \in ran(t)$$

Developing a controller in CSP allows trace behaviour to be analysed independently. Translation from CSP to B permits implementation of the controller within B. On the CSP analysis side the control variables introduce state into the CSP model, which needs to be dealt with carefully in the CSP analysis tools. The approach of this thesis keeps the state in the B machine by modelling the input and output in a slightly different way (chapter 6). Thus, derived controllers can be analysed without reference to a CLI.
In general there are two ways of dealing with I/O. When the temporal organisation of the operations is well established in the designer's mind, chaining I/O can be achieved by writing to, and reading from, global variables. In this way the B invariant checks that the consistency is maintained. This is a very constraining approach. A more flexible approach is achieved by linking consecutive operations output and input explicitly in the annotation, in-line with the philosophy adopted by Treharne. The main difference with the annotation approach is that the output that is passed to an intended following operation is not represented as a control variable in a CSP controller.

The proof obligations that arise after executing the sequence of operations in the controller is given as follows:

Definition 3 (CLI Proof Obligation).

\[(CLI \land I \land c_0 = \rho[p]) \Rightarrow [BBODY_{R_s}]CLI\]

The CLI must hold at all the points of recursion within the loops of the controller. We reproduce the example used to illustrate the CLI, below, and extend the B to illustrate the annotations that would avoid introducing a CLI.

The CLI is constructed from engineering judgement. A control variable is searched for that both captures what the looping action is trying to achieve and ensures the loop does not diverge. In this case we spot that a value is being passed between operations. The value must be in a defined range to avoid loop divergence.

VARIABLES \(V_{in}\)

IN Va RIANT \(V_{in} \in N\)

INITIALISATION \(nn := 0\)

OPERATION

\[in(xx) = PRE xx \in 2..10 \text{ THEN } nn := xx \ END/ * test(nn) NEXT \ */;\]

\[test(xx) = PRE nn = xx \land xx \in 2..10 \text{ THEN skip } END/ * in \ NEXT \ */ \]

The controller for the above machine is given below. Operations (op) in the B machine become actions (\(E_{op}\)) in the CSP controller. The relationship between operation and action is made explicit by subscripting the CSP action with the name of the B machine it calls (\(E_{op}\)). The controller is given as follows:

\[LOOP = S(1)\]

\[S(1) = E_{in}?x_c \in 2..10 \rightarrow S(x_c)\]

\[S(x_c) = E_{test}?x_c \rightarrow S(1)\]

The CLI is stated as follows: \((c_0 > 1 \land c_0 \leq 10) \Rightarrow nn = c_0\)

July 6, 2008
Chapter 1. Background Motivation and Notation

A cut down version of the table in Figure A.1, page 226 without the relationships to annotations is given in Figure 1.1. The translation of the controller fragments to predicate transformers is given. There are three proof obligations that need to be discharged to show that the CLI is maintained by the controller, because the machine is initialised and there are two points of recursion in the controller. The first shows that the CLI can be established by the initial conditions. The second establishes that the first recursive call $S'(1)$ establishes the CLI. Inputs from the environment are treated differently to the inputs given in table 1.1. Here the B ANY WHERE THEN clauses is used to get a fresh input. The third proof obligation results from the last mutually recursive process $S(x_c)$.

The proof seeks to establish that the CLI is maintained by the loop at the point of the recursive call. In the cases that consider the execution of the loop and not the initialisation, the CSP actions are translated to B and the B is substituted into the CLI. To ensure a meaningful substitution the value of the CSP variables are retrieved from the environment.

The translation environment $\rho$ contains the mappings from B variables to the controller variables. The proof seeks to establish that the CLI is maintained by the loop at the point of the recursive call. In the cases that consider the execution of the loop and not the initialisation, the CSP actions are translated to B and the B is substituted into the CLI. To ensure a meaningful substitution the value of the CSP variables are retrieved from the environment.

1. Correct initialisation ($[\text{Initialisation}; c_b := \rho[p]]CLI$):

$$\{nn := 0; c_b := 1\} CLI$$

which holds.

2. The other case is $S'(x)$ which has two sub-cases $S'(x_c)$ and $S'(1)$

2a. Correctness of $S'(1)$:

$$CLI \land I \land (c_b = 1) \Rightarrow [\text{ANY } x_b \text{ WHERE } x_b \in 2..10 \text{ THEN } in(x_b); \quad c_b := \rho \Theta \{x_c \mapsto x_b\}[x_c] \text{ END}]CLI$$

which holds.

2b. Correctness of $S'(x_c)$:

$$CLI \land I \land (c_b = \rho[x_c]) \Rightarrow [\text{test}(\rho[x_c]); \quad c_b := 1]CLI$$

Which simplifies (false implication leaves the precondition of test):

$$((c_b > 1 \land c_b \leq 10) \Rightarrow nn = c_b) \land I \land (c_b = \rho[x_c])) \Rightarrow (\rho[x_c] = nn \land \rho[x_c] \in 2..10)$$

JULY 6, 2008
which holds.

<table>
<thead>
<tr>
<th>CSP</th>
<th>Translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>({E_n \rightarrow P}_\rho)</td>
<td>(n; {P}_\rho)</td>
</tr>
<tr>
<td>({E_n ? x_c \rightarrow P}_\rho)</td>
<td>(n(\rho[x_c]); {P}_\rho)</td>
</tr>
<tr>
<td>({E_n! w_c \rightarrow P}_\rho)</td>
<td>(w_b \leftarrow n; {P}_(w_c \mapsto w_b))</td>
</tr>
<tr>
<td>({E_n! w_c ? x_c \rightarrow P}_\rho)</td>
<td>(w_b \leftarrow n(\rho[x_c]); {P}_(w_c \mapsto w_b))</td>
</tr>
<tr>
<td>({E_n \rightarrow P \parallel Q}_\rho)</td>
<td><strong>CHOICE</strong> (P) <strong>OR</strong> (Q) <strong>END</strong></td>
</tr>
</tbody>
</table>
| \(\{E_n! w_c \rightarrow if\ w_c\ then\ P\ else\ Q\}_\rho\) | \(w_b \leftarrow n;\)  
  |                                          | IF \(w_b\) 
  |                                          | THEN \(\{P\}_(w_c \mapsto w_b)\) 
  |                                          | ELSE \(\{Q\}_(w_c \mapsto w_b)\) 
  |                                          | **END**                                                                   |
| \(\{S(p')\}_\rho\)                      | \(c_b := \rho[p']\)                                                      |

Table 1.1.: B Mappings to CSP

The third proof obligation ensures that the \(in\) operation supplies a parameter satisfying the precondition of \(test\). This we do directly with annotations saving the need to state and discharge the CLI. The annotation of \(in\) states that \(test(nn')\) is next. In the annotations approach input/output piping is verified directly in the B. In this thesis the aim is to develop controllers from the annotated B and keep the state in the B model.

### 1.3. typographic convention

The typographic convention in this thesis is as given tables 1.2.
### Table 1.2: Typographical conventions

<table>
<thead>
<tr>
<th>Convention</th>
<th>Type/symbol</th>
<th>Example/meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>First time B key word introduced in text, or reference made directly to the clause it introduces</td>
<td>CAPITALISE MACHINE</td>
<td></td>
</tr>
<tr>
<td>Names of machines or operations</td>
<td>\textit{latexed}</td>
<td>Traffic</td>
</tr>
<tr>
<td>Annotation names</td>
<td>\textsc{Small Capitals}</td>
<td>\textsc{Next}</td>
</tr>
<tr>
<td>First time a concept introduced in text or used for a while</td>
<td>\texttt{typewriter type}</td>
<td>machine-consistency</td>
</tr>
<tr>
<td>Executable code extracts in text</td>
<td>\texttt{typewriter type}</td>
<td>chan struct \texttt{DATA aa_chan} ;</td>
</tr>
<tr>
<td>Example B machine operations fragments</td>
<td>\textit{latexed}</td>
<td>\textit{\texttt{op \triangleright PRE P_{op} THEN B_{op} END;}}</td>
</tr>
<tr>
<td>B proof obligations notation introduced in BNF</td>
<td>\textit{latexed or \texttt{typewriter type}}</td>
<td>\textit{1. \texttt{SKIP or 1. \texttt{SKIP}}}</td>
</tr>
<tr>
<td>Program code fragments</td>
<td>\texttt{typewriter type}</td>
<td>\texttt{signal_a &lt;= signal_b}</td>
</tr>
<tr>
<td>PO Definitions</td>
<td>\textit{latexed}</td>
<td>\textit{{T(P_j) \land \ldots \land T(P_k)}}</td>
</tr>
<tr>
<td>Definitions</td>
<td>\textit{latexed}</td>
<td>\textit{wp(G -&gt;, Q) = G \rightarrow wp(S, Q)}</td>
</tr>
<tr>
<td>B Key words in program fragments except BVHDL</td>
<td>\textbf{BOLD CAPITALISED}</td>
<td>\textbf{VARIABLE V_{in}}</td>
</tr>
<tr>
<td>BVHDL and BHDL specifications</td>
<td>\texttt{typewriter type}</td>
<td>\texttt{MACHINE aa_SM}</td>
</tr>
<tr>
<td>CSP scripts</td>
<td>\texttt{typewriter type}</td>
<td>\texttt{datatype OPCODE}</td>
</tr>
<tr>
<td>Change of emphasis of a word</td>
<td>\textbf{Bold}</td>
<td>\textbf{Good}</td>
</tr>
</tbody>
</table>

\textit{July 6, 2008}
2. The \textbf{NEXT} Annotation

2.1. Introduction

This chapter's contribution is the introduction of the general framework and first fundamental annotation \texttt{NEXT} which defines the control flow between two operations. The annotation is defined along with the proof obligation to show that it is machine-annotation consistent with the machine. Functions are introduced to help elicit information from the annotated machine to use in the consistency checking and proof. The CSP control fragments associated with the next annotation are introduced along with a set of functions to extract information. In Section 2.2, the general framework is introduced. In Section 2.3 a B machine is introduced along with the \texttt{NEXT} annotation. The proof obligations and control language fragments associated with the annotations are given in subsequent sections.

We restrict our attention in this thesis to correct B machines: those for which all proof obligations (POs) have already been discharged. We use \( I \) to refer to the invariant of the machine, \( T \) to refer to the machine's initialisation, \( P_i \) to refer to the precondition of the operation \( O_i \), and \( B_i \) to refer to the body of that operation. The POs for a consistent machine (machine-consistent) are given in section 2.3.1.

Controllers will be written in a simple subset of the CSP process algebraic language [Hoa85, Sch00], and this language will be explained as it is introduced. Controllers are considered as processes performing events, which correspond to operations in the controlled B machine. Thus operation names will appear in the controller descriptions as well as the B machine definitions.

2.2. The General Framework

The framework proposed in this thesis introduces annotations on B operations as a mechanism for bridging the gap between B machines and CSP controllers, while maintaining the separation of concerns. The approach consists of the following components:

- **Machine Definition**: the controlled component must first be defined and shown to be machine consistent.

- **Annotate Machine**: the initialisation and the operations in the machine definition are annotated with fragments of the requirements for control flow.
• Establish Machine-Annotation Consistency: Discharge POs between definitions and establish consistency of the annotations with the controlled machine. This means that the fragments of control flow captured by the annotations really are appropriate for the machine.

• Define controller: this is a process that describes the overall flow of control for the B machine.

• Establish Annotation-Controller Consistency: establishing that the controller is consistent with the annotations by showing that every part of the controller is supported by some annotation. Annotation-controller consistency is developed in two phases: firstly initial-consistency and secondly step-consistency.

• Refine and translate: refinement may be needed before a translation to Handel-C can be achieved. The translation is the final step and requires additional annotation directives to set type sizes and I/O ports characteristics.

Checking a CSP controller against a machine is thus reduced to checking it against the annotations and verifying that the annotations are appropriate for the machine. The relationship between the different parts of the approach are given in Figure 2.1.

The framework presented here is quite general, in that it may be applied to both Event-B and classical B. Additional annotations may be added along with supporting control operations as
required, provided that a consistency argument can be developed. The first step to be taken is therefore to fix on the control language and the associated annotations to be incorporated into the B machine descriptions.

2.3. The Approach

The approach is demonstrated with a simple model. The annotation considered in this chapter is the NEXT annotation. An extremely simple controller language consisting only of prefixing, choice, and recursion is used to develop the example.

2.3.1. A B Machine

The B-Method [Abr96] has evolved two major approaches: classical B and Event-B. Annotations can be used in either classical B machines, or Event-B systems. Classical B approaches specification in a state-oriented fashion. It focuses on the services that a system might provide, whereas Event-B focuses on the events that occur within the system. The generic classical B machine $M$, given in Figure 2.2, has variables, invariant, initialisation, and a set of operations $OP_1$ through to $OP_n$. The fundamental principles are introduced without I/O for simplicity. I/O is introduced in chapter 4.

```
MACHINE M
VARIABLES v
ININVARIANT v : t
INITIALISATION v : ∈ u
OPERATIONS
OP_1 ≜ P_1 \mid B_1;
OP_2 ≜ G_2 \Rightarrow B_2;
...
OP_n ≜ P_n \mid B_n
END
```

Figure 2.2.: A B Machine.

Classical B is used in this thesis. However, a guarded operation ($OP_2$) has been illustrated in Figure 2.2 to show the Event-B stylistic approach. In Figure 2.2 the machine operations are defined in Guarded Substitution Language (GSL). In classical B all operations must be preconditioned ($P$). Annotated B is derived by taking a B specification and adding the annotations (the annotations are not shown in Figure 2.2) introduced in the following chapters. The machine operations must be shown to be machine-consistent.
In classical B machine-consistency with respect to a particular operation is established by discharging a number of PO. We will assume for this thesis that machines have had these proof obligations discharged. Predicate (1) states that there exists values for the variables (v) of the machine that make the machine invariant \( I \) true. Central to the proofs in this thesis are predicates (2) and (3) in definition 4. Predicate (2) states that the initialisation \( T \) of the B machine establishes the invariant \( I \) of the B machine. The predicate transformer \([B]I\) produces a new predicate in which the substitutions of \( B \) are made in \( I \). PO (3) is read as follows: given a state in which the invariant \( I \) and the precondition \( P \) hold it follows that the invariant transformed with \( B \) holds. The POs in Definition 4 do not include the POs associated with machine constraints, constants or properties as they are not used in examples with annotations in this thesis. More details on these can be found in B texts [Abr96] [Sch01].

2.3.2. The Use of the term Consistency

In the following chapters the following consistency terms are used:

- **machine consistency** refers to B machines, and means that all the consistency proofs given in definition 4 have been discharged.

- **refinement consistency** refers to B machine refinements, and means that all the consistency proofs given in definition 6.0.1 have been discharged.

- **machine-annotation consistency** refers to the consistency between machines and their annotations, and is established by discharging the PO associated with each annotation in the machine.

- **annotation-controller consistency** refers to the consistency between the annotations and the controller, and is established by showing annotation consistency.

- **annotation consistency** is the term used for a controller for which initial-consistency and step-consistency has been demonstrated.

- **initial-consistency** exists if the initial actions of the controller are enabled by the annotations of the initialisation of the B machine.

- **step-consistency** exists when all actions of the controller are enabled by the B machine

- **machine-controller consistency** refers to the consistency between the machine and its controller, it is established by showing machine consistency, machine-annotation consistency and annotation consistency.

**Definition 4 (Machine Consistency).**

1. \( \exists v \cdot I \)

2. \([T]I\)

3. \( P \land I \Rightarrow [B]I \)
In Event-B, unlike classical B, new operations can be added during refinement. In the examples in chapter 6 the need for operations in the later stages of refinement are anticipated by introducing the signature of the operation with a body defined by the *skip* operator. The POs for Event-B refinement are not adapted. The refinement process may involve adding detail to the specification in a consistent way to realise an implementation, which is a key notion in B. After introducing the annotations it is shown how annotations can be refined. Refinement involves removing non-determinism and adopting concrete types. We add to the concept of B refinement with the annotations, by adding the notion of annotation control flow refinement.

2.4. The NEXT Annotation of a Consistent B Machine

The NEXT annotation is written in B comments as follows: /* { list of operations } NEXT */ (the short hand form of the annotation is /* { list of operations } */). They are used to make the intended execution of a machine’s operations explicit to a reviewer; or user of the machine; or synthesis tool. An annotation is associated with an operation in the same way that a precondition is. This contrasts with global predicates like the invariant. Annotations capture which operation the designer intends to be enabled after the current operation has terminated. To ensure that there is no deadlock in the machine every operation must have a NEXT annotation including the INITIALISATION clause. An example of its use in an arbitrary operation \( OP_i \) is:

Definition 5 (NEXT Annotation Syntax for Operations).

\[
OP_i \equiv (P_i \mid B_i) / \ast \{ OP_j, \ldots, OP_k \} \text{NEXT} \ast / 
\]

In definition 5 we state that after the execution of \( OP_i \) has completed \( OP_j \), through to \( OP_k \), are available to execute. The proof of this claim can be verified by discharging the proof obligation in definition 6:

Definition 6 (NEXT Proof Obligation for Operations). The operation

\[
OP_i \equiv (P_i \mid B_i) / \ast \{ OP_j, \ldots, OP_k \} \text{NEXT} \ast / 
\]

has the following PO:

\[
(I \land P_i \Rightarrow [B_i](P_j)) \quad \land \\
\quad \ldots \quad \land \\
(I \land P_i \Rightarrow [B_i](P_k))
\]

Definition 7 (NEXT Annotation Syntax for Initialisations).

\[
\text{INITIALISATION T} / \ast \{ OP_j, \ldots, OP_k \} \text{NEXT} \ast / 
\]

---

The range of next annotations are expanded in chapter 9 to include NEXT SEQUENCE and INTERLEAVED NEXT at which point an operation must have either a NEXT, NEXT SEQUENCE or INTERLEAVED NEXT annotation.

July 6, 2008
Definition 8 (Proof Obligation for Initialisation). The initialisation

\[ INITIALISATION \quad T \quad \ast \quad \{ \text{OP}_1, \ldots, \text{OP}_k \} \quad \text{NEXT} \quad \ast \]

has the following PO:

\[ [T](\text{OP}_j) \land \ldots \land [T](\text{OP}_k) \]

where \( T \) represents the body of the initialisation.

The \textsc{next} annotation for the initialisation and POs are given in definitions 7 and definition 8, respectively.

### 2.5. Controller Syntax and Auxiliary Functions for \textsc{next} Annotations

We start with a simple control syntax in order to develop the framework.

**Definition 9 (Controller Syntax).** Syntactically a basic controller \( R \) is given in the following:

\[
R ::= \begin{array}{c}
\text{a} \rightarrow R \\
R \Box R \\
S(p)
\end{array}
\]

Recursive definitions with a process variable has the following syntax:

\[ S(p) \triangleq R(p) \]

where \( S \) if the process name and \( p \) is the process variable. \( R \) is the recursive definition of \( S \) that is itself defined as above.

In definition 9 event \( \text{a} \) represents a machine-consistent operation execution (a CSP action represents a B machine operation). The prefix \( \text{a} \) followed by \( R \) is valid controller syntax, that means after \( \text{a} \) has occurred the controller is prepared to act like \( R \). The choice between two well formed controller fragments \( R_1 \) and \( R_2 \) is written \( R_1 \Box R_2 \). A recursive definition with a process variable is written \( S(p) \). All basic controllers can be composed with this syntax.

The meaning of annotation consistency is developed using the functions \textit{guarded} and \textit{init} defined on CSP processes. The functions are used in this and subsequent sections to calculate consistency. The \textit{guard} function checks that control loops are built up from action prefixes. This ensures that every operation has a next operation it can legally flow to next. \textit{init} is always non-empty for guarded controller fragments, which is essential as a basis for non-divergence.
2.5. Controller Syntax and Auxiliary Functions for NEXT Annotations

**Definition 10 (guarded on CSP Controller Process).**

\[
\text{guarded}(a \to R1) = \text{true} \\
\text{guarded}(R1 \sqcup R2) = \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(S(p)) = \text{false}
\]

We only consider recursive definitions whose definition is guarded, i.e.:

\[ S(p) \equiv R(p) \text{ where } \text{guarded}(R(p)) = \text{true} \]

The set of initial actions of a controller fragment are extracted with the \textit{init} function which is given in definition 11.

**Definition 11 (init on CSP Controller Process).**

\[
\text{init}(a \to R1) = \{a\} \\
\text{init}(R1 \sqcup R2) = \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(S(p)) = \text{init}(R(p))
\]

where recursive definitions with a process variable has the following form:

\[ S(p) \equiv R(p) \]

For example:

\[ \text{init}(\text{LOOP}) = \{a\} \]

if \( \text{LOOP} \triangleq a \to b \to \text{LOOP} \)

The \textit{init} function plays a key role in the proof that a machine-controller combination does not diverge. The \textit{init} function returns a set that represents the first elements of a given controller fragment. The \textit{init} function is used to develop a relationship between the controller fragments and NEXT annotations of operations.

We introduce a function to extract the set of operations associated with a NEXT annotations in an operation comments to help define consistency.

**Definition 12 (NEXT Annotation Listing).**

\[ \text{next}(OP_i) = \{OP_j, ..., OP_k\} \text{ if } OP_i \triangleq (P_i \mid E_i) / \{OP_j, ..., OP_k\} \text{ NEXT } /; \]

\text{undefined otherwise}
The next listing function defined in 12 lists the operations that are enabled by the current operation. The init and guard functions are applied to fragments of controllers or operations. The dual nature of the next function arises because operations and actions are the same thing in the different models. Hence, next(a) and next(a ≡ (P_a | B_a)/ * X NEXT */) produce the same result. If the next function is applied to an action then the action is unconditionally converted into its equivalent B operation. This holds true of all the listing functions defined in future chapters.

2.6. Annotation Consistency with Action-Prefix

Establishing annotation consistency is the second step in the development process (see Figure 2.1) after establishing machine-annotation consistency. Annotation consistency requires establishing initial-consistency and step-consistency. Initial consistency is concerned with the first step, and step-consistency with subsequent steps.

The notion of initial-consistency is developed over the first action of the controller fragment. It must be shown that the first action of the controller (init(M_CTRL)) is enabled by the B machine. The machine M has an initialisation clause that has to execute initially to get the machine into a state that satisfies the invariant of the machine. The first action of M_CTRL must be enabled by the initialisation of M. Annotation consistency for controllers that contain the basic control elements is defined in two parts: definition 13 (initial-consistency) and definition 14 (step consistency).

Definition 13 (Annotation Consistency with Action-Prefix).

1. a → R is annotation consistent with Machine M if a → R is initially consistent with M and a → R is step-consistent with M.

   where a → R is initially consistent if a ∈ next(INITIALISATION)

2. R1 ⊗ R2

   is annotation consistent with Machine M if R1 and R2 are annotation consistent with M.

3. S(p)
is annotation consistent with Machine $M$ if $R(p)$ is annotations consistent.

where $S$ if the process name and $p$ is the process variable. $R$ is the recursive definition of $S$ that is defined as above.

$S(p)$ is a family of recursive definitions $S(p) \equiv R(p)$.

In definition 13 case 1 the fragment $a \rightarrow R$ is initially-consistent if $a$ is enabled by the machine initialisation ($a \in \text{next(INITIALISATION)}$), and step-consistency is check with definition 14. In case 2 both choices offered must be annotation consistency. In case 3 recursive calls are annotation consistent if their recursive definition is annotation consistent.

Annotation consistency is evaluated by testing the current fragment for consistency and breaking down the current fragment and evaluating the parts. The $S(p)$ operator is the last operator in a control branch. If the evaluation reaches this point in a branch then the evaluation beyond this point is assumed to be annotation consistent (as it is recursively defined), and the actual annotation consistency of that branch is then only dependent on the fragments that make up that branch.

Definition 14 (Step-Consistency with Action Prefix).

1. $b \rightarrow R$

   is step-consistent with Machine $M$ if $\text{init}(R) \subseteq \text{next}(b)$ and $R$ is step-consistent with $M$.

2. $R1 \square R2$

   is step-consistent with $M$ if $R1$ and $R2$ are step-consistent with $M$.

3. $S(p)$

   is step-consistent with Machine $M$,

   if $R(p)$ is step-consistent where $S(p) \equiv R(p)$.

July 6, 2008

29
2.7. Proving Termination of Controlled Machines with \texttt{NEXT} Annotations

In this section demonstrating annotation-controller consistency is discussed. The proofs in this section establish that given machine-annotation consistency and annotation consistency it can be shown that every step of the machine-controller pair terminates, which implies non-divergence: every operation is called within its precondition. The proofs in this section relate to \texttt{NEXT} annotations and controllers with prefix actions. The notation of good traces are introduced to assist with proof. A good trace is a CSP trace that executes operations that at each step terminates, passing control successfully to the next operation in the trace as the trace evolves. A good trace is a trace that is not a divergence of \( M \).

Definition 15 (Good Traces \( tr \) of Consistent Controllers Terminate). \( tr \) is a good trace with respect to machine \( M \) if the following holds:

\[
\text{[initialisation; } tr\text{]} \text{ true}
\]

where initialisation is the initialisation of the \( B \) machine.

Theorem 1 (\texttt{NEXT} Annotated Machine Controller Combinations that are Machine-Annotation Consistent and Annotation Consistent are Divergence Free). If a \texttt{NEXT} annotated \( B \) machine \( M \) is machine-annotation consistent and the controller \( M\_CTRL \) is annotation consistent with the annotations of machine \( M \) then \( M \parallel M\_CTRL \) is divergence free.

To prove Theorem 1 we show that only good traces are produced by \( M\_CTRL \). We will prove by induction on the length of \( tr \) that if the annotations of \( M \) (that include \texttt{NEXT}) are machine-annotation consistent and \( M\_CTRL \) is annotation consistent and \( tr \) is a trace of \( M\_CTRL \) then \( tr \) is good. Hence \( M \parallel M\_CTRL \) it is divergence free as the controller can never drive an operation of \( M \) outside its precondition. We prove this by induction over cases of traces of \( M\_CTRL \).

Case (A) (): Initialisation

1. \([T]I\) by consistency of \( M \)
2. \([T]\text{true} \) by \([T]I \Rightarrow [T]\text{true} \)

Case (B) \((a) \in \text{traces}(M\_CTRL)\): A single operation occurs.
2.7. Proving Termination of Controlled Machines with \textsc{next} Annotations

1. \( \langle a \rangle \in \text{traces}(M_{CTRL}) \) and \( M_{CTRL} \) annotation consistent

2. \( a \in \text{next}(\text{INITIALISATION}) \) \text{ by 1 and def. of Lemma 1 below}

3. \([T]P_a \) \text{ by 2 and consistency of annotations 8}

4. \([T]I \) \text{ by def. of machine-consistency 4}

5. \([T](P_a \land I) \) \text{ by 3 and 4}

6. \([T](P_a \land I \land [B_a]I) \) \text{ by def. of machine-consistency 4: } P_a \land I \Rightarrow [B_a]I

7. \([T](P_a \land B_a)\) \text{ by } I \Rightarrow \text{true}

8. \([T][P_a | B_a]\text{true} \) \text{ by definition of } P ! B

9. \([\text{init}; a]\text{true} \) \text{ by definition of sequencing in } B ;

10. \( \langle a \rangle \) is good

The proof of theorem 1, case B, starts by restating initial-consistency, which implies that the singleton event is an element of the B initialisation \textsc{next} annotation. The rules of consistency are used to build up a sequence of operations (init; a) that establish true, which signifies that the sequence terminates and therefore does not diverge. The empty trace and the trace containing one action are special cases. In general traces contain arbitrary number of actions (case C), and it is necessary to prove that an arbitrary consistent trace terminates (is non-divergent). We need to prove that an arbitrary trace of a consistent machine controller pair \( M\parallel M_{CTRL} \) is non-divergent. We can assume that \( tr \sim \langle a, b \rangle \in \text{traces}(M_{CTRL}) \) and that \( M_{CTRL} \) is consistent (controller consistent) and \( M \) is machine consistent.
Chapter 2. The NEXT Annotation

Case (C) \( \langle a, b \rangle \in \text{traces}(M\text{-\text{CRTL}}) \) where \( tr \in \text{traces}(M\text{-\text{CRTL}}) \)

1. \( tr \ ^\ast \langle a \rangle \) is a trace of \( M\text{-\text{CTRL}} \) by prefix closure of traces

2. \( tr \ ^\ast \langle a \rangle \) is good by 1 and inductive hypothesis

3. \( b \in \text{next}(a) \) Lemma 2 below

4. \( P_a \land I \Rightarrow [B_a]P_b \) def. of annotations 6

5. \( [\text{init}; tr; (P_a \mid B_a)]true \) by 1 and def. of good 15

6. \( [\text{init}; tr](P_a \land [B_a]true) \) by 5 and def. of \( \mid \) and wp semantics of \( ; \)

7. \( [\text{init}; tr]I \) by def. of machine consistency

8. \( [\text{init}; tr]I \land P_a \) by 6 and 7

9. \( [\text{init}; tr][B_a]P_b \) by 8 and 4

10. \( [\text{init}; tr]P_a \land [B_a]P_b \) by 9 and 6

11. \( [\text{init}; tr; (P_a \mid B_a)]P_b \) by 10 and def. of \( \mid \) and wp semantics of \( ; \)

12. \( [\text{init}; tr; (P_a \mid B_a)]I \) by def. of machine consistency

13. \( P_b \land I \Rightarrow [B_b]I \) by def. of machine consistency

14. \( [\text{init}; tr; (P_a \mid B_a)]I \land P_b \) by 11 and 12

15. \( [\text{init}; tr; (P_a \mid B_a)][B_b]I \) by 13 and 14

16. \( [\text{init}; tr; (P_a \mid B_a)][P_b \land [B_b]I] \) by 15 and 11

17. \( [\text{init}; tr; (P_a \mid B_a); (P_b \mid B_b)]I \) by 16 and def. of \( \mid \) and wp semantics of \( ; \)

18. \( [\text{init}; tr; (P_a \mid B_a); (P_b \mid B_b)]true \) by 17 and def. of machine consistency

19. \( tr \ ^\ast \langle a \rangle \ ^\ast \langle b \rangle \) is good by 18 and def. of machine consistency

In the proof of theorem 1, case C, we start with the inductive hypothesis that the sub-trace \( tr \ ^\ast \langle a \rangle \) is good. Lemma 2 allows us to relate good controller traces to machine annotations, which gives access to the associated proof obligations. The PO predicates are used to show that the arbitrary trace is good.
Singleton Traces are Enabled by Initialisation Annotations

Lemma 1 (Singleton Traces are Enabled by Initialisation Annotations). Any action of a singleton trace of annotation consistent controller $R$ is also an initialisation annotation:

$$R \text{ initially-consistent } \land (a) \in \text{traces}(R) \Rightarrow a \in \text{next}(\text{INITIALISATION})$$

Proof of Lemma 1 by structural induction over case of controller fragments:

case 1 $c \rightarrow R$

1. $(a) \in \text{traces}(c \rightarrow R)$
   initial assumption
2. $(c \rightarrow R)$ initially-consistent
   initial assumption
3. $c \in \text{next}(\text{INITIALISATION})$
   2, annotation consistency def. 13
4. $a = c$
   1 and def. of CSP traces
5. $a \in \text{next}(\text{INITIALISATION})$
   3,4

QED

case 2 $R_1 \square R_2$

1. $(a) \in \text{traces}((R_1) \square (R_2))$
   initial assumption
2. $(R_1 \square R_2)$ initially-consistent
   initial assumption
3. $(a) \in \text{traces}(R_1) \text{ or } (a) \in \text{traces}(R_2)$
   by def. of CSP traces
4 sub-case: $(a) \in \text{traces}(R_1)$

5. $(R_1)$ initially-consistent
   2, def. initial consistency 13
6. $a \in \text{next}(\text{INITIALISATION})$
   sub-case 4,5, ind. hypothesis on $R_1$

7 sub-case: $(a) \in \text{traces}(R_2)$

8. $(R_2)$ initially-consistent
   2, def. annotation consistency
9. $a \in \text{next}(\text{INITIALISATION})$
   sub-case 7,8, ind. hypothesis on $R_2$

QED

case 3 $S(p)$

1. $S(p)$ initially-consistent
   initial assumption
2. $(a) \in \text{traces}(S(p))$
   initial assumption
3. $R(p)$ initially-consistent
   1, def. annotation consistency
4. $(a) \in \text{traces}(R(p))$
   2, $S(p) \equiv R(p)$
5. $(a) \in \text{next}(\text{INITIALISATION})$
   3,4, ind. hypothesis on $R(P)$

QED
In the proof of lemma 1 it is assumed that a controller fragment $R$ is initially-consistent and that the singleton trace emanates from $R$. In the case of the prefixed fragment the initial assumption states that the prefix is a member of the INITIALISATION NEXT annotation. In the case of the fragment offering a choice both choices are analysed. The proof takes advantage of the fact that a sub-fragment is initially consistent.

**Arbitrary Traces are Enabled by Annotations**

Lemma 2 (Arbitrary length traces of annotation consistent controllers are enabled by annotations). The actions of arbitrary length traces of annotation consistent controllers are enabled by operation annotations:

$$\langle R \text{ annotation consistent consistent} \land tr \cdotp (a) \cap (b) \in \text{traces}(R) \rangle \Rightarrow b \in \text{next}(a)$$

Proof of Lemma 2 by Cases of controllers:

<table>
<thead>
<tr>
<th>Case</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>$c \rightarrow R$</td>
</tr>
<tr>
<td>1</td>
<td>$c \rightarrow R$ step-consistent with $M$</td>
</tr>
<tr>
<td>sub-case</td>
<td>$tr = \emptyset$</td>
</tr>
<tr>
<td>2</td>
<td>$tr \cdotp (a) \cap (b) \in \text{traces}(c \rightarrow R)$</td>
</tr>
<tr>
<td>3</td>
<td>$(a) \cap (b) \in \text{traces}(c \rightarrow R)$</td>
</tr>
<tr>
<td>4</td>
<td>$a = c$ and $(b) \in \text{traces}(R)$</td>
</tr>
<tr>
<td>5</td>
<td>$b \in \text{init}(R)$</td>
</tr>
<tr>
<td>6</td>
<td>$b \in \text{next}(c)$</td>
</tr>
<tr>
<td>7</td>
<td>$b \in \text{next}(a)$</td>
</tr>
</tbody>
</table>

Initial assumption

QED

sub-case | $tr \neq \emptyset$

| 9    | $tr \cdotp (a) \cap (b) \in \text{traces}(c \rightarrow R)$ |
| 10   | let $tr = (e) \circ tr'$ |
| 11   | $(e) \circ tr' \cdotp (a) \cap (b) \in \text{traces}(c \rightarrow R)$ |
| 12   | $e = c \land tr' \cdotp (a) \cap (b) \in \text{traces}(R)$ |
| 13   | $tr' \cdotp (a) \cap (b) \in \text{traces}(R)$ |
| 14   | $R$ step-consistent with $M$ |
| 15   | $b \in \text{next}(a)$ |

13, 14, inductive hypothesis on $R$

QED
2.7. Proving Termination of Controlled Machines with \texttt{next} Annotations

\textbf{case B} \quad \text{\texttt{R1} \& \texttt{R2}}

1. \(tr \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{R1} \& \texttt{R2})\) \quad \text{initial assumption}
2. \(\texttt{R1} \& \texttt{R2} \text{ step consistent with } M\) \quad \text{initial assumption}
3. \(\texttt{R1} \text{ step-consistent with } M\) \quad 2, consistency def. 14
4. \(\texttt{R2} \text{ step-consistent with } M\) \quad 2, consistency def. 14
5. \(tr \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{R1}) \text{ or } \text{tr} \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{R2})\) \quad 1, def. trace

\text{sub-case: } tr \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{R1})

6. \(b \in \text{\texttt{next}}(a)\) \quad 3, inductive hypothesis on \texttt{R1}

\text{sub-case: } tr \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{R2})

7. \(b \in \text{\texttt{next}}(a)\) \quad 4, inductive hypothesis on \texttt{R2}

\text{QED}

\textbf{case C} \quad \text{\texttt{S(p)}}

1. \(tr \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{S(p)})\) \quad \text{initial assumption}
2. \(\texttt{S(p)} \text{ step-consistent with } M\) \quad \text{initial assumption}
3. \(tr \uparrow \langle a \rangle \uparrow \langle b \rangle \in \text{traces}(\texttt{R(p)})\) \quad \(\texttt{S(p)} \equiv \texttt{R(p)}\)
4. \(\texttt{R(p)} \text{ step-consistent with } M\) \quad 2, def. consistency 14
5. \(b \in \text{\texttt{next}}(a)\) \quad 3, 4, inductive hypothesis on \texttt{R(p)}

\text{QED}

\textbf{The Initial Action of a Consistent Controllers}

\textbf{Lemma 3 (Elements of machine-controller consistent singleton traces are in } init(R)).\textbf{)}

An element of the singleton trace of an annotation consistent controller fragment \(R\) which controls a machine-annotation consistent machine is contained in the init of \(R\):

\[ R \text{ annotation consistent and } \langle b \rangle \in \text{traces}(R) \Rightarrow b \in \text{init}(R) \]

\textbf{Proof of Lemma 3 by cases of controllers:}

\textbf{July 6, 2008}
Chapter 2. The next Annotation

case A  
\[ c \rightarrow R \]

1. \( (b) \in \text{traces}(c \rightarrow R) \) initial assumption
2. \( b = c \) 1, def. CSP traces
3. \( c \in \text{init}(c \rightarrow R) \) def. of init 11
4. \( b \in \text{init}(c \rightarrow R) \) 2, and 3  
\[ \text{QED} \]

case B  
\[ R1 \square R2 \]

1. \( (b) \in \text{traces}(R1 \square R2) \) initial assumption
2. \( (b) \in \text{traces}(R1) \) or \( (b) \in \text{traces}(R2) \) 1, def. of CSP traces
3. sub-case: \( (b) \in \text{traces}(R1) \)
4. \( b \in \text{init}(R1) \) 3, inductive hypothesis on R1
5. \( b \in \text{init}(R1 \square R2) \)
6. sub-case: \( (b) \in \text{traces}(R2) \)
7. \( b \in \text{init}(R2) \) 6, inductive hypothesis on R2
8. \( b \in \text{init}(R1 \square R2) \)  
\[ \text{QED} \]

case C  
\[ S(p) \]

1. \( (b) \in \text{traces}(S(p)) \) initial assumption
2. \( (b) \in \text{traces}(R(p)) \) \( S(p) \triangleq R(p) \)
3. \( b \in \text{init}(R(p)) \) 2, inductive hypothesis \( R(p) \)
4. \( b \in \text{init}(S(p)) \) \( S(p) \triangleq R(p) \)  
\[ \text{QED} \]

2.8. Picturing of next Annotation

A next annotation of an operation (definition 5) considered in isolation, is pictured as in Figure 2.3. Any transition by the current operation will lead to a state that can execute any of the operations from \( op_j \) through to \( op_k \).
2.9. Examples Utilising \textsc{Next}

The \textsc{Next} annotation will be used commonly when developing operations in B machines. This annotation captures execution sequences. The next step or choices between several next steps can be captured.

A simple traffic control system for the main street of the walled City of Carcassonne is specified. The main street is narrow and is heavily used by tourists and some motor vehicles brave enough to edge through the crowds. The system must allow traffic up into the city market square from the moat or down from the square to the moat gate along the same single width road. Ideally the system should allow time for motor vehicles to clear the road before changing direction and give the tourist time to wander on the road without traffic, but the details of timing have been abstracted away. One of the traffic lights is set to go (\textit{Aller}) at a time then turn to stop (\textit{Arret}). A choice is offered as to which light goes next. The Traffic machine has a \textsc{Sets} clause that describes the set \textit{Command} that can command \textit{Arret} or \textit{Aller}. The state \textsc{Variables} are \textit{Moat} and \textit{Square}, which represent traffic lights. The \textsc{Invariant} types the state variables asserts that either \textit{Moat} or \textit{Square} must be set to \textit{Arret}. The \textsc{Initialisation} sets \textit{Moat} and \textit{Square} to \textit{Arret}. The \textsc{Initialisation} annotations requires that the first command to be executed shall be \textit{Arret-All}. The \textit{Arret-All} operation sets the traffic lights to \textit{Arret}. The \textit{Aller-Moat} sets the \textit{Moat} traffic light to go when both lights are \textit{Arret}. The \textit{Aller-Square} sets the \textit{Square} traffic light to go when both lights are \textit{Arret}. The machine fulfils the requirement that traffic only travels in one direction.

A B machine that allows an external party to choose between the traffic flows is given in Figure 2.4. Its derived controller is given in Figure 2.5. The consistency PO of the Traffic machine 2.4 and controller 2.5 is calculated by first showing the POs hold then showing the annotations enable the controller sets. The Traffic machine POs are discharged in Figure 2.6, which shows that the machine is machine-annotation consistent. The consistency check is undertaken by stepping through each annotation path until each annotation jump has been checked for consistency once. The consistency of the controller with respect to the annotations is given in the proof detailed in Figure 2.7. The proof

\textsc{July 6, 2008}
MACHINE Traffic
SETS COMMAND = {Arret, Aller}
VARIABLES Moat, Square
INvariant
   Moat : COMMAND \wedge Square : COMMAND \wedge (Moat = Arret \lor Square = Arret)
INITIALisation Moat, Square := Arret, Arret /* {Arret-All}NEXT */

OPERATIONS

Arret-All \equiv \text{PRE} \text{ True THEN } Moat := Arret \land Square := Arret \text{ END}
   / */ \{Arret-Moat, Arret-Square\}NEXT */;

Aller-Moat \equiv \text{PRE} \text{ Moat = Arret \land Square = Arret THEN } Moat := Aller \text{ END}
   / */ \{Arret-All\}NEXT */;

Aller-Square \equiv \text{PRE} \text{ Moat = Arret \land Square = Arret THEN } Square := Aller \text{ END}
   / */ \{Arret-All\}NEXT */

END

Figure 2.4.: Traffic Machine

Traffic_CTRL = S_CTRL

S_CTRL = Arret-All \rightarrow ((Aller-Moat \rightarrow S_CTRL) \Box
   (Aller-Square \rightarrow S_CTRL))

Figure 2.5.: Traffic Controller

is built up from the process variables that are initially- and step- consistent by definition.
2.9. Examples Utilising NEXT

1 INITIALISATION PO : \[ T \] P_{Arret\_All} \\
\equiv [\text{Moat}, \text{Square} := \text{Arret}, \text{Arret}] \text{True} \\
\equiv \text{True}

2 Arret\_All operation PO : \( I \land P_{Arret\_All} \Rightarrow [B_{Arret\_All}] P_{Aller\_Moat} \) \\
\equiv (\text{Moat} : \text{COMMAND} \land \text{Square} : \text{COMMAND} \land (\text{Moat} = \text{Arret} \lor \text{Square} = \text{Arret})) \Rightarrow \\
[\text{Moat} := \text{Arret} \land \text{Square} := \text{Arret}] (\text{Moat} = \text{Arret} \land \text{Square} = \text{Arret}) \\
\equiv \text{True}

3 Arret\_All operation PO : \( I \land P_{Arret\_All} \Rightarrow [B_{Arret\_All}] P_{Aller\_Square} \) \\
\equiv (\text{Moat} : \text{COMMAND} \land \text{Square} : \text{COMMAND} \land (\text{Moat} = \text{Arret} \lor \text{Square} = \text{Arret})) \Rightarrow \\
[\text{Moat} := \text{Arret} \land \text{Square} := \text{Arret}] (\text{Moat} = \text{Arret} \land \text{Square} = \text{Arret}) \\
\equiv \text{True}

4 Aller\_Moat operation PO : \( I \land P_{Aller\_Moat} \Rightarrow [B_{Aller\_Moat}] P_{Arret\_All} \) \\
\equiv (\text{Moat} : \text{COMMAND} \land \text{Square} : \text{COMMAND} \land (\text{Moat} = \text{Arret} \lor \text{Square} = \text{Arret})) \land \\
(\text{Moat} = \text{Arret} \land \text{Square} = \text{Arret}) \Rightarrow [\text{Moat} := \text{Aller}] \text{True} \\
\equiv [\text{Moat} := \text{Aller}] \text{True} \\
\equiv \text{True}

5 Aller\_Square operation PO : \( I \land P_{Aller\_Square} \Rightarrow [B_{Aller\_Square}] P_{Arret\_All} \) \\
\equiv (\text{Moat} : \text{COMMAND} \land \text{Square} : \text{COMMAND} \land (\text{Moat} = \text{Arret} \lor \text{Square} = \text{Arret})) \land \\
(\text{Moat} = \text{Arret} \land \text{Square} = \text{Arret}) \Rightarrow [\text{Square} := \text{Aller}] \text{True} \\
\equiv [\text{Square} := \text{Arret}] \text{True} \\
\equiv \text{True}

Figure 2.6.: Traffic Machine POs
The goal is to show that the Traffic_CTRL is annotation-controller consistent

1. $S_{-CTRL}$ step-consistent
   - by definition 14

2. $\text{init}(S_{-CTRL}) \subseteq \text{next}(\text{Aller\_Square})$
   - by inspection

3. $(\text{Aller\_Square} \rightarrow S_{-CTRL})$ step-consistent
   - by 1, 2 and definition 14

4. $\text{init}(S_{-CTRL}) \subseteq \text{next}(\text{Aller\_Moat})$
   - by inspection

5. $(\text{Aller\_Moat} \rightarrow S_{-CTRL})$ step-consistent
   - by 1, 4 and definition 14

6. $(\text{Aller\_Moat} \rightarrow S_{-CTRL}) \square$
   - $(\text{Aller\_Square} \rightarrow S_{-CTRL})$ step-consistent
   - by 3, 5 and def 14

7. $\text{init}((\text{Aller\_Moat} \rightarrow S_{-CTRL}) \square$
   - $(\text{Aller\_Square} \rightarrow S_{-CTRL}) \subseteq \text{next}(\text{Arret\_All})$
   - by inspection

8. $\text{Arret\_All} \rightarrow$
   - $(\text{Aller\_Moat} \rightarrow S_{-CTRL}) \square$
   - $(\text{Aller\_Square} \rightarrow S_{-CTRL})$
   - step-consistent
   - by 6, 7 and definition 14

9. $\text{Traffic\_CTRL}$ step-consistent
   - by 1 and definition 14

QED

Figure 2.7.: Traffic Controller Consistency
3. The FROM and QUERY Annotations

The contribution of this chapter is adding an interrupt and query annotation, and the interrupt CSP operator to the control language. The query annotation indicates that the operation it annotates does not change the state of the machine. Adding new annotations to the framework follows the same definitional process set out in chapter 2. The three interrupt annotations will be introduced in the following sections: the FROM-ANY, FROM-SET, and FROM-SET-INIT. These annotations are used with the NEXT annotation in defined combinations. We do not rely on the from annotations to establish initial-consistency. There must be a NEXT annotation in the INITIALISATION and in every operation. In this thesis we restrict the annotations in operations to, at most, one from annotation accompanied with, at least, one next type annotation. The FROM-ANY and NEXT annotations produce very similar POs. In fact the from annotations can be expressed in terms of the NEXT annotation. This is not done for the following reasons:

- It is clearer when the annotation is put in as few places as necessary.
- The FROM I/O annotation is implemented in a different way to the NEXT annotation.
- FROM signifies a difference to normal looping behaviour defined by the NEXT annotations.

3.1. The FROM annotations

3.1.1. The FROM-ANY Annotation of a Consistent B Machine

The FROM-ANY annotation (the short form of the annotation is /* ! {*/ *//) is added to an operation to indicate the use of interrupting B operations that can follow any operation of the machine or INITIALISATION. An example of its use in an arbitrary operation $OP_i$ without the required NEXT annotation is:

Definition 16.

$$OP_i \triangleq (P_i \mid B_i)/ * FROM - ANY */;$$

In the above we state that after the execution of any operation of the machine, $OP_i$ will always be available for execution. The annotation gives rise to the proof obligation in definition 17 and definition 18:
Chapter 3. The FROM and QUERY Annotations

Definition 17 (FROM-ANY Proof Obligations for Operations).

\[ \forall op \in OPERATIONS \cdot (P_{op} \land I) \Rightarrow [B_{op}]P_i \]

Definition 18 (FROM-ANY Proof Obligations for Initialisations).

\[ [T]P_i \]

An operation annotated with a FROM-ANY annotation can be executed after any other operation of the machine. In definition 17 we investigate whether the interrupting operation is enabled after the execution of each of the other operations. If the invariant were strong enough it would already capture the notion of a state that holds after any operation has executed. Consider an invariant that insists the following: \( xx \in N \). If every operation and the initialisation of the machine makes \( xx \) even the invariant could have been strengthened to \( xx \in N \land xx \mod 2 = 0 \). Another approach to developing the PO for the FROM-ANY annotation is given in definition 19.

Definition 19 (Alternative FROM-ANY Proof Obligations for Operations).

\[ I \Rightarrow P_i \]

If the PO in definition 19 can not be discharged there are two choices: (1) strengthen the invariant to \( I \land P_i \), or if that is not permitted; (2) use the POs defined in 17 instead. The FROM-ANY annotation can be used in the same operation as the NEXT annotation.

3.1.2. The FROM-SET Annotation of a Consistent B Machine

The FROM-SET annotation extends the notion of interrupt by restricting the operations of a particular machine to a defined subset. The FROM-SET annotation is written \( \ast \) FROM-SET\( \{OP_j, ..., OP_k\} \ast \) (the short form of the annotation is \( \ast \, !_X \{OP_j, ..., OP_k\} \ast \)). This annotation is added to indicate the use of interrupting \( B \) operation that can follow any operations listed in the annotation. The set can not contain the initialisation.

A special annotation is introduced in the next section (FROM-SET-INIT) to capture the possibility of interrupting after a given set of operations or the initialisation. The reason that a special annotation is required is to do with the status of the initial action of the machine as compared to operation actions of the machine. The initialisation is not in the set of operations and therefore can not be invoked by the controller. The initialisation is not a language element of the controller nor the annotations. However, knowing if an operation can interrupt directly after the initialisation of the machine will have structural consequence.
for the construction of the controller. Because the initialisation action can not be added to, or left out of, the set of operations associated with the FROM-SET annotation in a straight forward way, a separate annotation is introduced: FROM-SET-INIT. The POs associated with the FROM-SET-INIT annotation prove that the operation it is associated with can follow any of the operations in the associated set or the initialisation.

An example of its use in an arbitrary operation $OP_i$ is:

**Definition 20.**

$$OP_i \equiv (P_i \mid B_i) \; / * \; FROM \; \; SET \{ OP_j, \ldots, OP_k \} \; * /;$$

In the above we state that after the execution of any of the operations in the given set $(OP_j, \ldots, OP_k)$, $OP_i$ will always be available to execute. The annotation gives rise to the following proof obligation in definition 21:

**Definition 21 (FROM-SET Proof Obligation for Operations).**

$$\forall op \in \{ OP_j, \ldots, OP_k \} \cdot (P_{op} \wedge I) \Rightarrow [B_{op}]P_i$$

The FROM-SET annotation can be used in the same operation as the NEXT annotation, but not with FROM-ANY.

### 3.1.3. The FROM-SET-INIT Annotation of a Consistent B Machine

This annotation has the initialisation implicitly added to the set of allowable operations to interrupt. The FROM-SET-INIT annotation is written FROM-SET-INIT $\{ OP_j, \ldots, OP_k \} \; * /$ (the short form of the annotation is $*/!^{x} \{ OP_j, \ldots, OP_k \} \; * /$). This annotation is added to indicate the use of interrupting B operations that can follow any operations listed in the annotation plus the initialisation. An example of its use in an arbitrary operation $OP_i$ is:

**Definition 22.**

$$OP_i \equiv (P_i \mid B_i) \; / * \; FROM \; \; SET \; \; INIT \{ OP_j, \ldots, OP_k \} \; * /;$$

In the above we state that after the execution of any of the operations in the given set or the initialisation, $OP_i$ will always be available to execute. The annotation gives rise to the following proof obligations in definition 23:
Chapter 3. The FROM and QUERY Annotations

Definition 23 (FROM-SET-INIT Proof Obligation).

\[(\forall \text{op} \in \{OP_j, ..., OP_k\} : (P_{\text{op}} \land I) \Rightarrow [B_{\text{op}}]P_i) \land \]

\[[T]P_i\]

The FROM-SET-INIT annotation can be used with the same operation as the NEXT annotation, but not with FROM-ANY.

3.1.4. Controller Syntax and Auxiliary Functions with FROM Annotations

The addition of the FROM annotation permits the extension of the controller syntax to include a global interrupt operator and conditional interrupts. The syntax of the controller, \(R\), extended with the defined interrupt is given in definition 24.

Definition 24 (Controller Syntax).

\[
R ::= a \rightarrow \Sigma 2 \mid R \circ R \mid R \triangle R \mid R \triangle_X R \mid R \triangle_X^I R \mid S(p)
\]

Recursive definitions with a process variable has the following syntax:

\[S(p) \triangleq R(p)\]

where \(R\) is defined as above.

The operators added to definition 9 are discussed. The global interrupt operator, \(\triangle\), permits the second controller fragment \(R\) to interrupt the former at any point including before the first action of the former has been performed. The defined interrupt operator \(\triangle_X\) permits the controller fragment to the right of the operator to interrupt the former controller fragment stated to the left of the interrupt, immediately after the execution of any operation in the set \(X\). The set \(X\) of \(\Delta_X\) may contain all the operators of the machine, but this does not make it operationally equivalent to the \(\Delta\) operator. The difference is maintained by properties of the \(\Delta\) operator, which permits the interrupting control fragment to cut in after the initialisation. The interrupting control fragment of the \(\Delta_X\) operator may not cut in directly after the initialisation, what is more the \(X\) set may not contain the initialisation. The \(\Delta_X^I\) operator overcomes this limitation. This operator has all the functionality of the \(\Delta_X\) operator plus the interrupting fragment of the \(\Delta_X^I\) operator may cut in directly after the initialisation. Effectively, the \(\Delta_X^I\) operator is like the \(\Delta_X\) operator with the initialisation included in \(X\). A possible use of an FROM-ANY annotation is exampled below.
\[ OP_i \equiv (P_i \mid B_i)/ \star \text{PROM} - \text{ANY} \star /\star \{OP_j\}\text{NEXT}\star /; \]

The meaning of consistency is extended by updating the *guarded* and *init* functions.

**Definition 25 (init on CSP Controller Process).**

\[
\begin{align*}
\text{init}(a \rightarrow R1) & = \{a\} \\
\text{init}(R1 \odot R2) & = \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(R1 \triangle R2) & = \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(R1 \triangle_X R2) & = \text{init}(R1) \\
\text{init}(R1 \triangle_X^i R2) & = \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(S(p)) & = \text{init}(R(p))
\end{align*}
\]

Note that the recursive definition of \( S(p) \) is given as follows: \( S(p) \equiv R(p) \)

*For example:* \( \text{init}(\text{LOOP}) = \{a\} \) if \( \text{LOOP} \equiv a \rightarrow b \rightarrow \text{LOOP} \)

**Definition 26 (guarded on CSP controller process with global interrupt).**

\[
\begin{align*}
\text{guarded}(a \rightarrow R1) & = \text{true} \\
\text{guarded}(R1 \odot R2) & = \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \triangle R2) & = \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \triangle_X R2) & = \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \triangle_X^i R2) & = \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(S(p)) & = \text{false}
\end{align*}
\]

eg. \( \text{guarded}(a \rightarrow P \triangle_X S(p)) = \text{true} \)

Only recursive definitions \( (S(p) \equiv R(p)) \) which have definitions \( R(p) \) that are guarded are considered, as in the previous chapter: \( \text{guarded}(R(p)) = \text{true} \) must be true

Hence \( S(p) \equiv R(p) \) defined if \( \text{guarded}(R(p)) = \text{true} \)

The listing function is extended to support the new annotation and introduce new functions to deal with the from annotations. The functions defined apply equally to the short form.
Chapter 3. The FROM and QUERY Annotations

Definition 27 (FROM-ANY Annotation Listing).
\[ \text{from - any}(O Pi) = \text{true} \quad \text{if} \quad O Pi \equiv (P_i | B_i)/* \text{ FROM - ANY */;} \]
\[ \text{from - any}(O Pi) = \text{false} \quad \text{otherwise} \]

Definition 28 (FROM-SET Annotation Listing).
\[ \text{from - set}(O Pi) = X \quad \text{if} \quad O Pi \equiv (P_i | B_i)/* \text{ FROM - SET X */;} \]
\[ \text{from - set}(O Pi) = \text{undefined} \quad \text{otherwise} \]

Definition 29 (FROM-SET-INIT Annotation Listing).
\[ \text{from - set - init}(O Pi) = X \quad \text{if} \quad O Pi \equiv (P_i | B_i)/* \text{ FROM - SET - INIT X */;} \]
\[ \text{from - set - init}(O Pi) = \text{undefined} \quad \text{otherwise} \]

The denotational definitions of interrupt given in definitions 30, 31 and 32 extends the CSP definition of interrupt to take account of the conditional nature of the definitions.

Definition 30 (Trace Definition of \( \Delta \)).
\[ \text{traces}(R1 \Delta R2) = \{ tr1 \uparrow tr2 \mid tr1 \in \text{traces}(R1) \land tr2 \in \text{traces}(R2) \} \]

Definition 31 (Trace Definition of \( \Delta_X \)).
\[ \text{traces}(R1 \Delta_X R2) = \{ tr1 \uparrow tr2 \mid tr1 \in \text{traces}(R1) \land \text{last}(tr1) \in X \land tr2 \in \text{traces}(R2) \} \cup \text{traces}(R1) \]

Definition 32 (Trace Definition of \( \Delta^X \)).
\[ \text{traces}(R1 \Delta^X R2) = \text{traces}(R1 \Delta_X R2) \cup \text{traces}(R2) \]

In definition 30 the case when R1 is not interrupted is captured by \( tr2 = () \). In definition 31 the case when R1 is not interrupted is captured by \( \text{traces}(R1) \). Hence \( \text{traces}(R1) \) is unionned with \( \{ tr1 \uparrow tr2 \mid tr1 \in \text{traces}(R1) \land \text{last}(tr1) \in X \land tr2 \in \text{traces}(R2) \} \). The function \( \text{last} \) returns the last element of the trace. If R1 is interrupted then the last action of R1 must be in the set X. In definition 32 the case when R1 is not interrupted is captured by definition 31. The interrupt may happen before R1 starts so \( \text{traces}(R2) \) is unionned in.

3.1.5. Annotation Consistency with Interrupts

M is machine-controller consistent with \( M_{CTRL} \) if \( \text{guarded}(M_{CTRL}) \) is true, M is machine-annotation consistent and \( M_{CTRL} \) is annotation consistent (initially-consistent and step-consistent - initial-consistency is considered for \( \text{init}(M_{CTRL}) \)). In short, an action of a
controller is consistent if the B machine $M$ enables it. The machine $M$ has an initialisation clause that has to execute initially to get the machine into a state that satisfies the invariant of the machine. The first action of $\text{M-CTRL}$ must be the initialisation of $M$.

**Definition 33 (Annotation Consistency with Interrupt(s)).**

1. $a \rightarrow R$
   
is annotation consistent with Machine $M$ if $a \rightarrow R$ is initially consistent and $a \rightarrow R$ is step-consistent with $M$
   
a $R$ is initially-consistent if $a \in \text{next}(\text{INITIALISATION})$

2. $R1 \sqcup R2$
   
is annotation consistent with Machine $M$ if $R1$ and $R2$ are annotation consistent with $M$.

3. $R1 \triangle R2$
   
is annotation consistent with Machine $M$ if $R1$ is annotation consistent and $\forall \text{op} \in \text{init}(R2) \cdot \text{from - any(op)} = \text{true}$ and $R2$ is step-consistent with $M$

4. $R1 \triangle X R2$
   
is annotation consistent with Machine $M$ if $R1$ annotation consistent with $M$ and $\forall \text{op} \in \text{init}(R2) \cdot X \subseteq \text{from - set(op)}$, and $R2$ is step-consistent with $M$

5. $R1 \triangle X R2$
   
is annotation consistent with Machine $M$ if $R1$ is annotation consistent with $M$ and $\forall \text{op} \in \text{init}(R2) \cdot X \subseteq \text{from_set_i(op)}$, and $R2$ is step-consistent with $M$

6. $S(p)$
   
is annotation consistent with Machine $M$

A family of recursive definitions $S(p) \equiv R(p)$ is annotation consistent with $M$'s annotations if each $R(p)$ is annotation consistent with $M$'s annotations.
When an interrupt occurs there is a break in the normal looping control behaviour. Normal control flow behaviour is captured with NEXT annotations in the B machine. Interrupt behaviour is captured in the machine with FROM annotations. The FROM annotation in machines relates to interrupts in the control language, whereas NEXT annotations relate to pre-fixes and external choices in the control language. Both normal and interrupt behaviour are justified with POs.

Annotation consistency of $R_1 \triangleleft R_2$ is developed with respect to three types of execution. There are two extreme behaviours. If $R_2$ never interrupts $R_1$ then $R_1$ must be annotation consistent. At the other extreme $R_2$ interrupts before $R_1$ starts hence $R_2$ must be annotation consistent. However, this must be established by reference to the POs ($\forall op \in \text{OPERATION} \cdot P_{op} \land I \rightarrow [B_{op}]P_i$ (all operations are enabled by the initialisation action of the machine)), which follow from $\forall op \in \text{init}(R_2) \cdot \text{from} - \text{any}(op) = \text{true}$ (all the initial actions in $R_2$ are FROM-ANY annotated). Annotation consistency can not be established recursively by calling on the definition for Annotation Consistency, definition 33, as was the case for $R_1$ which may establishes initial consistency with the NEXT annotations in clause 1. An interrupt control fragment like $R_2$ would not be part of the normal flow of the machine so would not be supported with NEXT annotations. In the case of $R_2$ if the PO is discharged then it has been shown that it can follow an initial action of a machine. All that is left to do is establish it continues on correctly, hence $R_2$ must be step-consistent. In summary $R_1 \triangleleft R_2$ is annotation consistent if $R_1$ is annotation consistent and $R_2$ is step-consistent. There are different considerations to be taken into account when considering $R_1 \triangleleft R_2$. $R_1$ must engage in some events before $R_2$ can interrupt. Not allowing $R_2$ to interrupt the initialisation means that $R_2$ only need be step-consistent. $R_1 \triangleleft R_2$ follows the same pattern as $R_1 \triangleleft R_2$.

Definition 34 (Step-Consistency with Interrupts).

The step-consistency of the controller fragment $R$ is defined as follows:

1. $b \rightarrow R_1$
   
   is step-consistent with Machine $M$ if $\text{init}(R_1) \subseteq \text{next}(b)$ and $R_1$ is step-consistent with $M$.

2. $R_1 \square R_2$
   
   is step-consistent with Machine $M$ if $R_1$ and $R_2$ are step-consistent with $M$.

3. $R_1 \triangleleft R_2$
   
   is step-consistent with Machine $M$ if $R_1$ is step-consistent with $M$ and if $\forall op \in \text{init}(R_2) \cdot \text{from} - \text{any}(op) = \text{true}$ and $R_2$ is step-consistent.
4. \( R_1 \triangle X R_2 \)

is step-consistent with Machine \( M \) if \( R_1 \) is step-consistent with \( M \) and
\( \forall op \in \text{init}(R_2) \cdot X \subseteq \text{from} - \text{set}(op) \) and \( R_2 \) is step-consistent with \( M \)

5. \( R_1 \triangle_x^i R_2 \)

is step-consistent with Machine \( M \) if \( R_1 \) is step-consistent and \( R_2 \) is step-consistent with \( M \) and \( \forall op \in \text{init}(R_2) \cdot X \subseteq \text{from} - \text{set} - \text{init}(op) X \)

6. \( S(p) \)

is step-consistent with Machine \( M \)

A family of recursive definitions \( S \equiv R \) is step-consistent with \( M \)'s annotations if each \( R \) is step-consistent with \( M \)'s annotations.

The treatment of the interrupt operators in terms of step-consistency is similar to the treatment given in the case of annotation-consistency. The step-consistency of the interrupt operators is established by considering the two possible behaviours that the operator might give rise to: the option to interrupt may or may not be taken where it is offered. The annotations in the B machine are shown, by discharging all the associated proof obligations, to support either outcome. The definition of the consistency for the from annotations includes a check to examine if the from annotation is present. In the case of the global (FROM-ANY) interrupt as the interrupt can occur after any operation we simple test both possible paths: the interrupt is taken (consider the interrupt path \( R_2 \)) or the interrupt is not taken (consider the normal path \( R_1 \)).

3.1.6. Proving Termination of Controlled Machines with FROM Annotations

The proofs in this section establish that it is enough to demonstrate annotation-consistency (given machine-annotation consistency) to show that the every step of the machine-controller pair terminates, which implies non-divergence: every operation is called within its precondition. The proofs in this section extend section 2.7 by introducing new proofs for machines with FROM annotations and controllers with interrupt operations.

We repeat the proof of theorem 1 in light of the new operators and new step-consistency definitions. We show that only good traces are produced by \( M_{CTRL} \). Hence \( M \parallel M_{CTRL} \) is divergence free. We reason over the lengths of the traces in the proof.

The theorem is restated as follows:

Theorem 2 (Machine-Annotation Consistent Machines and Annotation Consistent Controller Combinations are Divergence Free). If a from annotated machine \( M \)
is machine-annotation consistent and its controller $M_{CTRL}$ is annotation consistent then $M \parallel M_{CTRL}$ is divergence free.

We prove by induction over length of traces that traces of $M_{CTRL}$ are non-divergent.

The proof that arbitrary length traces are non-divergent:

Case (A) ()

Entirely similar to the proof of theorem 1, page 30, Case A.

Case (B) $(a) \in \text{traces}(M_{CTRL})$

1 $(a) \in \text{traces}(M_{CTRL})$ and $M_{CTRL}$ annotation consistent

initial assumption

2 $a \in \text{next}(\text{INITIALISATION}) \lor$

from-any($a$)$\lor$

from-set-init($a$) $\neq \{}$

1, Lemma 4 below

3 $[T]P_a$

by def. of PO of annotations in all cases

4 $[T]I$

by of machine-consistency def. 4

5 $[T](P_a \land I)$

by 3 and 4

Entirely similar to the proof of theorem 1, page 30, Case B lines 6 to end
3.1. The FROM annotations

Case (C) \( tr \ominus (a, b) \in \text{traces}(M_{\text{CRTL}}) \) where \( tr \in \text{traces}(M_{\text{CRTL}}) \)

1. \( \langle tr \ominus a \rangle \) is good by inductive hypothesis
2. \( b \in \text{next}(a) \lor \)
   from \(-\) \( \text{any}(b) \lor \)
   \( a \in \text{from} \(-\) \( \text{set}(b) \lor \)
   \( a \in \text{from} \(-\) \( \text{set-init}(b) \)
3. \( P_a \land I \Rightarrow [B_a]P_b \)
   def. of annotations in all case

Entirely similar to the proof of theorem 1, page 30, Case (C) lines 5 to end.

Singleton Traces are Enabled by Initialisation Annotations

Lemma 4 (Singleton Traces are Enabled by Initialisation Annotations). The actions of singleton traces from the annotated consistent controller \( R \) are enabled by initialisation annotation or from annotation in the associated machine-annotation consistent machine:

\[
\begin{align*}
\langle \text{machine-controller consistency} \land \\
(a) \in \text{traces}(R) \Rightarrow (a \in \text{next}(\text{INITIALISATION}) \lor \\
\text{from} \(-\) \text{any}(a) \lor \text{from} \(-\) \text{set-init}(a) \neq \{\})
\end{align*}
\]

Proof of Lemma 4 by structured induction over cases of controller fragments:

case 1 \( (c \rightarrow R) \)

1. \( (a) \in \text{traces}(c \rightarrow R) \)
   initial assumption
2. \( (c \rightarrow R) \) annotation consistent
   initial assumption
3. \( c \in \text{next}(\text{INITIALISATION}) \)
   2, def. of annotation consistency
4. \( a = c \)
   1 and def. of traces
5. \( a \in \text{next}(\text{INITIALISATION}) \)
   3,4
6. \( a \in \text{next}(\text{INITIALISATION}) \lor \)
   5 and \lor
   \text{from} \(-\) \text{set-init}(a) \neq \{\} \lor
   \text{from} \(-\) \text{any}(a)

QED

The annotation consistency of \( (c \rightarrow R) \) infers that \( c \in \text{next}(\text{INITIALISATION}) \). Although the initial action of a controller can be provided by an interrupting control fragment introduced with a FROM-ANY or FROM-SET-INIT annotation, the interrupt is not introduced in the control fragment being considered. Hence line 3 of the proof uses only the NEXT annotation.
Chapter 3. The PROM and QUERY Annotations

Case 2 \( R_1 \Box R_2 \)

1. \( (a) \in \text{traces}((R_1)\Box(R_2)) \) initial assumption
2. \( (R_1)\Box(R_2) \) annotation consistent initial assumption
3. \( (a) \in \text{traces}(R_1) \) or \( (a) \in \text{traces}(R_2) \) by 1 and def. of CSP traces

4. sub-case: \( (a) \in \text{traces}(R_1) \)

5. \( R_1 \) annotation consistent by 2 and 
   def. annotation consistency 33
6. \( a \in \text{next}(\text{INITIALISATION}) \lor 
   \text{from} - \text{set} - \text{init}(a) \neq \{\} \lor 
   \text{from} - \text{any}(a) \)
   sub-case 4,5, inductive hypothesis on \( R_1 \)

7. sub-case: \( (a) \in \text{traces}(R_2) \)

8. \( R_2 \) annotation consistent by 2 and 
   def. annotation consistency 33
9. \( (a) \in \text{next}(\text{INITIALISATION}) \lor 
   \text{from} - \text{set} - \text{init}(a) \neq \{\} \lor 
   \text{from} - \text{any}(a) \)
   sub-case 7,8, inductive hypothesis on \( R_2 \)

QED

Case 2 \((R_1 \Box R_2)\) considers the two controller fragments \( R_1 \) and \( R_2 \) separately. Line 6 and 9 use the inductive principle that allows the assumption of the conclusion of the lemma to apply to the sub-cases. The work of the proof is focused only on showing that the combination of \( R_1 \) and \( R_2 \) maintains the inductive assumptions, which is trivially true as a consequence of the or connective introduced in line 3. Cases 4, 5 and 6 use the inductive principle in a similar way.
3.1. The FROM annotations

case 3 $R1 \triangle R2$

1. $(a) \in \text{traces}(R1 \triangle R2)$ initial assumption
2. $(R1 \triangle R2)$ annotation consistent initial assumption
3. $(a) \in \text{traces}(R1)$ or $(a) \in \text{traces}(R2)$ by 1 and def. of CSP traces

4. sub-case: $(a) \in \text{traces}(R1)$

5. $R1$ annotation consistent 2, def. annotation consistency
6. $a \in \text{next(INITIALISATION)} \lor \text{from} - \text{any}(a) \lor \text{from} - \text{set} - \text{any}(a) \neq \{\}$

7. sub-case: $(a) \in \text{traces}(R2)$

8. $\forall \text{op} \in \text{init}(R2)$

9. $\text{from} - \text{any}(\text{op}) = \text{true}$

10. $\text{from} - \text{any}(c)$

11. $a = c$

12. $a \in \text{next(INITIALISATION)} \lor \text{from} - \text{any}(a) \lor \text{from} - \text{set} - \text{any}(a) \neq \{\}$

QED

case 4 $R1 \triangle X R2$

1. $(a) \in \text{traces}(R1 \triangle X R2)$ initial assumption
2. $(R1 \triangle X R2)$ annotation consistent initial assumption
3. $(a) \in \text{traces}(R1)$ by 1 and def. of CSP traces

4. $R1$ annotation consistent by 2, and def. annotation consistency

5. $a \in \text{next(INITIALISATION)} \lor \text{from} - \text{any}(a) \lor \text{from} - \text{set} - \text{any}(a) \neq \{\}$

QED
Chapter 3. The FROM and QUERY Annotations

case 5 \[ R_1 \Delta X R_2 \]

1 \[ \langle a \rangle \in \text{traces}(R_1 \Delta X R_2) \] initial assumption
2 \[ \text{\(R_1 \Delta X R_2\) annotation consistent} \] initial assumption
3 \[ \langle a \rangle \in \text{traces}(R_1) \text{ or } \langle a \rangle \in \text{traces}(R_2) \] def. of CSP traces

4 sub-case: \[ \langle a \rangle \in \text{traces}(R_1) \]

5 \[ R_1 \text{ annotation consistent} \] by 2 and def. of
6 \[ a \in \text{next(INITIALISATION)} \vee \]
   \[ \text{from} - \text{any}(a) \vee \]
   \[ \text{from} - \text{set} - \text{any}(a) \neq \{} \] annotation consistency 33

sub-case 4,5, inductive hypothesis on \(R_1\)

7 sub-case: \[ \langle a \rangle \in \text{traces}(R_2) \]
8 \[ \forall \text{op} \in \text{init}(R_2) \cdot \text{from} - \text{set} - \text{any}(\text{op}) = \text{true} \] by 2
9 \[ \text{from} - \text{set} - \text{any}(c) \neq \{} \] by 8, \(c \in \text{init}(R_2)\)
10 \[ a = c \] by def. traces
11 \[ \text{from} - \text{set} - \text{any}(a) \neq \{} \] by 10, 9
12 \[ a \in \text{next(INITIALISATION)} \wedge \]
   \[ \text{from} - \text{any}(a) \wedge \]
   \[ \text{from} - \text{set} - \text{any}(a) \neq \{} \] by \(\vee\) and by 11

QED

case 6 \[ S(p) \]

1 \[ S(p) \text{ annotation consistent} \] initial assumption
2 \[ \langle a \rangle \in \text{traces}(S(p)) \] initial assumption
3 \[ R(p) \text{ annotation consistent} \] 1, def. annotation consistency
4 \[ \langle a \rangle \in \text{traces}(R(p)) \] inductive hyp. on \(R(p)\)
5 \[ \langle a \rangle \in \text{next(INITIALISATION)} \vee \]
   \[ \text{from} - \text{any}(a) \vee \]
   \[ \text{from} - \text{set} - \text{init}(a) \neq \{} \] def. annotation consistency

QED

54 July 6, 2008
Arbitrary Traces are Enabled by Annotations

Lemma 5 (Arbitrary length traces of annotation consistent controllers are enabled by annotations). The actions of arbitrary length traces of annotation consistent interruptible controller fragments are enabled by operation annotations in the associated machine-annotation consistent machine:

\[(R \text{ annotation consistent } \land tr \uparrow (a) \land \langle b \rangle \in \text{traces}(R)) \Rightarrow b \in \text{next}(a) \lor
\]
\[\text{from } - \text{any}(b) \lor
\]
\[a \in \text{from } - \text{set}(b) \lor
\]
\[a \in \text{from } - \text{set } - \text{init}(b)
\]

Proof of Lemma 5 by Cases of controllers:

case A \quad c \rightarrow R

1 \quad c \rightarrow R \text{ annotation consistent with } M \quad \text{initial assumption.}

sub-case: \quad tr = ()

2 \quad tr \uparrow (a) \land \langle b \rangle \in \text{traces}(c \rightarrow R) \quad \text{initial assumption}

3 \quad \langle a \rangle \land \langle b \rangle \in \text{traces}(c \rightarrow R)

4 \quad a = c \land \langle b \rangle \in \text{traces}(R) \quad \text{def. of traces}

5 \quad b \in \text{init}(R)

6 \quad b \in \text{next}(c)

7 \quad b \in \text{next}(a)

8 \quad b \in \text{next}(a) \lor
\text{from } - \text{any}(b) \lor
\text{from } - \text{set}(b) \lor
\text{from } - \text{set } - \text{init}(b)

\[QED\]

sub-case: \quad tr \neq ()

9 \quad tr \uparrow (a) \land \langle b \rangle \in \text{traces}(c \rightarrow R) \quad \text{initial assumption}

10 \quad \text{let } tr = (e) \land tr'

11 \quad \langle e \rangle \land tr' \uparrow (a) \land \langle b \rangle \in \text{traces}(c \rightarrow R) \quad \text{initial assumption}

12 \quad e = c \land tr' \uparrow (a) \land \langle b \rangle \in \text{traces}(R) \quad 11, \text{def. traces}

13 \quad tr' \uparrow (a) \land \langle b \rangle \in \text{traces}(R)

14 \quad R \text{ annotation consistent with } M \quad \text{by inductive hypothesis}

15 \quad b \in \text{next}(a) \lor
\text{from } - \text{any}(b) \lor
\text{from } - \text{set}(b) \lor
\text{from } - \text{set } - \text{init}(b)

\[QED\]

July 6, 2008
Chapter 3. The FROM and QUERY Annotations

The proofs have exactly the same form as the proofs in lemma 2. They have been reproduced because they point to different lemmas. When considering the first two elements of the traces of the fragment $c \rightarrow R$, in the first sub-case, it must be remembered that the event $c$ must occur first. Here $c$ is introduced using the prefixed control element, which can only be introduced in turn, using the NEXT annotation. It can not be assumed that from annotations relates $a$ and $b$ as there is no explicit mention of a interrupt operator in the control fragment.

**case B \( R1 \sqcap R2 \)**

1. \( tr \uparrow (a) \uparrow (b) \in traces(R1\sqcap R2) \) initial assumption

2. \( R1 \sqcap R2 \) annotation consistent with \( M \) initial assumption

3. \( R1 \) annotation consistent with \( M \) 2, def. consistency 34

4. \( R2 \) annotation consistent with \( M \) 2, def. consistency 34

5. \( tr \uparrow (a) \uparrow (b) \in traces(R1) \) or
   \( tr \uparrow (a) \uparrow (b) \in traces(R2) \) 1, def. trace

**sub-case:** \( tr \uparrow (a) \uparrow (b) \in traces(R1) \)

6. \( b \in next(a) \) \( \lor \)
   \( from - any(b) \) \( \lor \)
   \( a \in from - set(b) \)
   \( a \in from - set - init(b) \)

**sub-case:** \( tr \uparrow (a) \uparrow (b) \in traces(R2) \)

7. \( b \in next(a) \) \( \lor \)
   \( from - any(b) \) \( \lor \)
   \( a \in from - set(b) \)
   \( a \in from - set - init(b) \)

**QED**
3.1. The FROM annotations

case C \( R1 \triangle R2 \)

1. \( tr \wedge (a) \wedge (b) \in \text{traces}(R1 \triangle R2) \)  
   initial assumption

2. \( R1 \triangle R2 \) annotation consistent with \( M \)  
   initial assumption

3. \( R1 \) annotation consistent with \( M \)  
   2, def. consistency 34

4. \( R2 \) annotation consistent with \( M \)  
   2, def. consistency 34

5. \( tr \wedge (a) \wedge (b) = tr_1 \wedge tr_2 \)  
   by traces def. 30

   where \( tr_1 \in \text{traces}(R1) \wedge \) \( tr_2 \in \text{traces}(R2) \)

sub-case: \( tr_1 = tr \wedge (a) \wedge (b) \)

6. \( (tr_1 = tr \wedge (a) \wedge (b) \wedge tr_2 = \emptyset) \vee \)  
   by 5

   \( (tr_1 = tr \wedge (a) \wedge tr_2 = (b)) \vee \)

   \( (tr_2 = tr_3 \wedge (a) \wedge (b) \wedge \) \( tr_3 = (\emptyset)) \)

7. \( tr_2 \wedge (a) \wedge (b) \in \text{traces}(R1) \)  
   by 5 and 6

8. \( b \in \text{next}(a) \vee \)  
   by 5 and 6

   \( \text{from} - \text{any}(b) \vee \)

   \( a \in \text{from} - \text{set}(b) \vee \)

   \( a \in \text{from} - \text{set} - \text{init}(b) \)

   3,7, inductive hypothesis on R1

sub-case: \( tr_1 = tr \wedge (a) \wedge tr_3 = (b) \)

9. \( tr \wedge (a) \in \text{traces}(R1) \wedge (b) \in \text{traces}(R2) \)  
   by 5 and 6

10. \( b \in \text{init}(R2) \)  
    def. of init 25

11. \( \text{from} - \text{any}(b) \)  
    2, 4 and def. of

    annotation consistency 34

sub-case: \( tr_3 = tr \wedge (a) \wedge (b) \)

by 11 and def. of \( \vee \)

12. \( b \in \text{next}(a) \vee \)  
    by 5 and 6

   \( \text{from} - \text{any}(b) \vee \)

   \( a \in \text{from} - \text{set}(b) \vee \)

   \( a \in \text{from} - \text{set} - \text{init}(b) \)

by 4,13 and

inductive hypothesis on R2

13. \( tr_3 \wedge (a) \wedge (b) \in \text{traces}(R2) \)  
    by 5 and 6

14. \( b \in \text{next}(a) \vee \)  
    by 4,13 and

   \( \text{from} - \text{any}(b) \vee \)

   \( a \in \text{from} - \text{set}(b) \vee \)

   \( a \in \text{from} - \text{set} - \text{init}(b) \)

   inductive hypothesis on R2

\[ \text{QED} \]
Chapter 3. The FROM and QUERY Annotations

case D \((R1 \Delta_X R2)\)

1 \(tr \uparrow (a) \uparrow (b) \in \text{traces}(R1 \Delta_X R2)\) initial assumption
2 \(R1 \Delta_X R2\) annotation consistent with \(M\) initial assumption
3 \(R1\) annotation consistent with \(M\) 2, def. consistency 34
4 \(R2\) annotation consistent with \(M\) 2, def. consistency 34
5 \(tr \uparrow (a) \uparrow (b) = tr_1 \uparrow tr_2\) define of traces model 31

\[\begin{align*}
&\left((tr_1 = tr \uparrow (a) \uparrow (b) \wedge tr_2 = (\cdot)) \vee \\
&(tr_1 = tr \uparrow (a) \wedge tr_2 = (b) \wedge a \in X) \vee \\
&(tr_2 = tr_3 \uparrow (a) \uparrow (b) \wedge \text{last}(tr_1) \in X)) \wedge \\
&\text{tr}_1 \text{ is a proper prefix of } tr
\end{align*}\]

\(\text{sub-case: } tr_1 = tr \uparrow (a) \uparrow (b) \wedge tr_2 = (\cdot)\) from 5

6 \(b \in \text{next}(a) \vee \) by 5 and sub-case
7 \(a \in \text{from} - \text{set}(b) \vee \) by 7 and inductive hypothesis
8 \(a \in \text{from} - \text{set} - \text{init}(b)\)

\(\text{sub-case: } tr_1 = tr \uparrow (a) \wedge tr_2 = (b) \wedge \text{last}(tr_1) \in X\)

9 \(tr_3 \uparrow (a) \in \text{traces}(R1) \wedge (b) \in \text{traces}(R2)\) by 5 and sub-case
10 \(a \in \text{from} - \text{set}(b)\) 4,9, and def. consistency 34
11 \(b \in \text{next}(a) \vee \) by 10, and def. of \(\vee\)
12 \(a \in \text{from} - \text{set}(b) \vee \)
13 \(a \in \text{from} - \text{set} - \text{init}(b)\)

\(\text{sub-case: } tr_2 = tr_3 \uparrow (a) \uparrow (b) \wedge \text{last}(tr_1) \in X\)

\(\text{sub-case: } tr_3 \uparrow (a) \uparrow (b) \in \text{traces}(R2)\) by 5 and sub-case
\(\text{by } 4, 12 \text{ and inductive hypothesis}\)

\(\text{QED}\)
3.1. The FROM annotations

\textbf{case E} \quad (R1 \triangle_X R2)

1. \quad tr \cap \{a\} \cap \{b\} \in \text{traces}(R1 \triangle_X R2) \quad \text{initial assumption}
2. \quad R1 \triangle_X R2 \text{ annotation consistent with } M \quad \text{initial assumption}
3. \quad R1 \text{ annotation consistent with } M \quad \text{2, def. consistency 34}
4. \quad R2 \text{ annotation consistent with } M \quad \text{2, def. consistency 34}
5. \quad tr \cap \{a\} \cap \{b\} = tr_1 \cap tr_2 \quad \text{definition of traces model 32}
\quad \text{where } tr_1 \in \text{traces}(R1) \land
\quad tr_2 \in \text{traces}(R2) \land
\quad tr_2 \neq \{} \Rightarrow \text{last}(tr_1) \in X
\quad \text{by 5 the semantics of } \triangle_X \text{ gives}

\text{sub-case 1: } \quad tr_1 = tr \cap \{a\} \cap \{b\}

6. \quad (tr_1 = tr \cap \{a\} \cap \{b\} \land tr_2 = \{\}) \lor
\quad (tr_2 = tr \cap \{a\} \cap \{b\} \land tr_1 = \{\}) \lor
\quad (tr_1 = tr \cap \{a\} \land tr_2 = \{b\}) \land
\quad \text{last}(tr_1) \in X \lor
\quad (tr_2 = tr_3 \cap \{a\} \cap \{b\} \land \text{last}(tr_1) \in X)

7. \quad tr \cap \{a\} \cap \{b\} \in \text{traces}(R1) \quad \text{by 5 and sub-case}
8. \quad b \in \text{next}(a) \lor \text{from - any}(b) \lor
\quad a \in \text{from - set}(b) \lor
\quad a \in \text{from - set - init}(b) \quad \text{7, inductive hypothesis on R1}

\text{sub-case 2: } \quad tr_2 = tr \cap \{a\} \cap \{b\}

9. \quad tr \cap \{a\} \cap \{b\} \in \text{traces}(R2) \quad \text{by 5 and sub-case}
10. \quad b \in \text{next}(a) \lor \text{from - any}(b) \lor
\quad a \in \text{from - set}(b) \lor
\quad a \in \text{from - set - init}(b) \quad \text{9, inductive hypothesis on R2}

\text{sub-case 3: } \quad tr_1 = tr \cap \{a\} \land tr_2 = \{b\} \land \text{last}(tr_1) \in X

11. \quad tr \cap \{a\} \in \text{traces}(R1) \land \{b\} \in \text{traces}(R2) \quad \text{by 5 and sub-case}
12. \quad a \in \text{from - set - init}(b) \quad \text{by 11 and 2}
13. \quad b \in \text{next}(a) \lor \text{from - any}(b) \lor
\quad a \in \text{from - set}(b) \lor
\quad a \in \text{from - set - init}(b) \quad \text{12 and def. of or}

\text{sub-case 4: } \quad tr_2 = tr_3 \cap \{a\} \cap \{b\} \land \text{last}(tr_1) \in X

14. \quad tr_3 \cap \{a\} \cap \{b\} \in \text{traces}(R2) \quad \text{by 5 and sub-case}
15. \quad b \in \text{next}(a) \lor
\quad \text{from - any}(b) \lor a \in \text{from - set}(b) \lor
\quad a \in \text{from - set - init}(b) \quad \text{14, inductive hypothesis on R2}

\textbf{QED}
Chapter 3. The FROM and QUERY Annotations

case G \( S(p) \)

1 \( tr \cap (a) \cap (b) \in \text{traces}(S(p)) \) initial assumption
2 \( S(p) \) annotation consistent with \( M \) initial assumption
3 \( tr \cap (a) \cap (b) \in \text{traces}(R(p)) \) \( S(p) \equiv R(p) \)
4 \( R(p) \) annotation consistent with \( M \) 2, 3 and def. consistency 34
5 \( b \in \text{next}(a) \) 3, 4, inductive hypothesis on \( R(p) \)

QED

The Initial Action of a Consistent Controller

Lemma 6 (Elements of annotation consistent singleton traces are elements of \( \text{init}(R) \)). An element of the singleton trace of an annotation consistent controller fragment \( R \) is contained in the \( \text{init} \) of \( R \):

\[ R \text{ annotation consistent and } ((b) \in \text{traces}(R)) \implies b \in \text{init}(R) \]

Proof of Lemma 6 by cases of controllers:

case A \((c \rightarrow R)\)

1 \( (b) \in \text{traces}(c \rightarrow R) \) initial assumption
2 \( b = c \) 1, def. of CSP traces
3 \( b \in \text{init}(c \rightarrow R) \) definition of \( \text{init} \) 25

QED

case B \((R1 \Box R2)\)

1 \( (b) \in \text{traces}(R1 \Box R2) \) initial assumption
2 \( (b) \in \text{traces}(R1) \) or \( (b) \in \text{traces}(R2) \) 1, def. of CSP traces
3 sub-case: \( (b) \in \text{traces}(R1) \)
4 \( b \in \text{init}(R1) \) 3 sub-case and inductive hypothesis on R1
5 sub-case: \( (b) \in \text{traces}(R2) \)
6 \( b \in \text{init}(R2) \) 5 sub-case and inductive hypothesis on R2

QED
3.1. The FROM annotations

case C \((R1 \triangle R2)\)
1. \((b) \in \text{traces}(R1 \triangle R2)\) initial assumption
2. \((b) \in \text{traces}(R1)\) or \((b) \in \text{traces}(R2)\) 1, def. of CSP traces

sub-case: \((b) \in \text{traces}(R1)\)
3. \(b \in \text{init}(R1)\) 2, inductive hypothesis on \(R1\)

sub-case: \((b) \in \text{traces}(R2)\)
4. \(b \in \text{init}(R2)\) 2, inductive hypothesis on \(R2\)
hence in both cases \(b \in \text{init}(R1 \triangle R2)\)

QED

case D \((R1 \triangle L_R \triangle R2)\)
1. \((b) \in \text{traces}(R1 \triangle L_R \triangle R2)\) initial assumption
2. \((b) \in \text{traces}(R1)\) def. of CSP traces

3. \(b \in \text{init}(R1)\) 2, inductive hypothesis

QED

case E \((R1 \triangle L_R \triangle R2)\)
1. \((b) \in \text{traces}(R1 \triangle L_R \triangle R2)\) initial assumption
2. \((b) \in \text{traces}(R1)\) or \((b) \in \text{traces}(R2)\) 1, def. CSP traces

sub-case: \((b) \in \text{traces}(R1)\)
3. \(b \in \text{init}(R1)\) 2, inductive hypothesis on \(R1\)

sub-case: \((b) \in \text{traces}(R2)\)
4. \(b \in \text{init}(R2)\) 2, inductive hypothesis on \(R2\)
hence in both cases \(b \in \text{init}(R1 \triangle L_R \triangle R2)\)

QED
case F \((S(p))\)

1 \((b) \in \text{traces}(S(p))\) initial assumption
2 \((b) \in \text{traces}(R(p))\) \(S(p) \equiv R(p)\)
3 \(b \in \text{init}(R(p))\) 2, inductive hypothesis on \(R(p)\)
4 \(b \in \text{init}(S(p))\) 2 and \(S(p) \equiv R(p)\)

\[\text{QED}\]

3.1.7. Picturing the FROM-ANY Annotation

The generic FROM-ANY annotation given in definition 16, can be pictured as in Figure 3.1. Any transition of any of the operations that range from \(op_a\) to \(op_z\) in the machine into the state may occur. The exiting operation \(op_j\) may follow any operation of the machine even the initialisation of the machine.

![Figure 3.1: Picturing the FROM-ANY Annotation of \(op_j\)](image)

3.1.8. Picturing the FROM-SET Annotation

The generic FROM-SET annotation given in definition 20, on its own, is pictured as in Figure 3.2. Any transition in the \(X\) set \({op_i, ..., op_k}\) may lead to a state that can execute the current operation \(op_j\). Let the operations in the machine range from \(op_a\) to \(op_z\).

3.1.9. Picturing the FROM-SET-INIT Annotation

The generic FROM-SET-INIT annotation 22, on its own, is pictured as in Figure 3.3. Any transition in the \(X\) set \({op_i, ..., op_k}\) or transition from initialisation may lead to a state.

July 8, 2008
3.1. The FROM annotations

Figure 3.2.: Picturing the FROM-SET\{op_1, ..., op_k\} Annotation of op_j

that can execute the current operation op_j. Let the operations in the machine range from op_a to op_z.

Figure 3.3.: Picturing the FROM-SET-INIT Annotation of op_j

3.1.10. Examples Utilising FROM-ANY

The requirement for the traffic control system for the main street of the walled City of Carcassonne is specified in section 2.9. The Aller_pedestrian operation of the B machine depicted in Figure 3.4 and Figure 3.5 stops all traffic and holds all lights on red in both directions to allow pedestrians to walk on the road. The other operations cycle interleaving access to the road for the car users travelling up from the moat or down from the square. The B machine given in Figure 3.4 and Figure 3.5 includes the addition of a set of lights to control the flow of pedestrians when the lights controlling the vehicle traffic are halted. In the
MACHINE Global_Traffic
SETS COMMAND = {Arret, Aller}
VARIABLES Moat, Square, Pedestrian
INVARIANT Moat : COMMAND \& Square : COMMAND \&
    Pedestrian : COMMAND \& (Moat = Arret \lor Square = Arret) \&
    (Pedestrian = Aller \implies Moat = Arret \land Square = Arret)
INITIALISATION Moat, Square, Pedestrian := Arret, Arret, Arret
/* {Aller_Moat} */

Figure 3.4.: Global Interrupting Traffic Machine - Part 1

machine the pedestrian lights are added as another variable of type COMMAND. The Moat
and Square traffic lights are given their own Arret operation Arret_Moat and Arret_Square.
The Aller_Pedestrian operation is added. It can be (Aller) when both the Moat and Square
traffic lights are set to Arret. The machine's consistent controller is given in Figure 3.6. All
the operations in the example have NEXT annotations added to them. The abbreviation for
the NEXT annotation is used ({{...}}). The Arret_All operation has a FROM ANY annotation
as well as a NEXT annotation. The abbreviation for the FROM ANY is used ({{*}}).

3.1.11. Examples Utilising FROM-SET

Like the FROM-ANY annotation, the FROM-SET annotation will be used more sparingly in B
machines. We redo the Traffic machine to interrupt only after halt operations.

The B machine depicted in Figure 3.7, on page 66 priorities traffic travelling out of the city
by interrupting the traffic travelling into the city from the moat when ever necessary. The
machine consistent controller is given in Figure 3.8 on page 67. An example trace is given
below:

{ Arret_All, Aller_Moat, Arret_All, Aller_Square, Arret_All, Aller_Moat, Arret_All, ... }

3.2. The QUERY Annotations

Query operations return values to the environment to allow the environment to understand
the current state of the system. Some actions may occur at any time. Query operations
are an example of such actions. This creates a new challenge for the definition and proof of
consistency. Any consistency predicate formulated before the occurrence of a query must be
maintained during the execution of the query. The occurrence of a query operation does not
change the state of the system. Therefore a consistent controller cannot be made inconsistent
by adding query operations to it. A new annotation will be introduced in this section: the
QUERY. An operation annotated with QUERY occurs when required by the environment.
The query operations must have no effect on the state of the machine which they are in.
We provide a theorem that demonstrates that because they cannot change the state of the

3.2. The QUERY Annotations

OPERATIONS

\[
\begin{align*}
\text{Aller}_\text{Moat} & \triangleq \text{PRE Square} = \text{Arret} \land \text{Pedestrian} = \text{Arret} & \text{THEN} & \text{Moat} := \text{Aller} \text{ END} & / \ast \{\text{Arret}_\text{Moat}\}! \ast/; \\
\text{Arret}_\text{Moat} & \triangleq \text{PRE Moat} = \text{Aller} \land \text{Pedestrian} = \text{Arret} & \text{THEN} & \text{Moat} := \text{Arret} \text{ END} & / \ast \{\text{Aller}_\text{Square}\}! \ast/; \\
\text{Arret}_\text{Square} & \triangleq \text{PRE Moat} = \text{Arret} \land \text{Pedestrian} = \text{Arret} & \text{THEN} & \text{Square} := \text{Aller} \text{ END} & / \ast \{\text{Arret}_\text{Square}\}! \ast/; \\
\text{Arret}_\text{All} & \triangleq \text{PRE True \ THEN} & \text{Square} := \text{Arret} \land \text{Moat} := \text{Arret} \text{ END} & / \ast \{\ast\} \{\text{Arret}_\text{Pedestrian}\}! \ast/; \\
\text{Arret}_\text{Pedestrian} & \triangleq \text{PRE Square} = \text{Arret} \land \text{Moat} = \text{Arret} & \text{THEN} & \text{Pedestrian} := \text{Arret} \text{ END} & / \ast \{\ast\} \{\text{Arret}_\text{Pedestrian}\}! \ast/; \\
\text{Arret}_\text{Pedestrian} & \triangleq \text{PRE Square} = \text{Arret} \land \text{Moat} = \text{Arret} \land \text{Pedestrian} = \text{Arret} & \text{THEN} & \text{Pedestrian} := \text{Arret} \text{ END} & / \ast \{\ast\} \{\text{Arret}_\text{Moat}\}! \ast/
\end{align*}
\]

END

Figure 3.5.: Global Interrupting Traffic Machine - Part 2

machine they can be dropped in anywhere in the controller sequence. So consistency can be determined after the asynchronous query occurrences have been hidden.

3.2.1. The QUERY Annotation of a Consistent B Machine

The QUERY annotation is written /\ast \text{QUERY} \ast/ (the short form of the annotation is /\ast \text{?} \ast/). This annotation is added to indicate the use of query B operations which can follow any operation of the machine or the initialisation.

An example of its use in an arbitrary operation (NEXT annotations can not be mixed with QUERY annotations) \(\text{OP}_i\) is:

Definition 35 (Query Annotation).

\[
\text{OP}_i \equiv (P_i \mid B_i)/ \ast \text{QUERY} \ast/;
\]

In the above we state that \(\text{OP}_i\), the query operation, is always ready for execution. All operations enabled before query execution remain enabled on completion of the query operation. The proof of this claim can be verified by discharging the proof obligation given in definition 36.
Traffic_CTRL = Aller_Moat \rightarrow Arret_Moat \rightarrow
Aller_Square \rightarrow Arret_Square \rightarrow Traffic_CTRL

Global_Traffic_CTRL = Traffic_CTRL
\\Delta (Arret_All \rightarrow Aller_Pedestrian \rightarrow
Arret_Pedestrian \rightarrow Global_Traffic_CTRL)

Figure 3.6.: Global Interrupting Traffic Controller

MACHINE Defined_Traffic
SETS COMMAND = \{Arret, Aller\}
VARIABLES Moat, Square
INVARIANT
   Moat : COMMAND \land Square : COMMAND \land (Moat = Arret \lor Square = Arret)
INITIALISATION Moat, Square := Arret, Arret \land \{Arret\}_! \land /
OPERATIONS
Arret_All \equiv PRE \ True \ THEN \ Moat := Arret \land Square := Arret END
   \{Arret\}_! \land /

Aller_Moat \equiv PRE \ Moat = Arret \land Square = Arret \ THEN \ Moat := Aller END
   \{Arret\}_! \land /

Aller_Square \equiv PRE \ Moat = Arret \land Square = Arret \ THEN \ Square := Aller END
   \{Arret\}_! \land /

END

Figure 3.7.: Defined Traffic Machine

query operations \(QP_q\) maintains all predicates \(P\) on the B state as a consequence of not altering the machine state. The B construct \(skip\) has no effect on state and is guaranteed to terminate. \(skip\) has all the properties that are required of \(B_q\). Hence we say that \(B_q\) refines \(skip\) (see definition of machine refinement 6.0.1, page 131). As a refinement \(B_q\) can not behaviours different from \(skip\). Formally we write \(skip \subseteq B_q\) \((skip\ is\ refined\ by\ \(B_q\))\). The context of the refinement is kept simple: the variables of the machine and its refinement are the same \((xx = XX \ etc.)\). Hence the linking invariant \(J\) relates machine variable with refinement variables \(J = (xx = XX \land yy = YY \land ... \land zz = ZZ)\). The proof obligation to show that \(QP_q\ is\ a\ query\ operation\ and\ therefore\ has\ no\ effect\ on\ state\ is\ as\ follows:
3.2. The QUERY Annotations

\[ Moat\_CTRL = Arret\_All \rightarrow Aller\_Moat \rightarrow Moat\_CTRL \]
\[ Defined\_Traffic\_CTRL = \]
\[ Moat\_CTRL \]
\[ \Delta_{\{Arret\_All\}}(Aller\_Square \rightarrow Defined\_Traffic\_CTRL) \]

Figure 3.8.: Defined Traffic Controller

Definition 36 (QUERY Proof Obligation).

(1) \( \text{skip} \subseteq B_q \) in the context of \( J \)

\[ J \Rightarrow [B_q] \neg [\text{skip}] \neg J \] by (1) and definition of refinement 6.0.1

\[ J \Rightarrow [B_q]J \] by \( J \equiv \neg[\text{skip}] \neg J \)

where the refinement invariant \( J \)
maps all the variable of the machine
back to versions of themselves in the refinement,
i.e: \( J = (xx = XX \land yy = YY \land ... \land zz = ZZ) \)

QUERY Operations are always enabled:

(2) \( I \Rightarrow P_q \)

Note that \( [T]P_q \) follows from (2) for consistent machines as \( [T]I \)

Definition 37 (Query Annotation Shorthand).

\[ OP_i \equiv (P_i | B_i)/* ? */; \]

The annotation short hand is given in definition 37.

3.2.2. Controller Syntax and Auxiliary Functions with QUERY Annotations

The addition of the QUERY annotation does not extend the syntax of the controller given in definition 24. The definition of \( \text{init} \) (definition 25) and \( \text{guarded} \) (definition 26) are not
changed. We extend the new listing function to support the new annotation and introduce a
new function to deal with the QUERY annotations.

Definition 38 (QUERY Annotation Listing of Operations).
\[ \text{query}(OP_i) \equiv \text{true} \quad \text{if} \quad OP_i \equiv (P_i \mid B_i)\star \text{QUERY}\star /; \]
\[ \text{query}(OP_i) \equiv \text{false} \quad \text{otherwise} \]

3.2.3. Annotation Consistency of Controllers with QUERY Action

\(M_{-CTRL}\) is annotation consistent with \(M\) if \(\text{guarded}(M_{-CTRL})\) is true and if \(M_{-CTRL}\) is
initially-consistent with \(M\) and in all subsequent actions of \(M_{-CTRL}\) it is step-consistent with
\(M\). In short, an action of a controller is annotation consistent if the B machine \((M)\) enables it,
including the initial action. Previously it has been proven that if the controller is annotation
consistent with the machine it controls then the pair are non-divergent. Theorem 3 is given
which states that a query controller set in parallel with a consistent machine is non-divergent
if the query controller is annotation consistent with the query actions hidden. Removing the
queries means that consistency is determined as in section 2.6 or section 3.1.5. Accordingly
we give definitions of the hiding operator used on controllers to assist in the proof.

Definition 39 (Hide operator in the trace model).
\[ (a \rightarrow P)\setminus S = \quad P\setminus S \quad \text{if} \quad a \in S \]
\[ = \quad a \rightarrow (P\setminus S) \quad \text{if} \quad a \notin S \]
\[ (P \square Q)\setminus S = \quad (P \setminus S \square Q \setminus S) \quad \text{only true in the traces model} \]

Definition 40 (Annotation consistency of a Controller with Query Action). \(R\) is
annotation consistent if \(R\setminus S\) is annotation consistent. (It is determined in the same way as
in definition 33).

Definition 41 (Step-consistency of a Controller with Query Actions). Step-consistency
of the controller fragment \(R\), which may contain queries, is determined in the usual way but
with the queries hidden. The consistency of \(R\) is the consistency of \(R'\), where \(R' = R\setminus Q\) and
\(Q\) is the set that contains all of the operations annotated with a QUERY annotation. (It is
determined in the same way as in definition 34)
3.2. The query Annotations

query annotated operations do not interfere with the development of consistency. Their proof obligations, when discharged, demonstrate that the operation can be executed from any state that the machine can be put in by the initialisation or operations. In addition query operations do not alter the state of a machine. Hence, expectations set up by next operations carry through query actions in controllers. Figure 3.9 demonstrates this principle, where $R$ is consistent with the operation annotations.

\[
\begin{align*}
\text{op}_a & \equiv ... / \{\text{op}_b\}! * /; \\
\text{op}_b & \equiv ... / \{\text{op}_c\}! * /; \\
\text{op}_c & \equiv ... / * ? * /; \\
\text{op}_d & \equiv ...
\end{align*}
\]

\[
R = \text{op}_a \to \text{op}_c \to \text{op}_b \to \text{op}_c \to \text{op}_d \to ...
\]

Figure 3.9.: Carrying Through NEXT Annotation Expectations Through Query Operation

It is necessary to show that the controller and the machine when put in parallel do not diverge. Under the assumption that the controller with query operations is step-consistent, and that the machine is machine-consistent, theorem 3 states that the pair in parallel are divergence free.

**Theorem 3 (query annotated Machine Controller Combinations that are machine-Annotation Consistent and Annotation Consistent are Divergence Free).** If a query annotated B machine $M$ is machine-annotation consistent and the controller $M\_CTRL$ is annotation consistent with the annotations of machine $M$ then $M\|M\_CTRL$ is divergence free.

The proof of Theorem 3 is not undertaken directly by showing only that good traces are produced by $M\_CTRL$ as was the case of the previous theorems. The query annotation has no effect on the state of a machine. It is this property that is used to obtain a proof of non-divergence. We use lemma 7 which states that if a annotation consistent controller is good with its query actions hidden then it is good with them visible.

\[
M \text{ Machine Annotation Consistent and} \\
M\_CTRL \setminus Q \text{ Annotation Consistent} \\
\Rightarrow \text{trace}(M\_CTRL) \text{ good}
\]

The proof of Theorem 3 is as follows:
Chapter 3. The FROM and QUERY Annotations

1. $M$ machine annotation consistent \(\Rightarrow\) initial assumption
2. \((M_{CTRL} \setminus Q)\) annotation consistent \(\Rightarrow\) initial assumption
3. $traces(M_{CTRL})$ good \(\Rightarrow\) 2, lemma 7 below
4. $traces(M_{CTRL} \parallel M)$ good \(\Rightarrow\) 1,3 and theorem 2

Hence $M_{CTRL} \parallel M$ divergent free

This proof of theorem 3 is rudimentary because we reuse the earlier work undertaken to show that controller-machine pairs are machine-controller consistent without query operations. The proof of lemma 7 remains to do. In previous chapters annotation consistency was established by following the rules stated for annotation consistency of any controller element with relation to the machine it controlled. This chapter is no different except that query operations in the controllers are hidden. It must be shown that all the traces of a machine with query operations are good. In order to establish that traces with query operation are good, it is necessary to show that the traces are good with the queries hidden (lemma 7). In turn it is necessary to show that a trace terminates when it has query operations hidden (lemma 8). The proof of lemma 8 uses another lemma. It is shown, in lemma 9, that adding a query action to an existing sequence of actions that terminates produces a sequence that still terminates. The facts about termination are used to prove step-consistency of the controller and then to show that the machine in parallel with the controller is non-divergent.

Lemma 7 (Controllers with Queries actions are good if they are annotation consistent with the Queries hidden). Traces of controllers with query operation in them are good if they are annotation consistent with the query operations hidden.

\[ M_{CTRL}\setminus Q \text{ annotation consistent } \Rightarrow traces(M_{CTRL}) \text{ good} \]

July 6, 2008
The proof of lemma 7

1. $M_{CTRL} \setminus Q$ Annotation Consistency assumption: where $M_{CTRL}$ has queries actions and $Q$ is the set of all query actions in $M_{CTRL}$

2. $\text{traces}(M_{CTRL} \setminus Q) \text{ good}$ by 1, def. of annotation-consistency 33 (Every event is enabled by an annotation which implies divergence freedom.)

3. $\forall tr \in \text{traces}(M_{CTRL})$ assume any trace $tr$

4. $tr \setminus Q \text{ good}$ by 2 and 3

5. $tr \text{ good}$ by lemma 8 below

6. $\text{traces}(M_{CTRL}) \text{ good}$ by 5 and generalisation

QED

Lemma 8 (A Trace will Terminate if it Terminates with the Queries Hidden). Removing all query operations from a good trace produces good trace.

Proof of lemma 8

Proof by induction on $n = \#(tr \mid Q)$ Case 1: $n=0$ (for traces $tr$ that have no operations in the sequence)

1. $[tr]true$ initial assumption

2. $\#(tr \mid Q) = 0$ from induction step

3. $tr' = tr \setminus Q$ define $tr'$

4. $tr = tr'$ 2 and 3

5. $[tr']true$ 1 and 4 

true for $n = 0$

Case 2: $n=k+1$ (traces $tr$ that have $k+1$ query operations in the sequence)
Chapter 3. The FROM and QUERY Annotations

1. \( \#(tr ∩ Q) = k + 1 \)
   - Consider

2. \( \#(tr_1 ∩ (q) ∩ tr_2 ∩ Q) = k + 1 \)
   - From \( tr = tr_1 ∩ (q) ∩ tr_2 \)

3. \( \#(tr_1 ∩ tr_2 ∩ Q) = k \)
   - By 2 and \( \#(q) = 1 \)

4. \( (tr_1 ^ tr_2) \text{ is good} \)
   - By inductive hypothesis as \( \#(tr_1 ^ tr_2) = k \)

5. \( (tr_1 ^ (q) ^ tr_2) \text{ is good} \)
   - By \( q ∈ Q \) and lemma 9

Hence \( tr \) is good if \( tr \setminus Q \) is good for any number of queries in \( tr \).

**Lemma 9** (if \( tr_1 ^ tr_2 \) is good and \( q \) is a query then \( tr_1 ^ (q) ^ tr_2 \) is good). Adding a single query operation to a terminating sequence does not cause divergence.

Proof of lemma 9

1. \( tr_1 ^ tr_2 ∈ traces(M_{CTRL}\{q\}) \)
   - Assume

2. \( tr_1 ^ tr_2 \text{ is good} \)
   - Assume

3. \( \text{[init;} tr_1 \text{]}tr_2\text{]}true \)
   - By 3 and definition 15, page 30

4. \( \text{[init;} tr_1\text{]}[(tr_2)]true \)
   - By wp of ; (semicolon)

5. \( \text{[init;} tr_1\text{]}I \)
   - By machine consistency

6. \( \text{[init;} tr_1\text{]}[I ∧ [tr_2]true] \)
   - From 4 and 5

7. \( \text{[init;} tr_1\text{]}\text{[skip]}[(tr_2)]true \)
   - By 4 and definition of skip

8. \( \text{[init;} tr_1\text{]}[B_q][(tr_2)]true \)
   - By 7 and def. 36 (1) since \( q \) is a query

9. \( \text{[init;} tr_1\text{]}P_q \)
   - By 5 and def. 36 (2)

10. \( \text{[init;} tr_1\text{]}[P_q ∧ B_q][(tr_2)]true) \)
    - By 8 and 9

11. \( \text{[init;} tr_1\text{]}[(P_q \mid B_q)][(tr_2)]true) \)
    - By 10 and def. of \( P|B \)

12. \( \text{[init;} tr_1\text{; q; tr}_2\text{]}true \)
    - By 11 and def. of ; (semicolon)

13. \( tr_1 ^ (q) ^ tr_2 \text{ good} \)
    - By 12

July 6, 2008
3.2.4. Picturing the Query Annotation

The generic Query annotation given in definition 35, on page 65 on its own, is pictured as in Figure 3.10. Any transition can precede the current operation $o_p_i$ including itself where $o_p_i$ is a query operation.

![Figure 3.10.: Picturing the $o_p_i$ Query Annotation](image)

3.2.5. Examples Utilising Query

The Query is used to communicate the internal state to the environment. We illustrate the use of query operations in the running Carcassonne traffic light example. The requirement for the traffic control system for the main street of the walled City of Carcassonne is specified in section 2.9. The B machine depicted in Figure 3.11 offers query operations to determine the current state of the moat light, with this information the environment can make an informed decision about the next state of the traffic lights. The machines consistent controller is given in Figure 3.12. Although the state of the traffic lights would be obvious to an observer, it needs to be relayed to the controller using query operations. It is not obvious from the controller which set of lights will be go next. This must be decided elsewhere. The choice of which traffic light set is turned on next is resolved by the environment.
MACHINE Querying_Traffic
SETS COMMAND = {Arret, Aller}
VARIABLES Moat, Square
INVARIANT Moat : COMMAND \( \land \)
\( \text{Square} : \text{COMMAND} \land \)
\( (\text{Moat} = \text{Arret} \lor \text{Square} = \text{Arret}) \)
INITIALISATION Moat, Square := Arret, Arret / \( \ast \{\text{Arret}_\text{All}\}\ast / \)

OPERATIONS
\( \text{Arret}_\text{All} \equiv \text{PRE True THEN} \text{Moat := Arret} \parallel \text{Square := Arret END} \)
\( \ast \{\text{Arret}_\text{Moat}, \text{Arret}_\text{Square}\}\ast / ; \)
\( \text{Aller}_\text{Moat} \equiv \text{PRE Moat = Arret} \land \text{Square = Arret THEN} \text{Moat := Aller END} \)
\( \ast \{\text{Arret}_\text{All}\}\ast / ; \)
\( \text{Out} \leftarrow \text{Moat}_\text{Query} \equiv \text{PRE True THEN} \text{Out := Moat END} \ast \{\text{Arret}_\text{All}\}\ast / ; \)
\( \text{Aller}_\text{Square} \equiv \text{PRE Moat = Arret} \land \text{Square = Arret THEN} \text{Square := Aller END} \)
\( \ast \{\text{Arret}_\text{All}\}\ast / ; \)

END

Figure 3.11.: Querying Traffic Machine

Querying_CTRL = Arret_All →

\( \ast \{\text{Aller}_\text{Moat} \rightarrow \text{Moat}_\text{Query} ? \text{out} \rightarrow \text{Querying}_\text{CTRL} \} \ast )

\( \{\text{Aller}_\text{Square} \rightarrow \text{Moat}_\text{Query} ? \text{out} \rightarrow \text{Querying}_\text{CTRL} \) \)

Figure 3.12.: Querying Traffic Controller
4. Adding I/O and **NE**X**T** I**N**V**A**RI**A**NT to Annotations

4.1. Communication Between B Operations, Controller and Environment

The contribution of this chapter is to introduce operation input and output (I/O) into the annotations. The annotation can be used to constrain the values that operation input can take. Operation I/O is represented in the controller channels as values. Additional syntax would be required to constraint the values on the channels. This is not done instead the I/O constraints are dealt with only in the annotations.

The generic B machine $M$ depicted in Figure 2.2, on page 23 is extended for I/O in Figure 4.1, on page 76. The definition has variables, invariant, initialisation, and a set of operations $OP_1$ through to $OP_n$. However the operation are given inputs and outputs in this instance. The operations are defined in GSL. In the definitions that follow it is assumed that the precondition operations and machine are consistent in the B sense.

There are three sources from which information may emanate: the environment, the controller, or the B operations. The B operations can receive input from the environment and/or the controller. A single action may contain input from both the environment and controller simultaneously. The environment and controller can receive input from the B operations. In CSP$\parallel$B the B is subordinate to the controller, and consequently all communication with the environment, by the B, must be made via the CSP controller. Our approach is not so controller centred. Inputs are permitted to flow into the B operations from the controller or from the environment. The input from the environment is an input to the B and the controller. Input is asynchronously read from the environment into the controller and the B machine. This is instead of latching values into the controller in one step then outputting the same values to the B machine in a subsequent step.

The first operation of the generic machine given in Figure 4.1 has an output list $y_i$ and input parameter lists $v_i$ and $e_i$. The reason for separating out the input into two different lists is to differentiate between the different sources of input the operation may receive. The input $v_i$ (little $v$) is incident from the environment, whereas the $e_i$ input is incident from the controller. The list $V$ (big $V$) is used in the annotations. The $V$ list is a list of B machine variables that are used as actual parameters. $E$ is the list of expressions used as actual parameters for the input to the operation identified in the annotation. $L_i$ is the **NE**X**T** **I****N**V**A**R**I**A**NT which is
Chapter 4. Adding I/O and next invariant to Annotations

MACHINE $M$
VARIABLES $\gamma$
INVARIA NT $\gamma : T_{\gamma}$
INITIALISATION $\gamma : \in T_{\gamma}$
OPERATIONS

$$y_i \leftarrow O_{p_i}(v_i, e_i) \equiv (P_i | B_i) / \star \{O_{p_j}(v!E)\} \text{NEXT } \star /$$

$$/ \ast \text{NEXT INVARIA NT } L_i \ast /;$$

$$y_i \leftarrow O_{p_j}(v_j, e_j) \equiv (P_j | B_j); \quad \text{...}$$

$$y_n \leftarrow O_{p_n}(v_n, e_n) \equiv G_n \implies B_n$$

END

where

$$v_j = v_{j_1}, v_{j_2}, ..., v_{j_p}$$

$$e_j = e_{j_1}, e_{j_2}, ..., e_{j_k}$$

$v_j$ and $e_j$ are free in $B_i$

$$T_v = T_1, T_2, ..., T_g$$

$$T_e = T_{g+1}, T_{g+2}, ..., T_{g+h}$$

$$P_j = v_{j_1} \in T_1 \land ... \land v_{j_p} \in T_p \land$$

$$e_{j_1} \in T_{g+1} \land ... \land e_{j_k} \in T_{g+h}$$

$$V = ?\gamma_1, ?\gamma_2, ..., ?\gamma_s$$

$$T_{\gamma} = T_{\gamma_1}, T_{\gamma_2}, ..., T_{\gamma_s}$$

$$E = E_1, E_2, ..., E_h$$

$$T_{AB} = T_{\gamma_1} \subseteq T_1 \land ... \land T_{\gamma_s} \subseteq T_g \land E_1 \in T_{g+1} \land ... \land E_{n+p} \in T_{g+h}$$

Figure 4.1.: A B machine with Operations with I/O.

The information flow between the environment, B and controller are illustrated by the diagrams given in figures 4.2, on page 77 and 4.3, on page 77. The diagrams illustrate the sources of I/O. The annotation of $O_{p_1}$ is illustrated. The operation has a variable list of both environmental and controller inputs; $V$ and $E$, respectively. The typing in the precondition sub-predicates is shown. The form of a next invariant (next invariant) is given in $O_{p_1}$. The predicate associated with the annotation is $L_i$. 

July 6, 2008
4.1. Communication Between B Operations, Controller and Environment

In Figure 4.2 the controller schedules an action $O_{pi}$ and gets the associated input required by the environment from the environment. In Figure 4.3 the operation $O_{pi}$ outputs to the environment and controller in response to initiation by the controller. In the figures all possible signals vectors are illustrated through in practice they may not all be used. The scheduled operation may read from either the environment or controller, or from neither. Similarly, the B operation may output to either the environment or the controller, or neither. From the B operation's point of view the $y$ list represents the outputs of the machine operation. The $v$ represents the operation's inputs supplied from the environment, whereas the $e$ represents the inputs supplied by the controller. In the definition of annotations the B operation output parameters depicted by $y$ are not utilised. All the annotation up to and including this chapter use annotations to reason about the invocation of operations, which makes the output of the operations being invoked not appropriate for inclusion.
4.2. I/O NEXT Annotations and the NEXT INVARIANT

In the controller a prefixed action may be used to introduce operations with I/O. From the perspective of the controller, output variables can be drawn from process variables or previously instantiated inputs in the control flow. As defined in chapter 2 the NEXT annotation of an operation \( O_p \) introduces another operation \( O_p \), or set of operations \( O_p \), which should be enabled after \( O_p \) has been executed. The updated annotation is shown in definition 42.

Definition 42 (NEXT Annotation Syntax with I/O for Operations).

\[
y_i \leftarrow O_p(v_i, e_i) \equiv \text{PRE } P_i \text{ THEN } B_i \text{ END } /* \{ O_p ?V_j\{3\} \}_{\text{NEXT }*/ ;
\]

A controller action will have a corresponding B operation.

\[
\text{VARIABLE } V_1
\]

...  

\[
y_i \leftarrow O_p(v_i, e_i) \equiv \text{PRE } P_i \text{ THEN } B_i \text{ END } /* \{ O_p ?V_j\{3\} \}_{\text{NEXT }*/ ;
\]

\[
y_j \leftarrow O_p(v_j, e_j) \equiv \text{PRE } P_j \text{ THEN } B_j \text{ END ;}
\]

In the above text the annotation /* \{ O_p ?V_j\{3\} \}_{\text{NEXT}*/ has the first actual parameters set to a B variable which has a fixed type declared in the INVARIANT clause. It matches with the \( v_j \) list, which is the environmental input. Marking the variable in the annotation with a ? means that the variable is supplying input from the environment. In chapter 6 the consequence of being an environmental variable will be explained. The second actual parameter is \( y_i \) it is the output of the current operation. The third actual parameter, 3, is not fixed by the context and therefore is not set by the current operation. The second and third parameters match with the \( e_j \) list as they are outputs of the annotations.

4.2.1. NEXT INVARIANT

Adding variables (flags) to the specifications whose only purpose is to direct the follow of control of the machine has been avoided where possible. The execution of a machine is
the sequential execution of its operation. The controller captures the intended execution behaviour. The consistency of the machine-controller pair is dependent on the consistency of the machine itself, the machine-annotations pairing, and the annotations-controller pairing. The policy to-date ensures that each operation in a sequence is enabled by the operation before it. This restriction is now relaxed to allow an operation $op_i$ to partially satisfy the precondition of $op_j$. In way of compensation however a local NEXT INVARIANT is introduced in comments in the operation it effects, to ensure that any next operation is fully enabled. The NEXT INVARIANT is a local predicate that must be true before and after the current operation, $op_i$, is executed. Together, the body of the current operation and its NEXT INVARIANT must establish the precondition of the next operation $op_j$. Rather than augment the precondition with control flow flags we capture this information in a special predicate. An example of the use of a NEXT INVARIANT is given in definition 43.

**Definition 43 (NEXT INVARIANT Annotation Syntax for Operations).**

$$OP_i = (P_i | B_i) // \{ \text{predicate} \} \text{NEXT INVARIANT} * /$$

The view of the world at this level of abstraction is of atomic action for both controller actions and B operations. That is to say, that a B operation is initiated and the output is made available in the same atomic action. In implementation a time delay will separate the invocation of the actions that may be supplied with inputs and the resulting output result. The time separating the input and output is likely to be at least one clock period if the logic is made synchronous to a clock waveform. Hence a goal of refinement is to separate out the input and output phase to reflect actual implementation constraints.

The operations depicted in definition 44 and definition 45 have a number of parameters. The $y_i$ input parameter corresponds to the list $y_1, ..., y_i$. It is noted that the annotation does not have output parameters. The annotation does have an input list $V_j$ which corresponds to the list $?V_1?...?V_g$. When it describes type $T^1$ is generally subscripted. The $V$ parameters are set by the environment so all that can be guaranteed is their type. The $T^v$ annotation parameters correspond to the types of the $v$ inputs of an operation. They are defined in Figure 4.1 introduced through $T_{AB}$. The $v_i$ actual parameters correspond to the $v_{i1}, ..., v_{ig}$ input parameters list in the operation. Similarly, the $E_1, ..., E_h$ parameters listed $(E)$ correspond to an operation input list $x$. The $e_j$ input corresponds to the list $e_{j1}, ..., e_{jk}$. $T_e$ is the type in the precondition $P_j$ associated with $e_j$. $V$ is used to signify value. $E$ signifies expression. The $V$ annotation parameters are variables typed in the machine. $v$ are input operation parameters typed by the precondition of the operation. Although, $E$ stands for expression, each expression must evaluate to a specific value, otherwise the B operation could not be called. Note that in examples of B machines with annotated operations a particular group of parameters may not be supplied. This corresponds to the list of variables being empty.

### 4.2.2. I/O NEXT and NEXT INVARIANT Proof Obligations

**Definition 44 (Proof Obligations of INITIALISATION Annotated with I/O NEXT).**

Given the following B initialisation and B operation with next invariant $L_j$:

$^1T$ is overloaded it is used to signify type or initialisation
Chapter 4. Adding I/O and NEXT INVARIANT to Annotations

VARIABLES $V_1, \ldots, V_g$

INVARIANT $V_i \in T_{V_i} \land \ldots \land V_g \in T_{V_g}$

INITIALISATION $T /* \{ O_{P_j} ? \{ V \} \} \} NEXT */$

$$y_j \leftarrow O_{P_j}(v_j, e_j) \equiv \text{PRE } P_j \text{ THEN } B_j \text{ END } /*\text{NEXT INVARIANT } L_j*/$$

The following PO arises:

$$(T_{A,B} \land [T][(P_j \land L_j)[V, E/v_j, e_j]])$$

where

$e_j = e_{j1}, e_{j2}, \ldots, e_{jk}$

$v_j = v_{j1}, v_{j2}, \ldots, v_{jk}$ are free in $T$, where $T_{A,B}$ is defined in Figure 4.1

The proof obligation states that the following holds:

1. $T_{A,B}$: the type of the formal parameters specified in the annotations for the initialisation is a subset or equal to the type of the actual parameter of the next operation actual parameter. This is a type check. It is included in the PO because it is not carried out by the BToolkit.

2. $([T][(P_j \land L_j)[V, E/v_j, e_j]]$: In this part of the PO there are two things to consider. Firstly, the annotation parameter values are substituted into the precondition and the next invariant of the next operations. Secondly, the initialisation of the machine establishes the predicate $(P_j \land L_j)$ updated with the actual parameters.

Definition 45 (Proof Obligations of Operations Annotated with I/O NEXT). We give the form of the B operation and its related PO (where $v_j$ and $e_j$ are the formal parameters of $O_{P_j}$):

$$y_i \leftarrow O_{P_j}(v_j, e_j) \equiv \text{PRE } P_i \text{ THEN } B_i \text{ END } /* \{ O_{P_j}(V) \} \} \} NEXT */ /* \{ \text{NEXT INVARIANT } L_i \} */$$

The following PO arises:

$I \land P_i \land L_i \Rightarrow (T_{A,B} \land [E][(P_j \land L_j)[V, E/v_j, e_j]]) \land$

$$(I \land P_i \land L_i) \Rightarrow [B_i](L_i)$$
where
\[ \varepsilon_j = \varepsilon_{j1}, \varepsilon_{j2}, \ldots, \varepsilon_{jk}, \]

\[ \nu_j = \nu_{j1}, \nu_{j2}, \ldots, \nu_{jk} \] are free in \( B_i \), where \( T_{AB} \) is defined in Figure 4.1.

The proof obligation states that given that the machine invariant, current precondition and next invariant of the current operation hold, then it implies that the following holds:

1. \( T_{AB} \): the types of the formal parameters specified in the annotations for the current operation are a subset or equal to the types of the actual parameters of the next operation's actual parameters. This is a type check. It is included in the PO because it is not carried out by the BToolkit.

2. \([B_j][(P_j \land L_j)[V, E/\nu_j, \varepsilon_j]]\): In this part of the PO there are two things to consider. Firstly, the annotation parameter values are substituted into the precondition and the next invariant of the next operations. Secondly, the body of the current operation establishes the predicate \((P_j \land L_j)\) updated with the actual parameters.

It also must be established that the current local invariant \( \text{NEXT INVARIANT} \) is re-established by the current operation:

1. \( I \land P_i \land L_i \Rightarrow [B_j][(L_i)] \)

### 4.3. Examples of the Use of I/O NEXT

In the example given in Figure 4.4 the parameter in the annotation is an expression type parameter, which means it obtains its value from some action of the next operation it annotates. In this case it can be evaluated as 2. Setting the input parameter of the next operation \( OP_j \) to 2 establishes its precondition.

The body of the current operation fixes the value of the \( nn \) variable of the next operation's precondition. By definition 45 the annotation POs for operation \( OP_i \) in Figure 4.4 is given as follows (Note that the invariant \( I \), the precondition \( P_i \), and the next invariant \( L_i \) have not been defined in this example and therefore are not in the PO below. Also note that the portion of the PO relating to type correctness \((T_{AB})\) has not been included (assuming that \( nn \in \mathbb{N} \) then \( T_{AB} = \mathbb{N} \subseteq \mathbb{N} \), which is trivially true)):

\[ [nn := 2](xx \mod 2 = 0 \land nn = 2)[nn / xx] \]

which reduces to:

\[ \text{true} \]

To construct a controller based on the annotations the expressions used as actual parameter like \( nn \times 2 \) have to evaluate to a constant value. Hence a valid controller fragment to drive the operations in the \textit{Evens} machine is as follows:

\[ \ldots \rightarrow OP ITV \rightarrow OP J'2 \rightarrow \ldots \]
Chapter 4. Adding I/O and NEXT INVARIANT to Annotations

MACHINE Evens

... 

OPERATIONS

\[ \text{OP}_i(yy) \triangleq \text{PRE} \quad \text{THEN} \quad nn := 2 \quad \text{END} \]
\[
\begin{array}{l}
/\star \{ \text{OP}_j ((nn)) \} \text{ NEXT } */ \\
\end{array}
\]

\[ \text{OP}_j(xx) \triangleq \text{PRE} \quad xx \in \mathbb{N} \land xx \mod 2 = 0 \land nn = 2 \quad \text{THEN} \quad \ldots \quad \text{END} \]

END

Figure 4.4.: Evens Machine

where \( p \) is defined earlier in the controller. In this case the controller, like in the CSP\|B approach, is driving the B operations.

In Figure 4.5 the variable \( \text{out-mm} \) is available to the annotation, because it is an output of the current operation. All inputs and outputs of the current operation are available to the annotations of current operations. Writing \( mm - 1 \) in the annotation would lead to state being carried in the CSP controller. Using \( \text{out-mm} \) avoids this. By definition 45 the annotation POs for operation \( \text{OP}_i \) in Figure 4.5 is (type correctness of annotation not included):

\[
[\text{out-mm} := mm - 1](zz \in \mathbb{N} \land xx \in \mathbb{N} \land xx < mm)[M_1, out-mm/zz, xx]
\]
\[
\equiv
\]
\[
(M_1 \in \mathbb{N} \land mm - 1 \in \mathbb{N} \land mm - 1 < mm)
\]
\[
\equiv
\]
\[
\text{true}
\]

A valid controller fragment for the Channelling machine could have the following form:

\[ \ldots \rightarrow \text{OP}_j?M_1!\text{out-mm} \rightarrow \ldots \]

The value denoted by \( M_1 \) is provided by the environment. It is an input to the controller and the B operation, whereas \( \text{out-mm} \) is issued by the controller and received by the operation. The arrangement of controller and operations is given in Figure 4.6. Arrows with labels signify information flow. The arrow without a label from \( \text{OP}_i!\text{out-mm!}xx \) to \( \text{OP}_j?M_1!\text{out-mm} \) signifies control flow in the CSP. The boxes at the bottom of the figure contain B operations. The arrangement is similar in CSP\|B, except that the environment can feed directly into the B operations. In chapter 6 a complete break from the CSP\|B approach is made.
MACHINE  
Chanelling
VARIABLES  
\[ mm, M1 \]
INVARIANT  
\[ mm \in \mathbb{N} \land mm \in \{0...10\} \land M1 \in \mathbb{N} \]
INITIALISATION  
\[ M1 :\in \mathbb{N} \mid mm := 2 /\ast \{ \ldots \} \text{NEXT} \ast/ \]
OPERATIONS  
\[
\begin{align*}
\text{out}_mm & \leftarrow \text{OPi}(xx) \triangleq \\
\text{PRE} & \ldots \text{THEN} \ldots \parallel \text{out}_mm := mm - 1 \text{ END} \\
& \ast/ \{\text{OPj} (IM1 !\text{out}_mm)\} \text{NEXT} \ast/;
\end{align*}
\]
\[
\begin{align*}
\text{OPj}(xx,xx) & \triangleq \text{PRE} xx \in \mathbb{N} \land xx \in \mathbb{N} \land xx < mm \text{ THEN} \ldots \text{END} \\
& \ast/ \{\ldots\} \text{NEXT} \ast/;
\end{align*}
\]
END

Figure 4.5.: Example of a NEXT Annotation with I/O
4.4. I/O NEXT Annotation Listing Functions

The listing functions are extended to support the annotation with I/O. The functions return particular aspects of the associated annotation. The operation - name function returns the operation set of the annotation. The operation - env - in function returns the input parameters set by the environment of an operation from the annotation set. The operation - ctrl - in function returns the set of annotation input parameters incident from the controller for a given operation in the annotation set. The operation - env - type function returns the set of types associated with the environment input parameters of an operation in the annotation. The operation - ctrl - type returns the list of types associated with the control data in an annotation. The context where it is not given by the functions is given in Figure 4.1.

4.4.1. Annotation Operation Listing Functions

Definition 46 (Annotation Operation Listing for Operation Names).

\[
\text{operation - name}(Y \leftarrow OP_i(V, E) \equiv (P_i \mid B_i)/ \ast E \text{ NEXT } \ast / ) = OP_i
\]

\[
\text{operation - name}(\text{INITIALISATION } \ast / \ast X \text{ NEXT } \ast / ) = \text{INITIALISATION}
\]
Definition 47 (Annotation Operation Listing for Operation Outputs).

\[
\text{operation} - \text{output}(OP_j(V, E)) = \{y_j\}
\]

where \(OP_j\) is an arbitrary operation: \(y_j \leftarrow OP_j(V, E) \equiv ...\)

Definition 48 (Annotation Operation Listing for Operation Environmental Inputs).

\[
\text{operation} - \text{env} - \text{in}(OP_i(?V!E)) = V
\]

Definition 49 (Annotation Operation Listing for Operation Controller Inputs).

\[
\text{operation} - \text{ctrl} - \text{in}(OP_i(?V!E)) = E
\]

Definition 50 (Annotation Operation Listing for Operation Environment Types).

\[
\text{operation} - \text{env} - \text{type}(OP_i(?V!E)) = T_v
\]

where the context is given in Figure 4.1

Definition 51 (Annotation Operation Listing for Operation Controller Types).

\[
\text{operation} - \text{ctrl} - \text{type}(OP_i(?V!E)) = T_c
\]

where the context is given in Figure 4.1

4.4.2. INITIALISATION I/O NEXT Annotation Listing Functions

The listing functions are used to extract the actions for the proof work. The direction of the I/O is not important. Hence, \(op_{ps,y}.V!.E\) is constructed instead of \(op_{ps}?y?V!E\). The listing functions given in definitions 46, 47, 48, 49, 50 and 51 are used to assemble the parts of the actions that can be used in the definition of annotation controller consistency. The original next function definitions have been overloaded for I/O.
Chapter 4. Adding I/O and NEXT INVARIANT to Annotations

**Definition 52 (NEXT with I/O Annotation Listing for Initialisation).**

\[
\text{next}(	ext{INITIALISATION} \ldots / \ast X \ast /) = \\
\{ op_x, y, V, E \mid op_x \in \text{operation} - \text{name}(X) \land \\
\text{operation} - \text{output}(op_x) = y \land \\
\text{operation} - \text{env} - \text{in}(op_x) = V \land \\
\text{operation} - \text{ctrl} - \text{in}(op_x) = E \}
\]

Note however that as \( V \) and \( E \) are used to introduce types then the NEXT listing function should strictly return a family of operations invoke all with a different value of the type. For brevity we do not enumerate the types. So for example in Figure 4.7, on page 98, \( \text{next}(	ext{INITIALISATION}) = \text{Set}([7..10]), \) because \( \text{store}_\text{env}_\text{in} \in 7..10. \) Strictly, the formulation should be \( \text{next}(	ext{INITIALISATION}) = \{ \text{Set}(7), \text{Set}(8), \text{Set}(9), \text{Set}(10) \} \)

### 4.4.3. Operation I/O NEXT Annotation Listing Function

When the operations were solitary operations they were indistinguishable from an action. Here we treat actions with inputs and output the same as operations with inputs and outputs. Strictly we should restate the functions for both actions and operations \( \text{next}(y \leftarrow \text{OP}_x(v_x, e_x) \ldots / \ast X \text{NEXT} \ast /) = \text{next}(op_x?y_x?v_x!e_x)). \) This will not however added anything new.

**Definition 53 (NEXT Annotation Listing).**

\[
\text{next}(y_x \leftarrow \text{OP}_x(v_x, e_x) \ldots / \ast X \text{NEXT} \ast /) = \\
\{ op_x, y, V, E \mid op_x \in \text{operation} - \text{name}(X) \land \\
\text{operation} - \text{output}(op_x) = y \land \\
\text{operation} - \text{env} - \text{in}(op_x) = V \land \\
\text{operation} - \text{ctrl} - \text{in}(op_x) = E \}
\]

Definition 53 illustrates how the actions are constructed from operation NEXT annotations with the listing functions.

### 4.5. I/O FROM Annotations

In this section the three annotations are re-introduced with I/O: the FROM-ANY, FROM-SET, and FROM-SET. These annotations are used with NEXT annotations. In this chapter we restrict the annotations in operation to one from an annotation, which should be accompanied with one next annotation.
4.5. I/O FROM ANY Annotation

The introduction of I/O changes the form of the FROM ANY annotation. It is augmented with a set of input constraints: /* FROM ANY { ? V | E } */. The input from the environment (? V) is separately introduced from the input from the controller (|E). Expanding the vectors out gives the following signature for the annotation: /* FROM ANY { ? V_{j1} ... ? V_{jt} | E_{j1} ... | E_{jt} } */

The definition of the use of the annotation is given in definition 16, which we restate here:

Definition 54 (FROM ANY with I/O Annotation Syntax for Operations).

\[ y_j \leftarrow OP_j(v_j, e_j) \equiv (P_j \mid B_j) / * FROM ANY { ? V | E } */; \]

The annotation states that the interrupting operation can be called with the given parameters. The previous PO definition 17 is updated to include the I/O.

Definition 55 (I/O FROM ANY Proof Obligations). We give the form of the B operation and its related PO (the NEXT INVARIANT introduced in definition 43 has been included to show the most general form of the PO):

\[ y_j \leftarrow Op_j(v_j, e_j) \equiv \text{PRE} P_j \text{ THEN } B_j \text{ END} \]

/* FROM ANY { ? V | E } */

/* NEXT INVARIANT L_j */

The following PO arises:

\[ (\forall x \in \text{OPERATIONS} \cdot I \land P_x \land L_x \Rightarrow (T_{AB} \land [B_x]((P_j \land L_j)[V_j, E_j/v_j, e_j]))) \land \]

\[ (T_{AB} \land [(T](P_j \land L_j)[V_j, E_j/v_j, e_j])) \]

where

T with no subscript is the initialisation of the machine.

\[ e_j = e_{j1}, e_{j2}, \ldots, e_{j_t}, \]

\[ v_j = v_{j1}, v_{j2}, \ldots, v_{j_t} \text{ are free in } B_t, \text{ where } T_{AB} \text{ is defined in Figure 4.1} \]

Definition 55 takes into account the variabilities of the inputs that can be taken by the current operation OPj and its predecessor operation OPi. Every possible input of the previous
operation $OP_i$ must enable the precondition of $OP_j$. The FROM-ANY annotation can be used in the same operation as the NEXT annotation.

### 4.5.2. The I/O FROM-SET Annotation

The FROM-SET annotation is given in definition 56 below.

**Definition 56 (FROM-SET with I/O Annotation Syntax for Operations).**

```latex
/* FROM-SET { OP_k,...,OP_i } {?V \& E } */
```

The vector lists are inputs from the environment and controller, respectively.

The short form of the annotation is:

```latex
/* \{ OP_k,...,OP_i \} {?V \& E } */
```

This annotation is added to indicate the use of interrupting B operations that can follow any operations listed in the operation annotation. It may not follow the initialisation action. The input parameter lists $V$ and $E$ supply the input for the interrupting operation, not the operations that are interrupted.

**Definition 57 (FROM-SET Proof Obligation).** We give the form of the B operation and its related PO:

```latex
y_j \leftarrow OP_j(v_j,e_j) \triangleq \text{PRE } P_j \text{ THEN } B_i \text{ END}
```

```latex
/* FROM-SET { OP_k,...,OP_i } {?V \& E } */
```

```latex
/* NEXT INVARIANT L_j */
```

The following PO arises:

$$(\forall x \in \{ OP_k,...,OP_i \} \cdot I \land P_x \land L_x) \Rightarrow (T_{AB} \land [B_z((P_j \land L_j)[V_j,E_j/v_j,e_j]))$$

where

- $e_j = e_{j_1}, e_{j_2},..., e_{j_h}$
- $u_j = u_{j_1}, u_{j_2},..., u_{j_g}$ are free in $B_i$, where $T_{AB}$ is defined in Figure 4.1

### 4.5.3. The I/O FROM-SET-INIT Annotation

The FROM-SET-INIT annotation is given in definition 58 below.
4.5. I/O FROM Annotations

Definition 58 (FROM-SET-INIT with I/O Annotation Syntax for Operations).

\[
/* \text{FROM-SET-INIT}\{OP_k, ..., OP_i\}\{?V_j, !E_j\}* /
\]

The vector lists are inputs from the environment and controller, respectively. The short form of the annotation is

\[
/* \forall X\{OP_k, ..., OP_i\}\{?V_j, !E_j\}* /
\]

This annotation is added to indicate the use of interrupting B operations that can follow any operations listed in the annotation or the initialisation action of the machine. An example of its use in an arbitrary operation \(OP_j\) is:

Typically the FROM-SET-INIT annotation will be used in the following way:

\[
OP_j(v_j, c_j) \equiv P_j \mid B_i \mid /* \forall X \{OP_k, ..., OP_i\}\{?V_j, !E_j\} */;
\]

In the above we state that after the execution of any of the operations in the given set or the execution of the initialisation \(OP_i\) will always be available to execute. The annotation gives rise to the following proof obligation:

Definition 59 (FROM-SET-INIT Proof Obligation). We give the form of the B operation and its related PO including a predicate \(L_j\) arising from the NEXT INVARIANT:

\[
y_j \leftarrow OP_j(v_j, c_j) \equiv \text{PRE} \quad P_j \quad \text{THEN} \quad B_j \quad \text{END}
\]

\[
/* \text{FROM-SET-INIT} X \{?V_j, !E_j\} */
\]

\[
/* \text{NEXT INVARIANT} \ L_j */
\]

where \(X = \{OP_k, ..., OP_i\}\)

\[
(V \in X \cdot I \wedge P_x \Rightarrow (T_{AB} \wedge [B_x)((P_j \wedge L_j)[V,E/v_j,c_j]))) \wedge
\]

\[
(T_{AB} \wedge ([T](P_j[V,E/v_j,c_j])))
\]

where

\[
e_j = e_{j1}, e_{j2}, ..., e_{jh},
\]

\[
v_j = v_{j1}, v_{j2}, ..., v_{jg} \text{ are free in } B_i, \text{ where } T_{AB} \text{ is defined in Figure 4.1}
\]
The listing functions are extended to support the new annotations.

**Definition 60 (I/O FROM-ANY Operation Annotation Listing).**

\[
\text{from} - \text{any} - \text{op}(y_j \leftarrow OP_j(v_j, e_j)) = \text{true} \\
\text{if } y_j \leftarrow OP_j(v_j, e_j) \equiv (P_j | B_j)/* ! X \{V_j, E_j\} */; \\
\text{from} - \text{any} - \text{op}(y_j \leftarrow OP_j(v_j, e_j)) = \text{false} \quad \text{otherwise}
\]

**Definition 61 (Annotation Operation Listing for I/O FROM-SET Operation Names).**

\[
\text{operation} - \text{from} - \text{set} - \text{name}(y_j \leftarrow OP_i(v_j, e_j)) \equiv (P_i | B_i) \\
/* FROM-SET X \{V_j, E_j\} */; = X
\]

**Definition 62 (I/O FROM-SET Annotation Listing).**

\[
\text{from} - \text{set} - \text{op}(y_i \leftarrow OP_i(v_i, e_i)) = \{ \text{op}_x.y_i,v_i,e_i \\
| \forall \text{op}_x \in \text{operation} - \text{from} - \text{set} - \text{name}(OP) \\
\text{operation} - \text{output}(\text{op}_x) = y_i \land \\
\text{operation} - \text{env} - \text{in}(\text{op}_x) = v_i \land \\
\text{operation} - \text{ctrl} - \text{in}(\text{op}_x) = e_i \land \\
\text{operation} - \text{env} - \text{type}(\text{op}_x) = V_i \land \\
\text{operation} - \text{ctrl} - \text{type}(\text{op}_x) = E_i \land \\
\text{u_i} \in T_v \land \\
\text{e_i} \in T_e \}
\]

where \( OP = y_i \leftarrow OP_i(v_i, e_i)(P_i | B_i)/* FROM-SET\_INIT X \{V_i, E_i\} */; \)

and \( T_v \) and \( T_e \) are defined in figure 4.1

The \text{from} - \text{set} - \text{op} function returns an action in which the events channels are not given type or direction. The typing is checked in the machine-annotation consistency PO checking.
Definition 63 (Annotation Operation Listing for I/O FROM-SET-INIT Operation Names).

\[
\text{operation} \leftarrow \text{from} \leftarrow \text{set} \leftarrow \text{init} \leftarrow \text{name}(Y_i \leftarrow OP_i(V, X) \equiv (P_i \mid B_i))
\]

\[
/* \text{FROM-SET-INIT } X \{?V_i,?E_i\} */ ; ) = X
\]

where \( X \) is an arbitrary set of operations.

Definition 64 (I/O FROM-SET-INIT Annotation Listing).

\[
\text{from} \leftarrow \text{set} \leftarrow \text{init} \leftarrow \text{op}(y_i \leftarrow OP_i(u_i, e_i) = \{op_x \mid y_i \leftarrow \text{operation} \leftarrow \text{from} \leftarrow \text{set} \leftarrow \text{init} \leftarrow \text{name}(OP).
\]

\[
\text{operation} \leftarrow \text{output}(op_x) = y_i \land
\]

\[
\text{operation} \leftarrow \text{env} \leftarrow \text{in}(op_x) = u_i \land
\]

\[
\text{operation} \leftarrow \text{ctrl} \leftarrow \text{in}(op_x) = e_i \land
\]

\[
\text{operation} \leftarrow \text{env} \leftarrow \text{type}(op_x) = V_i \land
\]

\[
\text{operation} \leftarrow \text{ctrl} \leftarrow \text{type}(op_x) = E_i \land
\]

\[
v_i \in T_v \land
\]

\[
c_i \in T_e
\]

where \( OP = y_i \leftarrow OP_i(u_i, e_i) \equiv (P_i \mid B_i) /* \text{FROM-SET-INIT } X \{?V_i,?E_i\} */ ; X \)

and \( T_v \) and \( T_e \) are defined in figure 4.1.

4.5.5. Controller language

Definition 65 (Controller Syntax with I/O Extensions).

\[
R ::= \\
\quad a?y?ule \rightarrow R(y) \\
\quad R \sqcup R \\
\quad R \triangledown R \\
\quad R \triangleleft X R \\
\quad R \triangleright X R \\
\quad S(p)
\]

The controller language defined in definition 65 extends the language given in definition 24 with I/O. The link between operation and action is maintained. A operation with I/O is
Chapter 4. Adding I/O and next invariant to Annotations

represented in the controller syntax as an action with I/O. The meaning of consistency is extended by updating the \( \text{init} \) functions.

**Definition 66 (init on CSP Controller Process with I/O Extensions).**

\[
\begin{align*}
\text{init}(a?y?vle \rightarrow R1) &= \{a,y,v,e\} \\
\text{init}(R1 \parallel R2) &= \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(R1 \_\Delta R2) &= \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(R1 \_\lambda X R2) &= \text{init}(R1) \\
\text{init}(R1 \_\lambda X R2) &= \text{init}(R1) \cup \text{init}(R2) \\
\text{init}(S(p)) &= \text{init}(R(p))
\end{align*}
\]

An action prefix must appear with output preceding input, from left to right, with the input from the environment proceeding the input from the controller. In definition 66 the \( \text{init} \) function extracts the action of the action prefix. The outputs and inputs of the action marry with the outputs and inputs of the \( B \) operation the action represents: the controller action outputs to the inputs of the \( B \) machine and the output of the \( B \) operation is input to the action.

The \( \text{guard} \) function is unchanged by the introduction of I/O (definition 67)

**Definition 67 (guarded on CSP controller process with I/O).**

\[
\begin{align*}
\text{guarded}(a?y?vle \rightarrow R1) &= \text{true} \\
\text{guarded}(R1 \parallel R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \_\Delta R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \_\lambda X R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(S(p)) &= \text{false} \\
\text{guarded}(S(p) \_\triangle R(p)) &= \text{true} \quad \text{if} \ \text{guarded}(R(p))
\end{align*}
\]

4.6. I/O NEXT and FROM Annotation Consistency

As before, consistency is broken down into annotation consistency and step-consistency. Definitions 68 and definition 69 expands on the previous definition 13 and definition 14. The three interrupt operators are modelled in control constructs. In case 3 of definition 68 the control fragment with the interrupt operator is used. It is initially-consistent if the first process is initially-consistent and all the initial actions of \( R2 \) are initially consistent. Initial-consistency in definition 68 is determined directly using the properties of the associated annotations. Annotation consistency requires that there is both initial-consistency and step-consistency.

**Definition 68 (Annotation-Consistency with I/O).** Where \( M \) is a machine and \( M\_CTRL \) is its controller.
4.6. I/O NEXT and FROM Annotation Consistency

1. $a?y?vle \rightarrow R$

   is annotation consistent with Machine $M$ if $a?y?vle \rightarrow R$ is initially consistent and
   $a?y?vle \rightarrow R$ is step-consistent with $M$

   $a?y?vle \rightarrow R$ is initially-consistent if $a?y?vle \in \text{next(INITIALISATION)}$

2. $R1 \square R2$

   is annotation consistent with Machine $M$ if $R1$ and $R2$ are annotation consistent with $M$.

3. $R1 \triangle R2$

   is annotation consistent with Machine $M$ if $R1$ is annotation consistent and
   $\forall \text{op} \in \text{init}(R2) \cdot \text{from} \equiv \text{any(op)} \equiv \text{true}$ and $R2$ is step-consistent with $M$

4. $R1 \triangle X R2$

   is annotation consistent with Machine $M$ if $R1$ is annotation consistent with $M$ and
   if $\forall \text{op} \in \text{init}(R2) \cdot X \subseteq \text{from} \cdot \text{set}(\text{op})$, and $R2$ is step-consistent with $M$

5. $R1 \triangle X \backslash X R2$

   is annotation consistent with Machine $M$ if $R1$ is annotation consistent with $M$ and
   if $\forall \text{op} \in \text{init}(R2) \cdot X \subseteq \text{from} \cdot \text{set} \cdot \text{i}(\text{op})$, and $R2$ is step-consistent with $M$

6. $S(p)$

   is annotation consistent with Machine $M$

   A family of recursive definitions $S(p) \equiv R(p)$ is annotation consistent with $M$'s annotations if each $R(p)$ is annotation consistent with $M$'s annotations.

Definition 69 (Step-Consistency with I/O). The step-consistency of the controller fragment $R$ is defined as follows:

1. $a?y?vle \rightarrow R1$

   is step-consistent with Machine $M$ if $\text{init}(R1) \subseteq \text{next}(a?y?vle)$ and
Chapter 4. Adding I/O and next invariant to Annotations

R1 is step-consistent with M

2. \( R1 \sqcap R2 \)
   
is step-consistent with Machine M if \( R1 \) and \( R2 \) are step-consistent with M.

3. \( R1 \triangle R2 \)
   
is step-consistent with Machine M if \( R1 \) is step-consistent with M and
   
   if \( \forall op \in \text{init}(R2) \cdot \text{from} - \text{any}(op) = \text{true} \) and \( R2 \) is step-consistent

4. \( R1 \triangle_X R2 \)
   
is step-consistent with Machine M if \( R1 \) is step-consistent with M and
   
   \( \forall op \in \text{init}(R2) \cdot X \subseteq \text{from} - \text{set}(op) \) and \( R2 \) is step-consistent with M

5. \( R1 \triangle'_X R2 \)
   
is step-consistent with Machine M if \( R1 \) is step-consistent and \( R2 \) is step-consistent
   
   with M and \( \forall op \in \text{init}(R2) \cdot X \subseteq \text{from} - \text{set} - \text{init}(op) \)

6. \( S(p) \)
   
is step-consistent with Machine M

A family of recursive definitions \( S \triangleq R \) is step-consistent with M's annotations if each

\( R \) is step-consistent with M's annotations.

4.7. Proving Termination of Controlled Machines with I/O Annotations

The proofs in this section establish that it is enough to demonstrate annotation-consistency
(given machine-annotation consistency) to show that the every step of the machine-controller
pair terminates, which implies non-divergence: every operation is called within its precondition.
The proofs in this section extend section 3.1.6 by introducing new proofs for machines
with I/O annotations and controllers with I/O operations.

Demonstration termination of the controller by proof does not involve the annotation I/O.
This is the case because the POs for the annotation I/O are checked when the consistency is
established between the operations and the annotations. At this time it is proven that the
I/O sources for the annotations are type correct with the constraints of the preconditions. Hence the proofs to demonstrate termination of consistent controller paired with controlled machine are the same as in chapter 3.

The approach to the proof of operations with I/O is to ignore the outputs and substitute in the input into the precondition that is being established, and reuse the existing proofs for non-I/O actions. The substitution can be made because type correctness of the substitution has been proven in definitions 44, 55, 57, 59, when the POs are discharged. In the proof of mixed I/O and non-I/O operation traces the operation actions with I/O would be converted into operations without I/O by substituting in the input as above. Then the proof would proceed as though all the actions had non-I/O.

We restate the theorem for controllers with I/O:

**Theorem 4 (Annotated Consistent Controller are Divergence Free).** If an I/O annotated B Machine \( M \) is consistent with a controller \( M_{CTRL} \) then \( M \parallel M_{CTRL} \) is divergence free.

The proof that arbitrary length traces are non-divergent:

Case (A) \( \{ \} \)

1. \([T]I \) by consistency of \( M \)
2. \([T]true \) by \([T]I \Rightarrow [T]true \)

Case (B) \( \langle a,y_a,v_a,e_a \rangle \in \text{traces}(M_{CTRL}) \)
Chapter 4. Adding I/O and next invariant to Annotations

1. \((a,y_a,v_a,e_a) \in \text{traces}(M\_CTRL)\)
and \(M\_CTRL\) annotation consistent

2. \(a.y_a, v_a, e_a \in \text{next}(<\text{INITIALISATION}>) \lor \text{from} - \text{any} \rightarrow \text{op}(y \leftarrow a(v_a, e_a)) \lor \text{from} - \text{set} - \text{init} \rightarrow \text{op}(y \leftarrow a(v_a, e_a)) \neq \emptyset\)
definition of Lemma 10 below

3. \([T]P_a[V_a, E_a/v_a, e_a]\)
by 2 and def. 44, 55, 57, 59

4. \([T]P'_a\)
by 3 and after substitution and as machine-annotation consistent

5. \([T]I\)
Machine-Annotation consistency 4

6. \([T](P'_a \land I)\)
by 3 and 4 when I/O dropped

Entirely similar to the proof of theorem 1, Case (B), 6 to end

Case (C) \((tr \land (a.y_a,v_a,e_a, b.y_b,v_b,e_b)) \in \text{traces}(M\_CTRL)\) where \(tr \in \text{traces}(M\_CTRL)\)

1. \((tr \land a.y_a,v_a,e_a)\) is good
by inductive hypothesis

2. \(b.y_b,v_b,e_b \in \text{next}(y_b \leftarrow a(v_b, e_b)) \lor \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b, e_b)) \lor \text{from} - \text{set} - \text{init} - \text{op}(y_b \leftarrow b(v_b, e_b)) \lor \text{from} - \text{any} - \text{op}(b)\)
Lemma 11 below

3. \(I \land P_a \Rightarrow [B_a](P_b[V_b, E_b/v_b, e_b])\)
by 2 and def. 55 and typing correct

4. \(I \land P_b \Rightarrow [B_a]P'_b\)
Machine-Annotation consistency hence I/O dropped

Entirely similar to the proof of theorem 2, Case (C), 3 to 19

Singleton Traces are Enabled by Initialisation Annotations

Lemma 10 (Singleton traces of consistent I/O annotated CTRL are enabled by annotations). The actions of singleton traces of consistent controller \(R\) is also an initial-
4.8. Picturing I/O Annotations

A picturing annotation or an I/O annotation:

\[
\langle a.y.v.x \rangle \in \text{traces}(M_{-\text{CRTL}}) \Rightarrow \\
\begin{align*}
 a.y.v.x &\in \text{next}(\text{INITIALISATION}) \vee \\
 \text{from} - \text{any} - \text{op}(y \leftarrow a(v, x)) &\vee \\
 \text{from} - \text{set} - \text{init} - \text{op}(y \leftarrow a(v, x)) &\neq \{
\end{align*}
\]

The proof of lemma 10 follows exactly the format as the proof of lemma 4. The only difference in form is that the proof of lemma 10 has I/O. The I/O does not affect the outcome of the proof. The type correctness of the inputs used in the annotations is guaranteed during the determination of machine-annotation consistency.

Arbitrary Traces are Enabled by Annotations

Lemma 11 (Arbitrary Length Traces of Annotation-Consistent Interruptible Controllers are Enabled by Annotations). The actions of arbitrary length traces of a annotation consistent I/O controller fragments are enabled by operation annotations:

\[
R \text{annotation consistent} \land tr \cap \langle a.y.v.x \rangle \cap \langle b.y.v.x \rangle \in \text{traces}(R) \Rightarrow \\
b.y.v.x \in \text{next}(y_a \leftarrow a(v_a, e_a)) \vee \\
\text{from} - \text{any} - \text{op}(y_b \leftarrow b(v_b, e_b)) \vee \\
\begin{align*}
a.y.v.x &\in \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b, e_b)) \vee \\
a.y.v.x &\in \text{from} - \text{set} - \text{init} - \text{op}(y_b \leftarrow b(v, e_b))
\end{align*}
\]

The proof of lemma 11 as with lemma 10 follows exactly the same form as lemma 5. This is because the I/O plays no part in the proof.

4.8. Picturing I/O Annotations

Adding I/O to the existing FROM and NEXT annotations does not alter the drawings of the annotations without I/O, which is defined in chapters 2 and 3.

4.9. I/O NEXT Examples

An example that uses the NEXT annotation with I/O is given in Figure 4.7. The figure serves to illustrate how the annotations specify the way in which output from one operation feeds in to the next. The Store\_in variable is the interface to the environment, and is required in the implementation. More details about implementing external ports are given in chapter 6. The example is of a simple specification that has an operation that can sets a memory store.
MACHINE SetDifferent

VARIABLES Store, Store_in

INVARIANT Store ∈ N ∧ Store_in ∈ N ∧ Store_in ∈ 7..10

INITIALISATION Store_in ∈ 7..10 || Store := 5 /* { Set(!Store_in) } NEXT */

OPERATIONS

Set(xx) ≡ PRE xx ∈ 1..10 ∧ xx /= Store THEN Store := xx END
/* { Different } NEXT */;

mm ← Different ≡ PRE true THEN mm := ((1..10) - {Store }) END
/* { Set(mm) } NEXT */

END

Figure 4.7.: SetDifferent Machine

CTRL = SetDifferent_CTRL(?)

SetDifferent_CTRL(u) = Set(u) → Different?v → SetDifferent_CTRL(v)

Figure 4.8.: SetDifferent Controller

to a value and creates a value with the other operation. The value offered must not be the last one accepted.

The example is analysed step-by-step to demonstrate that the NEXT I/O annotations are consistent (step-consistent and initially-consistent) in the B machine.

Establishing Machine-annotation consistency for the SetDifference Example

1. Initialisation: the initialisation clause must establish the precondition of all the operations identified in its annotation; in this case this is Set(Store_in), with precondition
4.9. I/O NEXT Examples

(7..10 ⊆ 1..10 ∧ [Store := 6])/((xx ∈ 1..10 ∧ xx ≠ Store)[Store_in / xx])
≡
(Store_in ∈ 1..10 ∧ Store_in ≠ 6)
≡
(7 ∈ 1..10 ∧ 7/ = 6) ∧ (8 ∈ 1..10 ∧ 8/ = 6) ∧
(9 ∈ 1..10 ∧ 9/ = 6) ∧ (10 ∈ 1..10 ∧ 10/ = 6)
≡ true

2. Set(xx): the Set operation must satisfy the precondition of the operations identified in its annotation; in this case **Different**, with precondition true, which is obviously satisfied

3. mm ← Different: the operation must satisfy the precondition of the operations identified in its annotation; in this case Set(mm), with precondition xx ∈ 1..10 ∧ xx /= Store. From definition 45, we must prove:

I ∧ P_Difference ⇒ (E_Difference ⊆ T_eSet ⊆ [E_Difference]P_SET[E_Difference/ e_Difference])
≡ [mm :∈ (1..10) → {Store}](xx ∈ 1..10 ∧ xx /= Store)[mm/ xx]
≡ [mm :∈ (1..10) → {Store}]( mm ∈ 1..10 ∧ mm /= Store)
≡ true

Establishing Annotation-Controller Consistency for the SetDifference Example

Secondly, we demonstrate annotation-controller consistency. To show that the controller CMD is consistent with SetDifferent machine we apply the definitions of step-consistent and annotation consistent from definitions 68 and 69.

Step-consistency is established by considering the parts of the definition of CMD:

- **SetDifferent***_CTRL(p): The process variable SetDifferent***_CTRL(p) is step-consistent, by the definition of step-consistency for process variables.

- **Different?v → SetDifferent***_CTRL(v): This is first sub-process working back from the step-consistent definition SetDifferent***_CTRL(v). The prefix rule for step-consistency from definition 69 requires that:

  \[ \text{init}(\text{SetDifference***_CTRL(v)}) \subseteq \text{next} (\text{Different?v}) \]
  \[ \equiv \text{Set(v)}[v/u] \subseteq \{\text{Set(v)}\} \]
  \[ \equiv true \]

- **Set?v → Different?v → S***_CTRL(v): Different?v → S***_CTRL(v) is step-consistent and the prefix rule requires that:
Chapter 4. Adding I/O and \textsc{next invariant} to Annotations

\begin{align*}
\text{init}(\text{Different?} v \rightarrow S_{\text{CTRL}}(v)) & \subseteq \text{next}(\text{Set?} v) \\
\equiv \text{Different?} v & \subseteq \{ \text{Different.e} | e \in 1..10 \} \\
\equiv \text{true}
\end{align*}

initial-consistency is also established since:

\begin{align*}
\text{init}(\text{CTRL}) & \subseteq \text{next}(\text{INITIALISATION}) \\
\equiv \{ \text{Set.7} \} & \subseteq \{ \text{Set.7, Set.8, Set.9, Set.10} \} \\
\equiv \text{true}
\end{align*}

\section*{4.10. \textsc{next invariant} Annotations Example}

To illustrate the use of the \textsc{next invariant} an example of a heating system is introduced in Figure 4.9 and Figure 4.10. Variables that describes the state of the system are captured in the \textsc{next invariant}. The difference with the annotation approach is that the state of the variables can be carried around in the annotations and not in the CSP code. The operation invariants are only established when necessary. If the next invariants were promoted up to the machine invariant level then they would have to be established for every operation execution.

\subsection*{4.10.1. Establishing Machine-controller Consistency of the Heater Example}

\textbf{MACHINE} \textit{Heater} \\
\textbf{SETS} \textit{STATE} = \{ \textit{Stopped, Running} \} \\
\textbf{VARIABLES} \textit{Status, TargetTemp, CurrentTemp,} \\
\textit{TargetTempIn, CurrentTempIn} \\
\textbf{INARIANT} \textit{Status} \in \textit{STATE} \land \textit{TargetTemp} \in \mathbb{N} \land \textit{CurrentTemp} \in \mathbb{N} \land \textit{TargetTempIn} \in \mathbb{N} \land \textit{TargetTempIn} \in 14..28 \land \textit{CurrentTempIn} \in \mathbb{N} \\
\textbf{INITIALISATION} \textit{Status} := \textit{Stopped} \lor \textit{TargetTemp} \in 0..37 \lor \textit{CurrentTemp} \in \mathbb{N} \lor \textit{TargetTempIn} \in 14..28 \lor \textit{CurrentTempIn} \in \mathbb{N} /* \{ \textit{Start, Halt} \} \text{ \textsc{next} */}

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure4.9.png}
\caption{Heater Machine Header}
\end{figure}
OPERATIONS
Start  ≡  PRE  Status = Stopped  THEN  Status := Running  END
    /* { SetTargetTemp(?TargetTempIn) } NEXT */ ;

Halt  ≡  PRE  true  THEN  Status := Stopped  END
    /* FROM-ANY */
    /* { Start } NEXT */ ;

SetTargetTemp(xx)  ≡  PRE  xx ∈ 0..37  THEN  TargetTemp := xx  END
    /* NEXT INVARIANT { Status = Running } */
    /* { ReadCurrentTemp(?CurrentTempIn) } NEXT */ ;

ReadCurrentTemp(yy)  ≡  PRE  yy ∈ N  THEN  CurrentTemp := yy  END
    /* NEXT INVARIANT { Status = Running } */
    /* { SwitchHeater } NEXT */ ;

mm ← SwitchHeater  ≡  PRE  Status = Running  THEN  mm ∈ {0,1}  END
    /* { ReadCurrentTemp(?CurrentTempIn) } NEXT */

END

Figure 4.10.: Heater Machine Operations

The machine-controller consistency (machine-annotation consistency and annotation-controller consistency) of the Heater examples in figures 4.10 and figures 4.11 is established in what follows. Firstly, the machine-annotation consistency of Figure 4.9 and Figure 4.10 is established. Secondly, the annotation-controller consistency of Figure 4.11 is established.

Establishing Machine-annotation Consistency

- Initialisation: the initialisation clause must establish the precondition of all the operations identified in its annotation; in this case this is Start with precondition Status = stopped, and Halt with precondition which is true.

\[
[T_{initialisation}]P_{start} \land [T_{initialisation}]P_{halt} \\
≡ [Status := Stopped]Status = Stopped \\
≡ true
\]
Chapter 4. Adding I/O and next invariant to Annotations

\[ S_{CTRL} = \text{Start} \rightarrow \text{SetTargetTemp ? TargetTempIn} \rightarrow R \]
\[ R = \text{ReadCurrentTemp ? CurrentTempIn} \rightarrow \text{SwitchHeater ? xx} \rightarrow R \]
\[ CTRL = S_{CTRL} \triangle \text{halt} \rightarrow S_{CTRL} \]

Figure 4.11.: Heater Controller Operations

- **Start**: the Start operation must establish the precondition of the operations identified in its annotation; in this case \( \text{SetTargetTemp(TargetTempIn)} \), with precondition: \( xx \in 0..37 \). The SetTargetTemp operation also has a next invariant: \( \text{Status} = \text{running} \). The resulting PO to establish satisfaction of the precondition and the annotation operation next invariant is:

\[
I \land P_{\text{start}} \Rightarrow (T_{\text{start}} \subseteq T_{\text{SetTargetTemp}} \land \\
[B_{\text{Start}}(P_{\text{SetTargetTemp}} \land L_{\text{SetTargetTemp}})]T_{\text{start}} / e_{\text{SetTargetTemp}}) \\
\equiv [\text{Status} = \text{Running}][xx \in 0.37 \land \text{Status} = \text{Running}][\text{TargetTempIn} / xx] \\
\equiv \text{true}
\]

- **Halt**: the Halt operation must be able to follow any operation as a virtue of the FROM-ANY, and it must establish the precondition of Start due to the NEXT annotation. The precondition of Halt is true which ensures that it is possible to execute Halt after any operation. The Halt operation must establish the precondition of the Start operation with precondition \( \text{Status} = \text{stopped} \). It has to be proven that:

\[
I \Rightarrow [\text{Status} := \text{Stopped}](\text{Status} := \text{Stopped}) \\
\equiv \text{true}
\]

- **SetTargetTemp(xx)**: The two main predicates that have to be proven are that the NEXT INVARIANT re-establishes itself and that the precondition and next invariant of the next operation are established by the current operation. The NEXT INVARIANT is re-established as \( I \land P_{\text{SetTargetTemp}} \land L_{\text{SetTargetTemp}} \Rightarrow L_{\text{SetTargetTemp}} \) is trivially true. The SetTargetTemp operation must establish the precondition of ReadCurrentTemp as a result of the NEXT annotation. The precondition of ReadCurrentTemp is \( yy \in N \). The SetTargetTemp operation must also establish the annotated next invariant of \( \text{ReadCurrentTemp}(yy) L_{\text{ReadCurrentTemp}} \) which is: \( \text{Status} = \text{Running} \). However, we have at our disposal the next invariant \( L_{\text{SetTargetTemp}} \) which is the same as \( L_{\text{ReadCurrentTemp}} \). The variable used in the annotation to invoke \( \text{ReadCurrentTemp} \) replaces \( yy \) in the precondition of \( \text{ReadCurrentTemp} \). It has to be proven that:
The global Invariant $I$ and next invariant $L_{SetTargetTemp}$ were instrumental in discharging the POs above. Passing the operation invariant on through the operations $SetTargetTemp$ and $ReadCurrentTemp$ save setting up explicit flags in the state.

- **ReadCurrentTemp**(yy): The two main predicates that have to be proven are that the NEXT INARIANT re-establishes itself and that the precondition and next invariant of the next operation are established by the current operation. The NEXT INARIANT is re-established as $I \land P_{ReadCurrentTemp} \land L_{ReadCurrentTemp} \Rightarrow L_{ReadCurrentTemp}$ is trivially true. The ReadCurrentTemp operation must establish the precondition of SwitchHeater courtesy of the NEXT annotation. The precondition of SwitchHeater is $Status = Running$

$$\begin{align*}
(I \land P_{ReadCurrentTemp} \land L_{ReadCurrentTemp}) \Rightarrow [B_{ReadCurrentTemp}]P_{SwitchHeater} \\
\equiv (I \land yy \in N \land Status = Running) \Rightarrow \\
\quad [CurrentTemp := yy \mid (Status = Running)] \\
\equiv true
\end{align*}$$

- **SwitchHeater**: the switch heater operation has a precondition $Status = Running$ which is set by $Start$ and maintain by all subsequent operations following on from it to this operation. The operation outputs a 1 or a 0 to signify either success or failure, respectively. The annotation sends the flow of control back to $ReadTargetTemp$. A refinement of the operation would turn the heater on or off according to the temperature read into CurrentTemp.

$$\begin{align*}
(I \land P_{SwitchHeater}) \Rightarrow [B_{SwitchHeater}]\left(P_{ReadCurrentTemp} \land L_{ReadCurrentTemp}\right) \\
\equiv (I \land Status = Running) \Rightarrow \\
\quad [CurrentTemp := yy \mid (Status = Running)] \\
\equiv true
\end{align*}$$
Establishing Annotation-controller Consistency

Secondly, we demonstrate step-consistency.

Step-consistency is established by considering the parts of the definition of CTRL:

- The process variable $R$ is step-consistent, by the definition of step-consistency for process variables.
  - $\text{SetTargetTemp. TargetTempIn} \rightarrow R$: the prefix rule for step-consistency from Definition 69 requires that:
    \[
    \text{init}(R) \subseteq \text{next}(\text{SetTargetTemp. TargetTempIn}) \equiv \\
    \{\text{ReadCurrentTemp. CurrentTempIn} \} \subseteq \{\text{ReadCurrentTemp. CurrentTempIn} \},
    \]
    which is true, since this predicate is true and the process variable $R$ is step-consistent.
  - $\text{Start} \rightarrow \text{SetTargetTemp. TargetTempIn} \rightarrow R$: the prefix rule for step-consistency from Definition 69 requires that:
    \[
    \text{init}(\text{SetTargetTemp. TargetTempIn} \rightarrow R) \subseteq \text{next}(\text{Start}) \equiv \\
    \{\text{SetTargetTemp} \} \subseteq \{\text{SetTargetTemp} \},
    \]
    which is true, since this predicate is true.
  - $S_{-CTRL} \triangleq \text{Start} \rightarrow \text{SetTargetTemp. TargetTempIn} \rightarrow R$: the recursive definitions rule for step-consistency from Definition 69 requires that
    $\text{Start} \rightarrow \text{SetTargetTemp. TargetTempIn} \rightarrow R$ be step-consistent which it is true.

- The process variable $R$ is step-consistent, by the definition of step-consistency for process variables.
  - $\text{SwitchHeater.xx} \rightarrow R$: the prefix rule for step-consistency from Definition 69 requires that:
    \[
    \text{init}(R) \subseteq \text{next}(\text{SwitchHeater.xx}) \equiv \\
    \{\text{ReadCurrentTemp. CurrentTempIn} \} \subseteq \{\text{ReadCurrentTemp. CurrentTempIn} \},
    \]
    which is true, since this predicate is true and the process variable $R$ is step-consistent.
  - $\text{ReadCurrentTemp. CurrentTempIn} \rightarrow \text{SwitchHeater.xx} \rightarrow R$: the prefix rule for step-consistency from Definition 69 requires that:
    \[
    \text{init}(\text{SwitchHeater.xx} \rightarrow R) \subseteq \text{next}(\text{ReadCurrentTemp}) \equiv \\
    \{\text{SwitchHeater.xx} \} \subseteq \{\text{SwitchHeater.xx} \},
    \]
    which is true, since this predicate is true.
  - $R \equiv \text{ReadCurrentTemp. CurrentTempIn} \rightarrow \text{SwitchHeater.xx} \rightarrow R$: the recursive definitions rule for step-consistency from Definition 69 requires that
    $\text{ReadCurrentTemp. CurrentTempIn} \rightarrow \text{SwitchHeater.xx} \rightarrow R$ be step-consistent which it is true.
  - $\text{Halt} \rightarrow S_{-CTRL}$: the prefix rule for step-consistency from Definition 69 requires that:
    \[
    \text{init}(S_{-CTRL}) \subseteq \text{next}(\text{Halt}) \equiv \{\text{Start} \} \subseteq \{\text{Start} \},
    \]
    which is true, since the this predicate is true and the process variable $S_{-CTRL}$ is step-consistent.

- The process variable $CTRL$ is step-consistent, by the definition of step-consistency for process variables.
  - $S_{-CTRL} \triangleq \text{halt} \rightarrow S_{-CTRL}$: the prefix rule for step-consistency from Definition 69 requires that $S_{-CTRL}$ is initially consistent and $\text{halt} \rightarrow S_{-CTRL}$ is
step-consistent and from — any — op(halt) = true, which is all true, since
init(S_CTRL) ⊆ next(INITIALISATION) and from — any — op(halt) is true.
- CTRL = S_CTRL ∆ halt → S_CTRL the recursive definitions rule for step-

consistency from Definition 69 requires that S_CTRL ∆ halt → S_CTRL be step-

consistent which it is true.

The three recursive definitions of the step controller (CTRL, R, S_CTRL) have been shown
to be step-consistent, hence the complete controller is step-consistent.

initial-consistency is also established since:

• next(INITIALISATION) ⊆ init(CTRL): is initially-consistent as

{start, halt} ⊆ {start, halt}
5. Extending the NEXT Annotation

5.1. The NEXT Extensions

The motivation for extending the annotations at this point is to introduce branching behaviour. The first new annotation, the SELECT NEXT, is introduced to direct the flow of control in an implementation of the B and a controller. The annotation constrains the choices between different execution options. Like the NEXT annotation all options must be enabled, but unlike the next annotation the options are guarded by conditions that constrain whether they can be chosen or not in the implementation. The options of a NEXT annotation can be reduced by refinement, but the options of a SELECT NEXT annotation may not. The second annotation, the CONDITIONAL NEXT annotation requires additional controller language operators. The annotation constrains the way that choices are made in the controller as in the normal if-then-else approach. The SELECT NEXT resolves the choice using the values of B variables. The CONDITIONAL NEXT resolves the choices based on the output of B variables. In this chapter the shorthand used for sets of operations is given in definition 70.

Definition 70 (The Abbreviations for Operation Names).

The singleton operation $OP_i(v_i, e_i)$ is shortened to $OP_i$
The set of operations $\{OP_{j_1}, ..., OP_{j_m}\}$ is shortened to $OP_J$
similarly $\{OP_{k_1}, ..., OP_{k_n}\}$ is shortened to $OP_K$
where $v_i$ and $e_i$ are detailed in Figure 4.1, page 76.

5.2. The SELECT NEXT Annotations

The SELECT NEXT annotation is written:

\[
/* \{cond_{j_1} : OP_{j_1}, ..., cond_{j_{m-1}} : OP_{j_{m-1}} ELSE OP_{j_m}\} SELECT NEXT */
\]

Ignoring the conditions, the annotation is fundamentally a NEXT annotation. The conditions have no affect on the POs. They serve as directives to be utilised at implementation. The condition $cond_{j_0}$ is a predicate expression which uses B variables. The condition action pairs should be consulted in order from left to right to decide which action is to be executed.
next when coming to implement the specification. However, any action can be selected for execution next as they are all enabled as is the case with the NEXT annotation. If all the conditions are false then the final unconditional action is executed when implemented. No new controller syntax is introduced. However, unlike the pure NEXT annotation all the choices in the SELECT NEXT have to be reproduced in the controller. By basing the choice on variables in the B machine we avoid carrying around addition state in the controller CSP. Although it would be possible in terms of POs to have sets of operations paired with conditions we limit the pairing to one conditions and one operation. If sets were used it would not be possible to indicate which of the operations has the highest priority during implementation. The SELECT NEXT annotation is associated with a choice in the controller. In the implementation the B guards would be utilised to decide which option is taken (conditional choice is introduced in the implementation).

**Definition 71 (SELECT NEXT Annotation for Initialisation).**

\[
\text{INITIALISATION } T
\]  
\[
/ \ast \{ \text{cond}_{j_1} : \text{OP}_{j_1}, \ldots, \text{cond}_{j_{n-1}} : \text{OP}_{j_{n-1}}, \text{ELSE O P}_{j_m} \} \text{ SELECT NEXT } / \ast
\]

**Definition 72 (SELECT NEXT Annotation for Operations).**

\[
\text{OP}_i \equiv \{ P_i \mid B_i \} / \ast \{ \text{cond}_{j_1} : \text{OP}_{j_1}, \ldots, \text{cond}_{j_{n-1}} : \text{OP}_{j_{n-1}}, \text{ELSE O P}_{j_m} \} \text{ SELECT NEXT } / \ast ;
\]

In definition 71 the initialisation enables a number of operations that could be executed next. All the options are enabled. A choice between the options is added in the implementation. The first in the list is the preferred option. It is stated in definition 72 that after the execution of \( \text{OP}_i \) has completed, \( \text{OP}_{j_1}, \ldots, \text{OP}_{j_m} \) are always available to execute, even if their associated condition prevents them from being actually selected. The proof of this claim can be verified by discharging the proof obligations in definitions 73 and 74.

**Definition 73 (SELECT NEXT Proof Obligation for Initialisation).**

\[
\text{VARIABLES } V_1, \ldots, V_g
\]

\[
\text{IN V A R I A N T } V_1 \in T_{V_1} \land \ldots \land V_g \in T_{V_g}
\]

\[
\text{INITIALISATION } T
\]

\[
/ \ast \{ \text{cond}_{j_1} : \text{OP}_1(V_{j_1}, E_{j_1}), \ldots, \text{ELSE O P}_m(V_{j_m}, E_{j_m}) \} \text{ SELECT NEXT } / \ast
\]

\[
y_{j_1} \leftarrow \text{OP}_{j_1}(v_{j_1}, e_{j_1}) \equiv \text{PRE } P_{j_1} \text{ THEN } B_{j_1} \text{ END ;}
\]

\[
\ldots
\]

\[
y_{j_m} \leftarrow \text{OP}_m(v_{j_m}, e_{j_m}) \equiv \text{PRE } P_{j_m} \text{ THEN } B_{j_m} \text{ END ;}
\]

\[
\ldots
\]
The following PO arises:

\[ \forall x \in \{ O_{p_j1}, ..., O_{p_jm} \} \cdot (T_{AB} \land [T]((P_z \land L_z)[V_x, E_z/v_x, e_z]))) \]

where

\( e_z \) and \( v_x \) are free in \( T \), and

\( T_{AB} \) is defined in Figure 4.1

**Definition 74 (SELECT NEXT Proof Obligation for Operations).**

\[ y_j \leftarrow O_{p_j}(v_{j1}, e_{j1}) \equiv \text{PRE } P_i \text{ THEN } B_i \text{ END} \]

\[ /* \{ \text{cond}_j, ..., \text{OP}_j, \{ \text{OP}_j, \ldots \} \} \text{ SELECT NEXT } */ ; \]

\[ y_{j1} \leftarrow O_{p_j1}(v_{j11}, e_{j11}) \equiv \text{PRE } P_{j1} \text{ THEN } B_{j1} \text{ END} ; \]

\[ y_{j2} ... \]

\[ y_{jm} \leftarrow O_{p_jm}(v_{jm1}, e_{jm1}) \equiv \text{PRE } P_{jm} \text{ THEN } B_{jm} \text{ END} ; \]

The following PO arises:

\[ \forall x \in \{ O_{p_j1}, ..., O_{p_jm} \} \cdot (I \land P_i \land L_i \Rightarrow (T_{AB} \land [B_i]((P_z \land L_z)[V_x, E_z/v_x, e_z]))) \]

where

\( e_z \) and \( v_x \) are free in \( B \), and \( T_{AB} \) is defined in Figure 4.1

The proof obligations are the same as with the NEXT annotation. The PO generated from NEXT annotations in operations (definition 6, page 25) and in the initialisation (definition 8, page 26) ensure that all the operations listed in the annotation are enabled.

**Definition 75 (SELECT NEXT Annotation Listing).**

\[ \text{select}(OP_i) \equiv OP_j \]

\[ \text{if } OP_i ... /* \{ \text{cond}_j, ..., \text{OP}_j, \ldots \} \text{ SELECT NEXT } */ ; \]

\[ \equiv \text{undefined otherwise} \]
5.3. The CONDITION NEXT Annotations

In some situations the next operation in the execution may be in some way related to the outcome of the current operation: one operation selects another. The CONDITION NEXT annotation is an extension to the NEXT annotation that supports conditional next path selection based on the output of the current operation, written:

```
/* OPJ OPK CONDITION NEXT */
```

The annotation has two comma separated sets of operations: OPJ and OPK. The annotation sets do not have to be the same size. The first set is a set of operations that can be executed if output of the current operation is true or any natural number except 0. The second set is a set of operations that can be executed if the output of the current operation is false or 0. The operation that carries this annotation must have a single boolean or a single natural number output. The actual execution, if the output of the current operation represents true, is to select an operation from the first set of operations, otherwise an operation from the second set is selected. This annotation is the basis for conditional behaviour. Unlike the SELECT NEXT annotation it is not necessary to have a catch-all operation at the end of the annotation. The initialisation can not be annotated with the CONDITION NEXT annotation, because the initialisation has no output.

Definition 76 (CONDITION NEXT Annotation).

```
yi ← OPi ≜ (P1 | B1); /* OPJ OPK CONDITION NEXT */
```

In definition 76 above, if the output of the current operation represents true then all the operations OPj1 through to OPjm must be available for execute after the current operation has terminated. If however the current operation returns false then the operations OPk1 through to OPkn are always available to execute. The proof of this claim can be verified by discharging the following proof obligation:
5.4. Controller Syntax and Auxiliary Functions with condition NEXT Annotations

Definition 77 (condition NEXT Proof Obligation for Operations).

\[\begin{align*}
(I \land P_i \Rightarrow (T_{AB} \land [B_i]((y_i = \text{true}) \Rightarrow P_j[V, E/v_j, e_j]))) \land ... \\
&\ldots \\
&\land (I \land P_i \Rightarrow (T_{AB} \land [B_i]((y_i = \text{false}) \Rightarrow P_j[V, E/v_j, e_j]))) \\
&\ldots \\
&\land (I \land P_i \Rightarrow (T_{AB} \land [B_i]((y_i = \text{false}) \Rightarrow P_j[V, E/v_j, e_j]))) \\
\end{align*}\]

The controller is constrained in how to resolve the choice by the output of the current operation. In Figure 77 the PO applies \(B_i\), assigned to \(y_i\). The proof obligations differ from NEXT version in that an extra conditional element is added. We introduce a new controller operator to handle the requirements of the condition NEXT annotation.

5.4. Controller Syntax and Auxiliary Functions with condition NEXT Annotations

We extend the new listing function to support the new annotation and introduce a new function to deal with the condition annotations.

Definition 78 (Condition Annotation Listing).

\[\text{condition}(y_i \leftarrow \text{OP}_i / \star \text{OP}_j \text{OP}_k \text{CONDITION NEXT } \star /) = y_i\]

if \(y_i \in \text{BOOL}\) or \(y_i \in \mathbb{N}\)

undefined otherwise

Definition 79 (condition-true Annotation Listing).

\[\text{condition-true}(y_i \leftarrow \text{OP}_i \leftarrow \ldots / \star \text{OP}_j \text{OP}_k \text{CONDITION NEXT } \star /) = \text{OP}_j\]

if \(y_i \in \text{BOOL}\) or \(y_i \in \mathbb{N}\)

undefined otherwise

July 6, 2008
Chapter 5. Extending the NEXT Annotation

Definition 80 (condition_false Annotation Listing).

\[
\text{condition\_false}(y_i \leftarrow OP; \equiv \ldots */ OPJ OPK \text{ CONDITION NEXT */})
\]

\[
= \text{OPK}
\]

if \( y_i \in \text{BOOL} \) or

\( y_i \in \text{N} \)

undefined otherwise

Definition 81 (Trace Definition of traces(if condition then \( R_1 \) else \( R_2 \))).

\[
\text{traces}(\text{if} \ b \ \text{then} \ R_1 \ \text{else} \ R_2) = \begin{cases} 
\text{traces}(R_1) \ \text{if} \ b \\
\text{traces}(R_2) \ \text{if} \ \neg b
\end{cases}
\]

Definition 82 (Controller Syntax with Conditional Extension).

\[
R ::= \begin{cases} 
\text{a}\?y?x!x \rightarrow R(y) \\
R \sqcup R \\
R \triangle R \\
R \triangle_x R \\
R \triangle^1_x R \\
\text{if} \ y \ \text{then} \ R \ \text{else} \ R \\
S(p)
\end{cases}
\]

The controller language definition of 65 on page 91 is extended in definition 82 to take account of the addition of the if-then-else operator. The new operator introduced above is: if \( y \) then \( R_1 \) else \( R_2 \).

Definition 83 (init on CSP Controller Process with Conditional Choice).

\[
\begin{align*}
\text{init}(a?y?x!x & \rightarrow R_1) = \{a.y.v.x\} \\
\text{init}(R_1 \sqcup R_2) & = \text{init}(R_1) \cup \text{init}(R_2) \\
\text{init}(R_1 \triangle R_2) & = \text{init}(R_1) \cup \text{init}(R_2) \\
\text{init}(R_1 \triangle_x R_2) & = \text{init}(R_1) \\
\text{init}(R_1 \triangle^1_x R_2) & = \text{init}(R_1) \cup \text{init}(R_2) \\
\text{init}(\text{if} \ y \ \text{then} \ R_1 \ \text{else} \ R_2) & = \text{init}(R_1) \cup \text{init}(R_2) \\
\text{init}(S(p)) & = \text{init}(R(p)) \quad \text{where} \ S(p) \equiv R(p)
\end{align*}
\]
5.5. SELECT NEXT and CONDITION NEXT Annotation Consistency

Definition 83 extends definition 66, on page 92 with an *init* definition of the conditional CSP operator. The *guard* function is extended in a similar way.

**Definition 84 (guarded on CSP Controller Process with Global Interrupt).**

\[
\begin{align*}
\text{guarded}(a?x:y \rightarrow R1) & = \text{true} \\
\text{guarded}(R1 \sqcap R2) & = \text{guarded}(R1) \wedge \text{guarded}(R2) \\
\text{guarded}(R1 \triangledown R2) & = \text{guarded}(R1) \wedge \text{guarded}(R2) \\
\text{guarded}(R1 \triangle X R2) & = \text{guarded}(R1) \wedge \text{guarded}(R2) \\
\text{guarded}(\text{if } y \text{ then } R1 \text{ else } R2) & = \text{guarded}(R1) \wedge \text{guarded}(R2) \\
\text{guarded}(S(p)) & = \text{false} \\
\text{guarded}(S(p) \equiv R(p)) & = \text{true} \quad \text{if} \quad \text{guarded}(R(p)) = \text{true}
\end{align*}
\]

5.5. SELECT NEXT and CONDITION NEXT Annotation Consistency

As before the derivation of annotation consistency is broken down into annotation and step-consistency. The revised definitions of annotation consistency and step-consistency are stated in definition 85 and definition 86, respectively. The SELECT NEXT annotation does not introduce a new CSP operator into the controller language hence the definitions do not change as a result of that (the choice operator implements the SELECT NEXT annotation). The CONDITION NEXT is not permitted as an annotation in the initialisation: controllers with initial if-then-else operators are not initially-consistent. However designers will want to use the if-then-else operator in controllers regardless of whether the operator is supported by an annotation or not. An example is:

\[
\begin{align*}
R(y) & = \text{if } (y \mod 2) = 0 \text{ then } c?z \rightarrow R(z) \text{ else } d?z \rightarrow R(z) \ldots \\
M_{CTRL} & = R(0)
\end{align*}
\]

At initialisation the consistency of the if-then-else operator can be deduced directly from the context of the control fragment quite easily. In general the annotations make the deduction of consistency easier by guaranteeing different consistency requirements.

**Definition 85 (Annotation Consistent with Branching).**

1. \(a?x:y\!\!u!e \rightarrow R\)

is annotation consistent with Machine \(M\) if \(a?x:y\!\!u!e \rightarrow R\) is initially consistent and
Chapter 5. Extending the next Annotation

\[ a?y?v|e \rightarrow R \text{ is step-consistent with } M \]
\[ a?y?v|e \rightarrow R \text{ is initially-consistent if } a?y?v|e \in \text{next}(INITIALISATION) \]

2. \( R1 \sqcup R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) and \( R2 \) are annotation consistent with \( M \).

3. \( R1 \triangleleft R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) is annotation consistent and
\[ \forall op \in \text{init}(R2) \cdot from \cdot \text{any}(op) = \text{true} \] and \( R2 \) is step-consistent with \( M \).

4. \( R1 \triangleleft_X R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) is annotation consistent with \( M \) and
\[ \forall op \in \text{init}(R2) \cdot X \subseteq from \cdot \text{set}(op), \] and \( R2 \) is step-consistent with \( M \).

5. \( R1 \triangleleft_Y R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) is annotation consistent with \( M \) and
\[ \forall op \in \text{init}(R2) \cdot X \subseteq from \cdot \text{set}(op), \] and \( R2 \) is step-consistent with \( M \).

6. \( \text{if } b \text{ then } R1 \text{ else } R2 \)

is annotation consistent with Machine \( M \) if it is step-consistent.

7. \( S(p) \)

is annotation consistent with Machine \( M \)

A family of recursive definitions \( S(p) \triangleq R(p) \) is annotation consistent with \( M \)'s annotations if each \( R(p) \) is annotation consistent with \( M \)'s annotations.

Definition 86 (Annotation-controller Step-consistency with Branching). The step-consistency of the controller fragment \( R \) is defined as follows:

1. \( a?y?v|e \rightarrow R1 \)

is step-consistent with Machine \( M \) if \( \text{init}(R1) \subseteq \text{next}(a?y?v|e) \) and
\( R1 \) is step-consistent with \( M \).
5.6. Proving Termination of Controlled Machines with \textsc{condition next} Annotations

2. \(R_1 \square R_2\)

is step-consistent with Machine \(M\) if \(R_1\) and \(R_2\) are step-consistent with \(M\).

3. \(R_1 \vartriangle R_2\)

is step-consistent with Machine \(M\) if \(R_1\) is step-consistent with \(M\) and
\[
\forall \text{op} \in \text{init}(R_2) : \text{from} \rightarrow \text{any}(\text{op}) = \text{true} \quad \text{and} \quad R_2 \text{ is step-consistent}
\]

4. \(R_1 \vartriangle X R_2\)

is step-consistent with Machine \(M\) if \(R_1\) is step-consistent with \(M\) and
\[
\forall \text{op} \in \text{init}(R_2) : X \subseteq \text{from} \rightarrow \text{set}(\text{op}) \quad \text{and} \quad R_2 \text{ is step-consistent with } M
\]

5. \(R_1 \triangleleft X R_2\)

is step-consistent with Machine \(M\) if \(R_1\) is step-consistent and \(R_2\) is initially-consistent with \(M\) and \(\forall \text{op} \in \text{init}(R_2) : X \subseteq \text{from} \rightarrow \text{set} - \text{init}(\text{op}) T\)

6. \(\text{if } y \text{ then } R_1 \text{ else } R_2\)

is step-consistent with Machine \(M\) if \(y \in \text{BOOL}\) or \(y \in \text{N}\) and \(R_1\) and \(R_2\) are step-consistent with \(M\) and
\[
\forall b \in \text{init}(R_1) : b \in \text{condition}\_true(a.y.v.x) \quad \text{and} \quad
\forall c \in \text{init}(R_2) : c \in \text{condition}\_false(a.y.v.x)
\]

7. \(S(p)\)

A family of recursive definitions \(S \equiv R\) is step-consistent with Machine \(M\)'s annotations if each \(R\) is step-consistent with \(M\)'s annotations. In the above step-consistent definition the action \(a.y.v.x\) is used in the \text{condition}\_true and \text{condition}\_false functions. A one-to-one translation between action and operation is assumed.

5.6. Proving Termination of Controlled Machines with \textsc{condition next} Annotations

The proofs in this section establish that it is enough to demonstrate annotation-consistency (given machine-annotation consistency) to show that the every step of the machine-controller pair terminates, which implies non-divergence: every operation is called within its precondition. The proofs in this section extend section 4.7 by introducing new proofs for machines.
with CONDITION NEXT annotations and controllers with branching operations. There are no proofs associated with the SELECT NEXT annotation as it has the same POs as the NEXT annotation and is associated with the same controller fragments as the NEXT annotations.

The proof of theorem 2 is repeated, but this time with if-then-else in the controller language. It must be shown that only good traces are produced by M_CTRL. Hence M || M_CTRL is divergence free. The reasoning is done over the lengths of the traces in the proof.

We restate the theorem for controllers for CONDITION NEXT:

Theorem 5 (Annotated Machine Controller Combinations that are Machine-Annotation Consistent and Annotation Consistent are Divergence Free). If a CONDITION NEXT annotated B Machine M is annotation consistent with a controller M_CTRL is annotation consistent then M || M_CTRL is divergence free.

The proof that arbitrary length traces are non-divergent (I/O not considered when referencing existing proof work):

Case (A) () The empty trace
1 \[T\]I by consistency of M
2 \[T\]true by \[T\]I \(\Rightarrow\) \[T\]true

Case (B) \((a.y_a.v_a.x_a) \in \text{traces}(M_{-CTRL})\) The singleton trace
1 \((a.y_a.v_a.x_a) \in \text{traces}(M_{-CTRL})\) and initial assumption
2 \(a.y_a.v_a.x_a \in \text{next(INITIALISATION)} \lor \text{from-any-op}(y_a \leftarrow a(v_a, e_a)) \lor \text{from-set-init}(y_a \leftarrow a(v_a, e_a)) \neq \{\}\) def. of Lemma 4, on page 51
3 \[T\]Pa by def. of PO of annotations in all cases
4 \[T\]I by def. of machine-consistency 4
5 \[T\][P_a \land I] by 3 and 4

Entirely similar to the proof of theorem 1, page 30, Case B lines 6 to end.

Case (C) \(tr \cap (a.y_a.v_a.e_a, b.y_b.v_b.e_b) \in \text{traces}(M_{-CTRL})\) where \(tr \in \text{traces}(M_{-CTRL})\)

The arbitrary trace...
5.6. Proving Termination of Controlled Machines with CONDITION NEXT Annotations

1 \( (r \cap a.y.a.v.a.e.a) \) is good 
by inductive hypothesis

Case (C1) The existing annotations

2 \( b.y.b.v.b.e.b \in \text{next}(a.y.a.v.a.e.a) \lor \)
def. of lemma 12, on page 118
from-any-op(b.y.b.v.b.e.b) \lor
\( a.y.a.v.a.e.a \in \text{from-set-op}(b.y.b.v.b.e.b) \lor \)
def. of lemma 12, on page 118
\( a.y.a.v.a.e.a \in \text{from-set-init-op}(b.y.b.v.b.e.b) \lor \)
Entirely similar to the proof of theorem 4, page 95, Case (C) lines 3 to end.

case (C2) condition_true\( (a.y.a.v.a.e.a) \neq \{\} \lor \)
condition_false\( (a.y.a.v.a.e.a) \neq \{\} \)
The new condition annotations

case (C2a) The true condition

1 \( condition(a.y.a.v.a.e.a) = true \)
def. of condition annotation listing 78

2 \( y.a = true \)

3 \( P.a \land I \Rightarrow [B.a](y.a \Rightarrow P.c) \land \)
case 2 and def. 77 condition next
\( P.a \land I \Rightarrow [B.a](-y.a \Rightarrow P.d) \)
POs with I/O dropped, on page 110

4 \( P.a \land I \Rightarrow [B.a](P.c) \)
Entirely Similar to
the proof of theorem 1, page 30,
case C lines 5 - 18.

5 \( c = b \)
let arbitrary \( c \) equal \( b \)

6 \( (r \cap a \cap b) \) is good 
by 18, 19 and
def. of machine consistency

July 6, 2008
Chapter 5. Extending the NEXT Annotation

Case (C2b)  The false condition -
1 \[ \text{condition}(a.y_a,v_a,e_a) = false \]
2 \[ \neg y_a = false \]
3 \[ P_a \land I \Rightarrow [B_a](y_a \Rightarrow P_c) \land \]
\[ P_a \land I \Rightarrow [B_a](\neg y_a \Rightarrow P_d) \quad \text{case 2 and def. of PO 77} \]
4 \[ P_a \land I \Rightarrow [B_a](P_d) \quad 2 \text{ and } 3 \]

Entirely similar to
the proof of theorem 1, page 30,
case C lines 5 - 18.

\[ d = b \]
\[ \text{let arbitrary } d \text{ equal } b \]

\[ \langle tr \leftarrow a \leftarrow b \rangle \text{ is good by 18 and 19 and} \]
\[ \text{def. of machine consistency} \]

Case C is a new case that introduces the conditional operator, which has a true and false branch.

Arbitrary Traces are Enabled by Annotations

Lemma 12 (Arbitrary Length Traces of Annotation Consistent Controllers are Enabled by Annotations). The actions of arbitrary length traces of annotation-controller consistent controllers with branching fragments are enabled by operation annotations:

\[
\begin{align*}
\text{annotation} - \text{controller consistent} \land \langle tr \leftarrow (a.y_a,v_a,e_a) \leftarrow (b.y_b,v_b,e_b) \rangle \in \text{traces}(R) \Rightarrow \\
b.y_b.v_b.e_b \in \text{next}(y_a \leftarrow a(v_a,e_a)) \lor \\
\text{from} - \text{any} - \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \\
a.y.v.x \in \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \\
a.y.v.x \in \text{from} - \text{set} - \text{init} - \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \\
(\text{condition}(y_a \leftarrow a(v_a,e_a)) \land \\
b?y_b?v_ble_b \in \text{condition_true}(y_a \leftarrow a(v_a,e_a)) \lor \\
(\text{condition}(y_a \leftarrow a(v_a,e_a)) \land \\
b?y_b?v_ble_b \in \text{condition_false}(y_a \leftarrow a(v_a,e_a)))
\end{align*}
\]

Proof of Lemma 12 by cases of controllers ignoring I/O which has been established to be type correct by machine-annotation consistency.

Case A  \[ c?y_a?v_cle_c \rightarrow R \]

Entirely similar to the proof of Lemma 5, page 55, case A, using lemma 13, page 120, instead of lemma 6, page 60.
5.6. Proving Termination of Controlled Machines with \textsc{condition next} Annotations

case B \hspace{1em} R1 \square R2
Entirely similar to the proof of Lemma 5, case B.

case C \hspace{1em} R1 \bigtriangleup R2
Entirely similar to the proof of Lemma 5, case C.

case D \hspace{1em} (R1 \bigtriangleup_x R2)
Entirely similar to the proof of Lemma 5, case D.

case E \hspace{1em} (R1 \bigtriangleup_y R2)
Entirely similar to the proof of Lemma 5, case E.

case F \hspace{1em} S(p)
Entirely similar to the proof of Lemma 5, case F.
Chapter 5. Extending the NEXT Annotation

case G if condition then R1 else R2

1 \[ \text{tr} \uparrow (a.y_a,v_a,e_a) \cap (b.y_b,v_b,e_b) \in \text{traces}(\text{if } y \text{ then } R_1 \text{ else } R_2) \] initial assumption

2 if \( y \) then \( R_1 \) else \( R_2 \) step - consistent with \( M \) initial assumption

3 \( R_1 \) step-consistent with \( M \) 2, def. consistency 86

4 \( R_2 \) step-consistent with \( M \) 2, def. consistency 86

case 1 \( y = true \) or \( y \neq 0 \)

5 \[ \text{tr} \uparrow (a.y_a,v_a,e_a) \cap (b.y_b,v_b,e_b) \in \text{traces}(R_1) \] by 3 and \((y \neq 0 \text{ or } y = true)\)

6 \[ b \uparrow y \uparrow v_0 \mid e_0 \in \text{next}(y_a \leftarrow a(v_a,e_a)) \lor \] from - any \( \neg \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \)

\[ a.y.v.x \in \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \] a.y.v.x \( \in \) from - set - init \( \neg \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \)

\[ \text{(condition}(y_a \leftarrow a(v_a,e_a)) \lor \] condition_true \( (y_a \leftarrow a(v_a,e_a)) \neq \{\} \lor \)

\[ \text{(condition}(y_a \leftarrow a(v_a,e_a)) \lor \] condition_false \( (y_a \leftarrow a(v_a,e_a)) \neq \{\} \) 3, 5, and ind. hyp. on \( R_1 \)

QED

case 2 \( y = false \) or \( y = 0 \)

7 \[ \text{tr} \uparrow (a.y_a,v_a,e_a) \cap (b.y_b,v_b,e_b) \in \text{traces}(R_2) \] by 4 and \((0 \text{ or } y = false)\)

8 \[ b \uparrow y \uparrow v_0 \mid e_0 \in \text{next}(y_a \leftarrow a(v_a,e_a)) \lor \] from - any \( \neg \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \)

\[ a.y.v.x \in \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \] a.y.v.x \( \in \) from - set - init \( \neg \text{op}(y_b \leftarrow b(v_b,e_b)) \lor \)

\[ \text{(condition}(y_a \leftarrow a(v_a,e_a)) \lor \] condition_true \( (y_a \leftarrow a(v_a,e_a)) \neq \{\} \lor \)

\[ \text{(condition}(y_a \leftarrow a(v_a,e_a)) \lor \] condition_false \( (y_a \leftarrow a(v_a,e_a)) \neq \{\} \) 4, 7, and ind. hyp. on \( R_2 \)

QED

The Initial Action of a Consistent Controller

Lemma 13 (Elements of initially-consistent singleton traces are in init(\( R \))). An element of the singleton trace of an annotation-controller consistent controller fragment \( R \) is contained in the init of \( R \):

\[ (b.y_v,v_b,e_b) \in \text{traces}(R) \Rightarrow b.y_v,v_b \in \text{init}(R) \]
Proof of Lemma 13 follows the same pattern as the proof in Lemma 6, on page 60, by cases of controllers ignoring I/O which has already been proven to be machine-annotation consistent:

- **case A** \( (c ? y_c ? v_c ! e_c \rightarrow R) \)
  Entirely similar to the proof of Lemma 6, case A, page 60.

- **case B** \( (R1 \sqcup R2) \)
  Entirely similar to the proof of Lemma 6, case B.

- **case C** \( (R1 \vartriangle R2) \)
  Entirely similar to the proof of Lemma 6, case C.

- **case D** \( (R1 \vartriangle_X R2) \)
  Entirely similar to the proof of Lemma 6, case D.

- **case E** \( (R1 \vartriangle^i_X R2) \)
  Entirely similar to the proof of Lemma 6, case E.

- **case F** \( (\delta(p)) \)
  Entirely similar to the proof of Lemma 6, case F.

- **case G** \( (\text{if } y \text{ then } R1 \text{ else } R2) \)

  \[ 1 \quad (b) \in \text{trace}(\text{if } y \text{ then } R1 \text{ else } R2) \]
  Entirely similar to the proof of Lemma 6, case B, lines 2 to 6.

### 5.7. Examples Utilising \texttt{SELECT NEXT} and \texttt{CONDITION NEXT}

The examples in Figures 5.1, 5.2, 5.3, and 5.4 illustrate the use of the \texttt{SELECT NEXT} annotation and \texttt{CONDITIONAL NEXT} annotation. The machine is introduced in Figures 5.1, Figures 5.2
Chapter 5. Extending the NEXT Annotation

and 5.3. The first of the three figures introduces the declarative parts and the second figure introduces operations: `create_store_ok`, `enter_code_ok`, `read_store_ok`, `drive_switch_ok` and `safe_store`. The second part depicts mostly state manipulating operations. The `safe_store` operation is a state modifying operation as in `enter_code`, `read_store` and `drive_switch`. The annotations of the figures use the `M(skey)` definition, which represents a set of operations parameterised with the current `skey`. Very little detail is given in the operations. Generally, the operation bodies produce new values for state non-deterministically or the bodies are just `skip`. The machine just gives a flavour of the use of the annotations. A controller for the machine is given in Figure 5.4.

The safety switch may be associated with several sets of saved codes. Each set is referred to by a unique key. Codes are driven through the safety switch to open it. The codes are loaded one-by-one into a memory with the `enter_code` command to an appropriate set associated with the current key. The codes can be driven through the switch by outputting them one at a time from the store referenced by a key using the `drive_switch` command. The saved stored codes can be overridden with a predefined safe pattern using the `safe` command. The codes can be inspected by driving them out through a port other than the drive port one-by-one with the `read_store` command. New storage is allocated with the `create_store` command. State querying actions are available to test the state before committing to a state modifying action. Type preconditions are defined in the operations signatures.

The B machine introduces, through the machine interface, three unique codes and a `CODE` set type. The length of a code store is set by the machine parameter `store_len`, which must be greater than 0. The number of codes in the given `CODE` set must be greater than 3. The set `SKEY` is set to contain three elements: `uqs1`, `uqs2`, and `uqs3`. A number of variables are introduced in the `VARIABLES` clause. The variables `skey_set` and `store`, are a power set of `SKEY` and a sequence of codes indexed by an element from the `SKEY` type. The variables `vSKEY`, `vCODE` and `vPOS` are used by the annotations to get new values from the environment. There are other variables but they are used for internal storage. The idea here is to introduce variables that will act as ports to the outside world when translation to a hardware description language.

The first four operations are query type operations. They are not marked with query annotations, because they are used as part of an annotation branch. Hence they are marked with condition next annotations. The bodies of the operations do not give any details of how the operations are implemented. The detail would be introduced during refinement. The conditional next annotation of the `create_store_ok` branches to either the `create_store` operation next or the collection of operations defined by `M(skey)`. The definition shorthand `M(skey)` is introduced as a shorthand to describe the collection of operations: `create_store(ok), enter_code_ok(skey), read_store_ok(skey), drive_switch_ok` and `safe(skey)`.

The `enter_code_ok` operation has a possible branch to the `enter_code` operations conditional on whether the argument `skey` is a member of the `skey_set`. The other `vCODE` is not known at this point in the specification, and is introduced by allowing it to take any value that the variable of the same name can accept. In Figure 5.4 a controller for the machine is given. The controller is developed systematically from the B annotations. Each operation becomes a process with an initial action of the same name. Like `enter_code_ok`, the operations `read_store_ok` and `drive_switch_ok` have annotations that are conditional and take input from the environment. The `safe_store` sets the store of the `skey` passed in to a predefined

122 July 6, 2008
5.7. Examples Utilising SELECT NEXT and CONDITION NEXT

pattern. The create_store operation creates a new skey and code store. A select annotation is used to control the branching after the create_store, read_store and drive_switch operations. The read_store operation creates a code and a position to store the code at. The operation design suggests that a loop could be used to read out all the values from a particular store. The drive_switch_op annotation like the read_store operation is suggestive of a possible loop implementation as well.

MACHINE SSM(aa, bb, xx, CODE, store_len)

CONSTRAINTS

\[ CODE \in \mathbb{P}(N) \land \text{card}(CODE) \geq 3 \land aa \in CODE \land bb \in CODE \land \]
\[ xx \in CODE \land store\_len \in N \land store\_len > 0 \land \text{card}\{\{aa, bb, xx\}\} = 3 \]

SETS SKEY = \{uqs1, uqs2, uqs3\}

VARIABLES skey_set, store, vSKEY, vCODE, vPOS

INVARIANT skey_set \in \mathbb{P}(SKEY) \land store \in \mathbb{P}(SKEY \to \text{seq}(CODE)) \land

\[ vSKEY \in SKEY \land vCODE \in CODE \land vPOS \in 1..\text{store\_len} \]

INITIALISATION

\[ skey\_set = \{} || \text{store} = \{} || vSKEY :\in SKEY || vCODE :\in CODE || \]
\[ vPOS :\in 1..\text{store\_len} / \ast \{\text{create\_store}\} \text{NEXT} \ast / \]

Figure 5.1.: Safety Switch Machine - Header

The controller in Figure 5.4 is derived directly from the B specification in Figure 5.3. An effort has been made to synthesise that CSP from the annotations to illustrate standard translation templates. The synthesis guidelines are captured in the table in Figure 5.5, page 127.

Annotations must refer to operation I/O in a consistent way. If one annotation refers to operation \( op \) with input fixed by the environment then all annotations must refer to that operation in the same way. The B initialisation is translated into CREATE_Store process. The CREATE_Store initialisation is invoked from a process called CTRL. The create_store_ok operation translates to a CSP if-then-else, as does the enter_code_ok; the read_store_ok and the drive_switch_ok operations. Most of the operations have skey as a parameter. The need to retain the skey value means it is a process variable. The drive_switch_ok B operation inputs vPOS and vSKEY values and passed on as process variables. The create_store operation has a select annotation, therefore translates with an external choice next.
Chapter 5. Extending the NEXT Annotation

OPERATIONS /* See defs. clause at the end of machine for details of \( M(skey) \). */
/* The checking operations used by the environment are defined first: create_ok, */
/* enter_code_ok, read_store_ok, drive_switch_ok. The safe_store operation is not a */
/* checking operation. All these operations can be called by the environment at any time */
/* when they are enabled by the controller. */
/*
/* CREATE_OK — The creation of a store for the next stream of codes */
/* On failure the annotations restrict the alternative to driving */
bb ← create_store_ok ≡ PRE true THEN bb := bool(skey_set ≠ SKEY) END
/* {create_store} \( M(skey) \) CONDITION NEXT */;

/* ENTER_CODE_OK — */
/* Establishes if a code can be entered into a specific code store */
/* On failure the annotations permit any operation to */
/* be carried out on the current store */
bb ← enter_code_ok(skey) ≡
PRE skey ∈ SKEY THEN bb := bool(skey ∈ skey_set) END
/* {enter_code(?vCODE!skey)} \( M(skey) \) CONDITION NEXT */;

/* READ_STORE_OK — */
/* Tests that store can be read. */
bb ← read_store_ok(skey) ≡ PRE skey ∈ SKEY THEN bb ∈ BOOL END
/* {read_store(?vPOS!skey)} \( M(skey) \) CONDITION NEXT */;

/* DRIVE_SWITCH_OK — */
/* Tests whether it is safe to output the contents of the stores */
/* A failure to drive does not prevent current store operations being carried out */
bb ← drive_switch_ok ≡
PRE true THEN bb ∈ BOOL || vPOS ∈ 1..store_len || vSKEY ∈ skey_set END
/* {drive_switch(?vSKEY?vPOS)} \( M(skey) \) CONDITION NEXT */;

/* SAFE_STORE */
/* Overwrites the contents of a store with a safe pattern */
safe_store(skey) ≡
PRE skey ∈ SKEY THEN store(skey) := (1..store_len) * \{ aa \} END
/* \( M(skey) \) NEXT */;

Figure 5.2.: Safety Switch Machine - Environmental Operations
/* The controller can call create, enter_code, drive_switch, only when their */
/* preconditions are met and there are next in the control sequence */

/* Create safe key and safe pattern */
skey ← create_store ≡ PRE SKEY → skey_set ≠ {} THEN
  ANY ss , pp WHERE
    ss : SKEY → skey_set ∧
    pp : seq(CODE) ∧ size(pp) = store_len
  THEN
    skey := ss∥ vSKEY := ss∥
    skey_set := skey_set ∪ {ss}∥ store := store ∪ {ss ↔ pp}
  END
END / * {true : safe_store(skey),*/
/* true : enter_code_ok(skey),*/
/* ELSE create_store_ok } SELECT NEXT */;

/* No operation body defined in abstraction */
enter_code(code, skey) ≡ PRE code ∈ CODE ∧ skey ∈ skey_set THEN skip END
/* M(!skey) NEXT */;

/* Value given to vPOS to exercise annotation. */
code ← read_store(skey) ≡
PRE skey ∈ SKEY THEN code := { aa, bb, xx }∥ vPOS := 1..store_len END */
/* { code = xx : safe_store(skey),*/
/* vPOS + 1 ≤ store_len : read_store(skey), */
/* ELSE create_store_ok } SELECT NEXT */;

code ← drive_switch(pp, skey) ≡
PRE pp ∈ N ∧ skey ∈ SKEY ∧ skey ∈ skey_set ∧ pp ∈ 1..store_len THEN
  code := (CODE = xx)∥ vPOS := 1..store_len || vSKEY := skey_set
END /* { vPOS ≤ store_len : drive_switch(?vPOS?vSKEY),*/
/* ELSE create_store_ok } SELECT NEXT */;

DEFINITIONS
M(!skey) == { create_store_ok, enter_code_ok(!skey),
              read_store_ok(!skey), drive_switch_ok, safe_store(!skey)}

END

Figure 5.3.: Safety Switch Machine - Controller Operations
CTRL = CREATE_STORE

CREATE_STORE_OK(skey) = create_store_ok?bb
if bb then CREATE_STORE else M(skey) ——/ * CONDITION NEXT */

ENTER_CODE_OK(skey) = enter_code_ok?bb!skey
if bb then ENTER_CODE(skey) else M(skey) ——/ * CONDITION NEXT */

READ_STORE_OK(skey) = read_store_ok?bb!skey
if bb then READ_STORE(skey) else M(skey) ——/ * CONDITION NEXT */

DRIVE_SWITCH_OK(skey) = drive_switch_ok?bb
if bb then DRIVE_SWITCH else M(skey) ——

SAFE_STORE(skey) = safe_store!skey → M(skey)

CREATE_STORE = create_store?skey →
   (SAFE_STORE(skey) —— SELECT true
   □ ENTER_CODE_OK(skey) —— SELECT true
   □ CREATE_STORE_OK)
      —— SELECT ELSE

ENTER_CODE(skey) = enter_code?vCODE!skey → M(skey) ——NEXT

READ_STORE(skey) = read_store?code!skey →
   (SAFE_STORE(skey) —— SELECT code = xx
   □ READ_STORE(skey) —— SELECT vPOS + 1 ≤ store_len
   □ CREATE_STORE_OK)
      —— SELECT ELSE

DRIVE_SWITCH = drive_switch?vCODE?vPOS?vSKEY →
   DRIVE_SWITCH —— SELECT vPOS ≤ store_len

M(skey) = (CREATE_STORE_OK
   □ ENTER_CODE_OK(skey)
   □ READ_STORE_OK(skey)
   □ DRIVE_SWITCH_OK(skey)
   □ SAFE_STORE(skey))

Figure 5.4.: Safety Switch Machine Controller Operations
### 5.7. Examples Utilising `SELECT NEXT` and `CONDITION NEXT`

<table>
<thead>
<tr>
<th>B</th>
<th>CSP</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Operation name</strong></td>
<td>A process with the same name in capitals</td>
</tr>
<tr>
<td><strong>Operation I/O</strong></td>
<td>A channel with the same name in lower case</td>
</tr>
<tr>
<td><strong>Annotation operation parameters</strong></td>
<td>Process variable and channel output</td>
</tr>
<tr>
<td>set in previous operation</td>
<td></td>
</tr>
<tr>
<td><strong>Annotation operation parameters</strong></td>
<td>channel input</td>
</tr>
<tr>
<td><em>not set in previous operation</em></td>
<td></td>
</tr>
<tr>
<td><strong>Operation parameters set by</strong></td>
<td>channel input</td>
</tr>
<tr>
<td>environment</td>
<td></td>
</tr>
<tr>
<td><strong>Operation output</strong></td>
<td>channel input</td>
</tr>
<tr>
<td><strong>CONDITION NEXT annotations</strong></td>
<td>if-then-else branch to other processes</td>
</tr>
<tr>
<td><strong>NEXT annotations</strong></td>
<td>Prefixed action the process representing the next operation</td>
</tr>
<tr>
<td><strong>SELECT NEXT annotations</strong></td>
<td>Prefixed action to external choice branch of other processes (the guards are not utilised in the CSP)</td>
</tr>
</tbody>
</table>

Figure 5.5.: Annotated B to CSP Synthesis Guide
Chapter 5. Extending the NEXT Annotation.
6. Translating and Refining Annotations

The motivation for this chapter is to discuss how the annotations can be extended to deal with implementation and refinement of critical controller specifications. This chapter examines the refinement and translation of B annotated machines and their controllers, and follows a similar pattern to other technical chapters. The new annotations are introduced. A definition of consistency is offered along with proof that shows that annotation-consistent controllers are divergent free. At the end of the chapter the refinements and translations to hardware are introduced.

Only translations to Handel-C are considered. To take advantage of the Handel-C hardware description language a new annotation is introduced, INTERLEAVED NEXT. It generates proof obligations that test whether stated operations can be executed concurrently. This annotation allows a developer to take advantage of the parallelism available in hardware. The SEQUENCE NEXT annotation is introduced to assist refinement. This annotation allows a developer to state control refinements. In general annotations are added to suit the different application being model and source languages targeted. Importantly, the underlying B is not changed.

The chapter introduces one approach to annotated interleaved execution. It is termed weak annotation interleaving: all of the operations available for interleaving might not occur before the interleaved operation is said to be complete (some actions in the interleaved annotation set might not be executed). The strength of the annotation interleaving is dictated by the precondition of the operation following the interleaving. The measure of strength of the interleaving relates to how the operation following the interleaved operations is enabled. Weak annotation interleaving has a next operation that any one of the operations of the interleaving can enable individually (any of the interleaved operations are sufficient to enable the next operation). In strong interleaving each operation of the interleaving must add towards the enablement of the next operation (all the interleaved operations are necessary to enable the following operation). Strong annotation interleaving is not considered in-depth in this thesis. However, the goal is to construct specifications that utilise annotations that use strong interleaving in the future. Weak interleaving is considered here first as a basis for strong interleaving. Strong interleaving has the same basic requirements as weak interleaving. The only difference is that each interleaved operations adds towards establishing the precondition of the next operation. Like weak interleaving strong interleaving requires that the interleaved operations have independent variables. Similarly, the next operation is the point at which the interleaving converges.

When considering the proof of the termination of traces of operations with interleaved annotations, later in the chapter, it will be necessary to identify which operation two consecutive interleaved operations relate to. The FROM INTERLEAVED annotation is introduced to keep a track of where interleaved operations flow from. Two consecutive operations a and b in a
trace $tr$ are generated by a consistent machine-controller pair if $a$ enables $b$ or if $a$ and $b$ are enabled by another operation $z$ and $a$ does not disable $b$. We only consider the simplest form of interleaving in this chapter. To keep things at their most basic the interleaved behaviour is brought to an end quickly.

A different I/O model is utilised to that of chapter 4, where I/O is produced by three separate interacting subsystems: the environment, the B model and the CSP controller. The I/O model is now simplified to ease implementation for the purposes of this chapter. The control elements of examples given in this thesis are primarily sequential. Parallelism is only introduced to aid data processing, and only considers a subset of the I/O detailed in chapter 4. The new view of the interaction between environment and the execution of the annotated B specification is given in diagram 6.1. The system can be viewed in two ways: as a CSP model or as an annotated B model. The CSP controller can be thought of as a view of the final implementation that details only control aspects, whereas the pure B is a view of the system that details data manipulations. The annotated B contains both views. In this chapter the focus is on the implementation aspects of the research. Simple controllers are considered, and as no data is transferred between controller and B annotations the I/O signature is restricted to the following: $y_i \leftarrow Op_i(n_i)$. The $e_i$ input from the controller has been dropped in this chapter.

![Diagram 6.1: Different Views of the Same Action.](image)

Let us re-cap the definition of the basic annotations. We annotate operations of a B machine with a NEXT annotation that supports operations with I/O. If the conjunction of PO's for all the annotations are discharged then we say that the annotations are consistent with the machine: machine-annotation consistent. A consistent controller that evolves in accordance with the next annotations steps will not diverge or deadlock. Stepping through the controller showing that annotations support the controller execution demonstrates annotation-controller consistency. If a machine-controller pair is both machine-annotation and annotation-controller consistent then the pair when executed together will not diverge. Consider the NEXT anno-
A NEXT annotation on the current operation $OP_i$ (where $OP_i$ represents $y_i \leftarrow O_i(v_i)$ and $y_i$ is the output vector, $y_1 \ldots y_n$, and $v_i$ is the input parameter vector, $v_1 \ldots v_m$) introduces another operation $OP_j$, or set of operations $\{OP_j, \ldots, OP_{j_n}\}$, which will be enabled after $OP_i$ is executed (where an operation in the annotation $OP_j$ represents $O_j(v_j)$ and $v_j$ is the input expression vector, $v_1 \ldots v_m$). In the NEXT annotation $v_j$ is a list of expressions which serves as inputs on which $OP_j$ can be called next. In this chapter annotation input expression list will be restricted to variables $v$ defined in the B machines, which will supply inputs in the hardware implementation.

### 6.0.1. A Refined B Machine

A refinement of the B machine introduced in Section 4.1 in Figure 4.1 is given in Figure 6.2. This refinement produces a subset of the refinement POs defined in [Abr96] and [Sch01]. The MACHINE keyword is replaced with REFINEMENT, but the machine that is refined is given in the REFINES clause. New variables, invariant and initializations are added. The invariant property $J$ plays a key role in defining the correctness of the refinement. It introduces a predicate that relates the state of the machine and the state of the refinement. The refinement proof obligations considered in this thesis are detailed in definition 87. The first PO (PO 1) states that:

\[
\text{the transitions of the refined initialisation } T' \text{ establishes}
\]
\[
\text{that it is not the case that the transitions of the machine}
\]
\[
\text{initialisation } T \text{ do not establish the gluing invariant } J.
\]

In short PO 1 means that for every refined initialisation that establishes the linking invariant there is a machine initialisation that establishes it. PO 2 is similar but it is the bodies that are used to establish the state, with the addition that the outputs of the machine operation and the refined operation are the same each time. Hence PO 2 states that for every refined operation transition that establishes the linking invariant there is a machine operation transition that establishes it.

The examples used in this chapter have straightforward refinement relations so the refinement PO are discharge automatically by the development tool.

**Definition 87 (Refinement Proof Obligations).**

\[
PO 1 \quad [T'] - [T] \rightarrow J
\]

\[
PO 2 \quad I \land J \land P \Rightarrow [B'[y_i/y_i]] - [B] \land ([J \land \text{out}' = \text{out})]
\]
Chapter 6. Translating and Refining Annotations

REFINEMENT $R$
REFINES $M$
VARIABLES $V'$
INVARIANT $J$
INITIALISATION $T'$
OPERATIONS
$$y'_i \leftarrow OP_i(w, a_i) \triangleq (P'_i \mid B'_i)$$
END

Figure 6.2.: A B Refinement with Operations and I/O.

The refinement proof obligations in definition 87 formally connects the machine and refinement specifications.

6.1. The INTERLEAVED NEXT Annotation

Data processing routines in some instances are able to process several objects in parallel to increase throughput. The applications that have independent objects that do not rely on each other in any way are the easiest to apply parallel execution to. The target implementation language that is introduced later in this chapter allows parallel execution of independent objects. Hence the aim of future work is to introduce several annotations that extend the capabilities of B and permit parallel execution where it is safe. In this chapter a intermediate step to full parallel execution is taken: interleaved execution. The proof obligations that are introduced in this chapter are strong enough to support parallel execution. However, opting for interleaved behaviour means that synchronised action is not factored into the proof obligations.

Operations can be annotated to indicate interleaved execution with the INTERLEAVED NEXT annotation. Two or more operations are introduced (only two illustrated in definition 88). It would be possible to extend the annotation to introduce sets of operations in place of the single operations. Then any one of the operations from a set could be run in parallel with the operations from any of the other sets. This idea is returned to briefly in the tables discussing translations and refinements. Interleaved behaviour is not supported from the initialisation by annotations. There is no technical reason why this cannot be added in future work, but for now no PO for this is given. The first PO given is for interleaved behaviour after the current operation $OP_i$. The previous work on next invariants is not used in this chapter for brevity. However, there is no practical reason why they cannot be added. To include the NEXT INVARIANTS the PO must be extended as in chapter 4.
6.1.1. INTERLEAVED NEXT Proof Obligations

Definition 88 (Proof Obligations of INTERLEAVED NEXT). The annotation and PO are given as follows:

\[ y_i \leftarrow \text{Op}_i(v_i, e_i) \triangleq \text{PRE } P_i \text{ THEN } B_i \text{ END} \]

\[ /* \text{OPJ INTERLEAVED NEXT */ ; \]

The following PO arises:

\[ (I \land P_i \Rightarrow (T_{AB} \land [B_i(P_h[V/v_j]))) \land \cdots \land \]

\[ (I \land P_i \Rightarrow (T_{AB} \land [B_i(P_h[V/v_j]))) \land \]

\[ (\forall x, y \in OPJ \cdot x \neq y \Rightarrow (\text{variable-used}(x) \cap \text{variable-used}(y) = \{\})) \land \]

\[ (\forall x, y \in OPJ \cdot (\text{next}(x) = \text{next}(y))) \]

where

\[ v_j = v_{j1}, v_{j2}, \ldots, v_{j9} \text{ are free in } B_i \text{, where } T_{AB} \text{ is defined in Figure 4.1, page 76} \]

The interleaved annotation offers the option to execute two or more operations interleaved after the current operation, provided they do not set or read any variables in common. The proof obligation ensures that all the operations in the annotations are enabled after the current operation. To avoid interleaved execution continuing on after the stated next interleaved action each operation that can be interleaved must have the same next annotation (as stated in the PO). It ensures that a specific target operation is enabled after every branch of the interleaved execution has completed. In future work the full version of the annotation can be used which uses sets of next operations instead of individual next operations. Refinement reduces the set of possible alternative operations (possibly to the one). The form of this annotation is given in definition 89. The PO must take account of the other operations in the set, but the details of the PO are not given.

Definition 89 (The INTERLEAVED NEXT annotation over sets).

\[ y_i \leftarrow \text{Op}_i(v_i) \triangleq \text{PRE } P_i \text{ THEN } B_i \text{ END} \]

\[ /* \text{OPJ OPK INTERLEAVED NEXT */ ; \]

In the proof work to show that annotation-controller consistent machine-controller pairs do not diverge, lemma 14 is helpful. Because showing that the operations that the machines use do not overlap, in terms of state, which means that the operations are independent. In undertaking the proofs to establish the lemma it was shown that preconditions can not be used within the bodies of preconditioned operators.
6.1.2. State Independent Operations Have Disjoint Executions

Lemma 14 (Operations that do not modify each other's state are disjoint).

\[ \forall x, y \in \text{OPERATION} \cdot (\text{variable-used}(x) \cap \text{variable-used}(y) = \{\}) \Rightarrow (P_x \land P_y \land I \Rightarrow [B_a]P_y) \]

The proof of lemma 14 is derived of the substitutions in AMN:

1. \[ \text{variable-used}({\{O_{P_x}\}} \cap \text{Initial assumption} \setminus \text{variable-used}({\{O_{P_y}\}}) = \{\}) \]
2. \[ P_x \land P_y \land I \quad \text{Initial assumption} \]

Case A: \[ B_a = [a := E] \]

3. \[ a \notin \text{variable-used}(P_x) \quad \text{by 1 and Case A} \]
4. \[ P_y \quad \text{by 2} \]
5. \[ [a := E]P_y \quad \text{by 3 and 4} \]
6. \[ [B_a]P_y \quad \text{5 and case A} \]
7. \[ P_x \land P_y \land I \Rightarrow [B_a]P_y \quad \text{6 and 2} \]

QED

Case B: \[ B_a = [\text{skip}] \]

8. \[ P_y \quad \text{by 2} \]
9. \[ [\text{skip}]P_y \quad \text{8 and def. wp of skip} \]
10. \[ [B_a]P_y \quad \text{9 and case B} \]
11. \[ P_x \land P_y \land I \Rightarrow [B_a]P_y \quad \text{10 and 2} \]

QED
6.1. The interleaved next Annotation

Case C: $B_x = [S][T]$

12 $P_x \wedge P_y \wedge I \Rightarrow [S]P_y$ 
   Inductive hypothesis

13 $P_x \wedge P_y \wedge I \Rightarrow [T]P_y$ 
   Inductive hypothesis

14 $P_x \wedge P_y \wedge I \Rightarrow [S]P_y \wedge [T]P_y$ 
   by 12 and 13

15 $P_x \wedge P_y \wedge I \Rightarrow [S][T]P_y$ 
   14 and def. wp

QED

Case D: $B_x = [Q \Rightarrow S]$

16 $P_x \wedge P_y \wedge I \Rightarrow [S]P_y$ 
   Inductive hypothesis

17 $P_x \wedge P_y \wedge I \Rightarrow (\neg Q \lor [S]P_y)$ 
   by 14 and def. of $\lor$

18 $P_x \wedge P_y \wedge I \Rightarrow (Q \Rightarrow [S]P_y)$ 
   by 15 and def. $\Rightarrow$

19 $P_x \wedge P_y \wedge I \Rightarrow ([Q \Rightarrow S]P_y)$ 
   16 and def. wp

QED

Case E: $B_x = [IF Q THEN S ELSE U END]$

20 $P_x \wedge P_y \wedge I \Rightarrow [S]P_y$ 
   Inductive hypothesis

21 $P_x \wedge P_y \wedge I \Rightarrow [U]P_y$ 
   Inductive hypothesis

22 $P_x \wedge P_y \wedge I \Rightarrow (\neg Q \lor [S]P_y)$ 
   by 20 and def. $\lor$

23 $P_x \wedge P_y \wedge I \Rightarrow (Q \lor [U]P_y)$ 
   by 21 and def. $\lor$

24 $P_x \wedge P_y \wedge I \Rightarrow ((Q \Rightarrow [S]P_y) \land (\neg Q \Rightarrow [U]P_y))$ 
   by 22 and 23

25 $P_x \wedge P_y \wedge I \Rightarrow ([IF Q THEN S ELSE U END]P_y)$ 
   by def. of GSL.

QED
Case F: \( B_o = [\text{ANY } z \text{ where } z \in Z \text{ THEN } S \text{ END}] \)

26 \( P_x \land P_y \land I \Rightarrow [S]P_y \)  
Inductive hypothesis

27 \( \forall z \cdot z \in Z \Rightarrow (P_x \land P_y \land I \Rightarrow [S]P_y) \)  
by 26 and Generalisation of S

28 \( P_x \land P_y \land I \Rightarrow (\forall z \in Z \Rightarrow ([S]P_y)) \)  
by 27 and by \( x \) free in \( P_x, P_y, \) and \( I \)

29 \( P_x \land P_y \land I \Rightarrow [\text{ANY } z \text{ where } z \in Z \text{ THEN } S \text{ END}] \)  
by 28 and def. of \( \text{ANY} \)

QED

6.1.3. The FROM INTERLEAVED Annotation

The FROM INTERLEAVED annotations is added to allow backward reasoning to the operation that started the interleaving. In this chapter the FROM INTERLEAVING may not refer back to the initialisation. It may not follow the INITIALISATION.

Definition 90 (The FROM INTERLEAVING Annotation and Proof Obligation).

\( y_j \leftarrow O_{P_i}(w_j) \equiv \text{PRE } P_j \text{ THEN } B_j \text{ END} \)

\(*\text{FROM INTERLEAVED}(O_{P_i})(V_j)*/;

The following PO arises:

\( (I \land P_i \Rightarrow (T_{AB} \land [B_j](P_j[V_j/w_j]))) \)

where \( T_{AB} \) is defined in Figure 4.1, page 76.

The FROM INTERLEAVED annotation can be used in the same operation as the NEXT annotation, but not with FROM-ANY, FROM-SET or FROM-ANY-INIT. Disallowing FROM INTERLEAVING being used with interrupting annotations prevents any confusion arising from not knowing if the current operation was enabled by an interrupting annotated operation or an interleaved operation. The FROM INTERLEAVED annotation identifies which operation the current operation can be invoked from as part of an interleaving. It also identifies the input variable list \( (V_j) \) used with the invocation.

6.1.4. The SEQUENCE NEXT Annotation

In previous chapters annotation reasoning has been limited to adjacent operations. There are cases when this reasoning needs to be extended over a sequence of operations. For example
there may be a requirement to execute a specific sequence of operations once the first operation has started. The SEQUENCE NEXT annotation is used in this chapter to introduce refinements. Under refinement one operation can be replaced by a sequence of operations. To maintain any annotation relationships that existed before the refinement the new sequence of instructions must maintain a single continuous flow of enabled operations from beginning to end. The initialisation cannot be annotated with SEQUENCE NEXT annotations, because it is not allowed in this chapter for INITIALISATIONS to be control refined. The definition of the annotation and its POs has been restricted to a sequence of single operations rather than a sets of operations. The arrangement of sets of operations is useful when refinement can be used to reduce the sets. This situation is alluded to in the tables that follow.

Definition 91 (Proof Obligations of SEQUENCE NEXT).

\[ y_i \leftarrow \text{OP}_i(v_i) \triangleq \text{PRE } P_i \text{ THEN } B_i \text{ END } \]

\[
\text{/* } \{ \text{OP}_{j_1}, \ldots, \text{OP}_{j_k} \} \text{ SEQUENCE NEXT */ } \]

The following PO arises:

\[
(I \land P_i \Rightarrow (T_{AB} \land [B_i(V/v_j)])) \land
\]

\[
(I \land P_j \land \Rightarrow (T_{AB} \land [B_j(V/v_{j+1})])) \land \ldots \land
\]

\[
(I \land P_k \land \Rightarrow (T_{AB} \land [B_k(V/v_k)])) \land
\]

where \( T_{AB} \) is defined in Figure 4.1, page 76

The SEQUENCE NEXT annotation is conceptually different from the NEXT annotation, because it captures specific paths of executions that must exist in the controller. The current operation \( \text{OP}_i \) must enable the operation \( \text{OP}_j(v_j) \), and in turn that operation must enable \( \text{OP}_{j+1}(v_{j+1}) \), etc.

Definition 92 (INTERLEAVED NEXT Annotation Listing).

\[ \text{next} \text{ - interleaved}(y_i \leftarrow \text{OP}_i(v_i) / \text{OP}_j \text{ INTERLEAVED NEXT } */ ; ) = \]

\[ \{ \text{OP}_{j_1}(v_{j_1}), \ldots, \text{OP}_{j_n}(v_{j_n}) \} \]

where reference is made to Figure 4.1, page 76

Definition 93 (FROM INTERLEAVED Annotation Listing).

\[ \text{interleaved} - \text{from}(y_i \leftarrow \text{OP}_i(v_i) / \text{OP}_j \text{ FROM INTERLEAVED } */ ; ) = \]

\[ \{ \text{OP}_{j_1}(v_{j_1}), \ldots, \text{OP}_{j_n}(v_{j_n}) \} \]

where reference is made to Figure 4.1

JULY 6, 2008
6.1.5. A Simple Controller Language

Together all the annotations in a machine specify the controller design requirements/constraints. The CSP controller represents a refined view of the annotated B system.

To support translation a distinction is needed between operations that respond to external commands and those that are driven internally. A development will begin with a description of a number of operations: things that the system must do when commanded. During the development refinements will introduce internal operations. We distinguish between external and internal operations by marking the external operations with /* ext */ annotations, which are discussed in more detail in the refinement and translation section 6.3.

Definition 94 details the CSP subset of control fragments used in this chapter: event prefix, choice, interleaving, if-then-else, and recursion control.

Definition 94 (Controller Syntax with Interleaving).

\[
R ::= \square ayledy*v \rightarrow R(v) | \\
R \square R | \\
R \Delta R | \\
R \Delta_x R | \\
R \Delta^i_x R | \\
\text{if } y \text{ then } R \text{ else } S | \\
( \big| \square a_{i | y_u | v_u \rightarrow \text{skip}; R | \\
S(p) \\)
\]

In this chapter the CSP controller is a different view of the annotated B specification. A more complex arrangement arises if the CSP controller is permitted to carry around local state. The simplified view is pictorially represented in Figure 6.1, page 130. An annotated B machine output is the same as a CSP controller output. In definition 94 the channel \(a\), in the controller fragment \(a| y*v \rightarrow R\), is an operation name with a choice over all possible outputs \(y\) (from the controller's point of view, if \(a\) is called then any output \(y\) within the bounds of \(y\)'s type should be allowed). The outputs are fresh and modelled as a distributed external choice ranging over the type given in the B (the type is not always given in the controller definition). The channel has an input vector \(v\). To accommodate analysis, finite types are used in the CSP. The same restriction does not exist in the B. Hence the CSP representation of the B operation may not be a true representation in terms of input and output ranges. The interleave operator executes the two or more processes concurrently that do not synchronise on any events. The remaining operators are described in definition 82, page 112. In a controller definition, all process variables used are bound by some recursive definition.

A major constraint is enforced on the way controllers can be written, which facilitates translations, but turns out not to be so troublesome as it first appears. Definition 95 gives the constrained arrangement. Controllers must start with an initial loop (R1), then enter a main
6.1. The interleaved next Annotation

loop \((S \equiv R2)\). A controller \(CTRL\) has a definition, \(R1\), given in definition process variables are the same, \(S\). The definition of \(S\) is \(R2\) which is given in definition 94. The only recursive calls allowed are to \(S\).

**Definition 95 (Controller Syntax with I/O).**

\[
CTRL \triangleq R1 \\
S \triangleq R2
\]

where \(R1\) and \(R2\) are terms from definition 94 and

\(S\) is the only recursive variable allowed and

\(R2\) is guarded as defined in definition 98.

The results presented in this chapter require that all recursive definitions are guarded, which means that at least one event must occur before a recursive call. The meaning of consistency between the controller and the annotations is given in terms of the \(init\) functions. The \(init\) function returns a set of operations available next and is developed in definition 96.

**Definition 96 (init for CSP Controller Process Fragments with Interleaving).**

\[
init(\square \alpha) y?v \rightarrow R1 = \{a,y,v\} \\
init(R1 \square R2) = init(R1) \cup init(R2) \\
init(R1 \triangle R2) = init(R1) \cup init(R2) \\
init(R1 \triangle x R2) = init(R1) \\
init(R1 \triangle y R2) = init(R1) \cup init(R2)
\]

\[
init((\bigparallel_{i=1}^n \square a_i) y, u \rightarrow \text{skip}; R) = \{\}
\]

\[
init(\text{if } y \text{ then } R1 \text{ else } R2) = init(R1) \cup init(R2) \\
init(S(y)) = init(R(y))
\]

An action prefix must appear with output on the left. In the first case of the \(init\) definition 96 the head of the control fragment is extracted. The outputs and inputs of the action are the same as the outputs and inputs of the \(B\) operation. The \(init\) of a prefixed action is the action. The \(init\) of a choice between two processes is the union of the \(init\) of the individual processes. The \(init\) of the interleaving is empty. The initial actions of interleaved control fragments is obtained using the \(init - \text{par}\). Annotations clearly show an ordering of operations: an initial operation and a set of next operations. The \(guard\) function is defined in definition 84. Prefixed operations are guarded. A fragment with an external choice separating the two processes is prefixed if the individual processes are guarded. Similarly with the if-then-else. The parameterised process variable is not guarded, whereas the recursive definition is guarded if the body is guarded.

In this chapter the requirement for a special \(init\) operation, \(init - \text{par}\), arise as a result of the need to ensure that an action prefix to an interleaving is correctly annotated.
6. Translating and Refining Annotations

Definition 97 (init for CSP Controller Process Fragments with Interleaving).

\[
\begin{align*}
\text{init} - \text{par}(&\emptyset a!y?v \rightarrow R1) = \{\} \\
\text{init} - \text{par}(R1 \sqcup R2) &= \text{init} - \text{par}(R1) \cup \text{init} - \text{par}(R2) \\
\text{init} - \text{par}(R1 \triangle R2) &= \text{init} - \text{par}(R1) \cup \text{init} - \text{par}(R2) \\
\text{init} - \text{par}(R1 \triangle X R2) &= \text{init} - \text{par}(R1) \\
\text{init} - \text{par}(\langle \prod_{i=1}^{n} a_i!y_{a_i}?v_{a_i} \rightarrow \text{skip} \rangle; R1) &= \{a_i | i = 1...n\} \\
\text{init} - \text{par}(\text{if } y \text{ then } R1 \text{ else } R2) &= \text{init} - \text{par}(R1) \cup \text{init} - \text{par}(R2) \\
\text{init} - \text{par}(S(p)) &= \text{init} - \text{par}(R(p))
\end{align*}
\]

Definition 98 (guarded on CSP Controller Process with Interleaving).

\[
\begin{align*}
\text{guarded}(\emptyset a!y?v \rightarrow R1) &= \text{true} \\
\text{guarded}(R1 \sqcup R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \triangle R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(R1 \triangle X R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(\langle \prod_{i=1}^{n} a_i!y_{a_i}?v_{a_i} \rightarrow \text{skip} \rangle; R1) &= \text{true} \\
\text{guarded}(\text{if } y \text{ then } R1 \text{ else } R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(\text{if } \text{false } y \text{ then } R1 \text{ else } R2) &= \text{guarded}(R1) \land \text{guarded}(R2) \\
\text{guarded}(S(p)) &= \text{false} \\
\text{guarded}(S(p) \equiv R(p)) &= \text{true} \quad \text{if } \text{guarded}(R(p)) = \text{true}
\end{align*}
\]

6.2. Annotation Consistency with Interleaving

Annotation consistency between a guarded controller and the annotated B machine is broken down into annotation consistency (definition 99) and step-consistency (definition 100).

Definition 99 (Annotation Consistency with Interleaving). The annotation consistency of the controller fragment \( R \) is defined as follows:

1. \( a!y?v \rightarrow R \)

is annotation consistent with Machine \( M \) if \( a!y?v \rightarrow R \) is initially consistent and \( a!y?v \rightarrow R \) is step-consistent with \( M \)

\( a!y?v \rightarrow R \) is initially-consistent if \( a!y?v \in \text{next(INITIALISATION)} \)
2. \( R1 \sqcup R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) and \( R2 \) are annotation consistent with \( M \).

3. \( R1 \triangle R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) is annotation consistent and

\[ \forall \text{op} \in \text{init}(R2) \cdot \text{from} - \text{any}(\text{op}) = \text{true} \]

and \( R2 \) is step-consistent with \( M \).

4. \( R1 \triangledown X R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) is annotation consistent with \( M \) and

\[ \forall \text{op} \in \text{init}(R2) \cdot X \subseteq \text{from} - \text{set}(\text{op}), \]

and \( R2 \) is step-consistent with \( M \).

5. \( R1 \triangledown X R2 \)

is annotation consistent with Machine \( M \) if \( R1 \) is annotation consistent with \( M \) and

\[ \forall \text{op} \in \text{init}(R2) \cdot X \subseteq \text{from} - \text{set}\_\text{i}(\text{op}), \]

and \( R2 \) is step-consistent with \( M \).

6. \( \text{if} \ b \ \text{then} \ R1 \ \text{else} \ R2 \)

is annotation consistent with Machine \( M \) if it is step-consistent.

7. \( \forall a_1 Y_{a_1} : ?va_1 \rightarrow \text{skip}; \)

is annotation consistent with Machine \( M \) if it is step-consistent.

8. \( S(p) \)

is annotation consistent with Machine \( M \).

A family of recursive definitions \( S(p) \triangleq R(p) \) is annotation consistent with \( M \)'s annotations if each \( R(p) \) is annotation consistent with \( M \)'s annotations.

An initialisation can not have an output which rules out the use of an \( \text{if} \ - \ \text{then} \ - \ \text{else} \) annotation on the initialisation. The interleaving operator given in definition 100 is generic. The examples are restricted to two interleaved operations to demonstrate the principle.

**Definition 100 (Step-Consistency with Interleaving).** The step-consistency of the controller fragment \( R \) is defined as follows:

July 6, 2008

141
Chapter 6. Translating and Refining Annotations

1. \( \Box_y a y? v \to R \)
   is step-consistent with Machine \( M \) if \((\forall b \in \text{init}(R) \cdot b \in \text{next}(y \leftarrow a(v))) \) or \((\forall b \in \text{init} - \text{par}(R) \cdot b \in \text{next} - \text{interleaved}(y \leftarrow a(v)))\), and \( R \) is step-consistent with \( M \).

2. \( R_1 \sqcap R_2 \)
   is step-consistent with Machine \( M \) if \( R_1 \) and \( R_2 \) are step-consistent with \( M \).

3. \( R_1 \bigoplus R_2 \)
   is step-consistent with Machine \( M \) if \( R_1 \) is step-consistent with \( M \) and \( \forall z \in \text{init}(R_2) \cdot \text{from} - \text{any} - \text{op}(y_z \leftarrow x(v_z)) = \text{true} \) and \( R_2 \) is step-consistent.

4. \( R_1 \bigtriangledown_R R_2 \)
   is step-consistent with Machine \( M \) if \( R_1 \) is step-consistent with \( M \) and \( \forall b \in \text{init}(R_2) \cdot X \subseteq \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b)) \), and \( R_2 \) is step-consistent with \( M \).

5. \( R_1 \bigtriangledown_X R_2 \)
   is step-consistent with Machine \( M \) if \( R_1 \) is step-consistent with \( M \) and \( \forall b \in \text{init}(R_2) \cdot X \subseteq \text{from} - \text{set} - \text{init} - \text{op}(y_b \leftarrow b(v_b)) \) and \( R_2 \) is step-consistent with \( M \).

6. if \( y \) then \( R_1 \) else \( R_2 \)
   is step-consistent with Machine \( M \) if \( y \in \text{BOOL} \) or \( y \in \text{N} \) and \( R_1 \) and \( R_2 \) are step-consistent with \( M \) and \( \forall b \in \text{init}(R_1) \Rightarrow b \in \text{condition_true}(a y? v) \) and \( \forall c \in \text{init}(R_2) \Rightarrow c \in \text{condition_false}(a y? v) \).

7. \( ( \bigcup_{i=1}^n \Box_y a_i y? v_{a_i} \to \text{skip})_R \)
   is step-consistent with Machine \( M \) if \( \exists v. \forall a : 1..n \cdot v = \text{from} - \text{interleaved}(a_z) \) and
6.2. Annotation Consistency with Interleaving

\[ \forall \text{op} \in \{a_1, \ldots, a_n\} \cdot \text{init}(R) \subseteq \text{next}(\text{op}) \text{ and } R \text{ is step-consistent with } M \]

8. \( S(p) \) is step-consistent with Machine \( M \)

A family of recursive definitions \( S = R \) is step-consistent with \( M \)'s annotations if each \( R \) is step-consistent with \( M \)'s annotations.

The interleaving operator can only be shown to be consistent in a very limited sense. Actions are allowed to occur interleaved provided they do not attempt to change the variables used by the other actions.

We re-cap on the main proof in this thesis which is that if \( R \) is annotation consistent with the annotations of a machine \( M \), and the annotations of \( M \) are consistent with machine \( M \), then operations of \( M \) called in accordance with the control flow of \( R \) will never be called outside their preconditions. The key feature of the proof of this main result is an argument that no trace of \( R \) leads to an operation of \( M \) called outside its precondition or guard. This is established by building up the traces of \( R \) and showing that at each step an operation called outside its precondition cannot be introduced, by appealing to the relevant annotation and applying its proof obligation. The benefit of this result is that the details of the operations of \( M \) are required only for checking the consistency of the annotations, and are not considered directly in conjunction with the controller. The annotations are then checked against the controller using the definition of consistency above. This enables a separation of concerns, treating the annotations as an abstraction of the B machine.

6.2.1. Proving Termination of Controlled Machines with interleaved NEXT

The proofs in this section establish that it is enough to demonstrate annotation-consistency (given machine-annotation consistency) to show that the every step of the machine-controller pair terminates, which implies non-divergence: every operation is called within its precondition. The proofs in this section extend section 5.6 by introducing new proofs for machines with interleaved NEXT annotations and controllers with interleaved operations.

The proof of theorem 5, page 116, is repeated with interleaving in the controller language to show that only good traces are produced by \( M \_CTRL \). Hence \( M \ || \ M \_CTRL \) is divergence free. The length of the traces are reasoned over in the proof.

The theorem for controllers for \( \text{NEXT INTERLEAVED} \) is restated:

**Theorem 6 (\text{NEXT INTERLEAVED} Annotated Consistent Controller are Divergence Free).** If a \text{NEXT INTERLEAVED} annotated \( B \) Machine \( M \) is consistent with a controller \( M \_CTRL \) then \( M \ || \ M \_CTRL \) is divergent free. The inductive hypothesis is that any sub-trace of \( M \_CTRL \) is good.

When the trace is initially-consistent:
Case (A) () The empty trace

Entirely similar to the proof of theorem 1, page 30, Case A

Case (B) \((a, y, v) \in \text{traces}(M_{CRTL})\) The singleton trace

No new annotation added for INITIALISATION.

Entirely similar to the proof of theorem 5, Case B.

Case (C) When the trace is step-consistent:

\((tr) \wedge (a, y_a, v_a, b, y_b, v_b) \in \text{traces}(M_{CRTL})\) where \(tr \in \text{traces}(M_{CRTL})\)
6.2. Annotation Consistency with Interleaving

1. \( \text{tr} \cap \{a.y.a, v_a\} \) is good

by inductive hypothesis

There are 3 major cases C1 - C3

(C1) The case of the existing PO structure -

\[
\begin{align*}
&b.y_a.v_b \in \text{next}(y_a \leftarrow a(v_a)) \lor \\
&\text{from} - \text{any} - \text{op}(y_b \leftarrow b(v_b)) \lor \\
&a.y_a.v_a \in \text{from} - \text{set} - \text{op}(y_b \leftarrow b(v_b)) \lor \\
&a.y_a.v_a \in \text{from} - \text{set} - \text{init} - \text{op}(y_b \leftarrow b(v_b)) \lor \\
&(\text{condition}(y_a \leftarrow a(v_a)) \land \\
&b.y_b.v_b \in \text{condition_true}(y_a \leftarrow a(v_a)) \lor \\
&\neg \text{condition}(y_a \leftarrow a(v_a)) \land \\
&b.y_b.v_b \in \text{condition_false}(y_a \leftarrow a(v_a)) \lor \\
&(b.y_b.v_b \in \text{next} - \text{interleaved}(a.y.v) \land \\
&a.y_a.v_a \in \text{from} - \text{interleaved}(b.y_b.v_b)) \\
\end{align*}
\]

(C2) Two interleaved operations

\[
\begin{align*}
&(\exists v \cdot (\forall x \in \{t_1, \ldots, t_n, a.y_a.v_a, b.y_b.v_b\} \\
\cdot (v \in \text{from} - \text{interleaved}(x) \Rightarrow \\
\text{tr} \cap \{a.y_a.v_a, b.y_b.v_b\} = \\
(tr_0 \cap \{v\} \cap (t_1, \ldots, t_n) \cap \{a.y_a.v_a, b.y_b.v_b\}))))) \text{ consistency rules} \\
\end{align*}
\]

(C3) At the end of interleaving

\[
\begin{align*}
&(\exists v \cdot (\forall x \in \{t_1, \ldots, t_n, a.y_a.v_a\} \\
\cdot (v \in \text{from} - \text{interleaved}(x) \land \\
b \in \text{next}(X) \Rightarrow \\
\text{tr} \cap \{a.y_a.v_a, b.y_b.v_b\} = \\
(tr_0 \cap \{v\} \cap (t_1, \ldots, t_n) \cap \{a.y_a.v_a, b.y_b.v_b\}))))) \text{ consistency rules} \\
\end{align*}
\]

There are three cases to consider to complete this proof. The annotations that are associated with the \( P_a \land I \Rightarrow [B_a]P_b \) PO are dealt with first in proof section case C1. They include the NEXT, FROM, NEXT INTERLEAVED and FROM INTERLEAVED. The NEXT INTERLEAVED and FROM INTERLEAVED are associated and are used to mark the beginning of a interleaved sequence. The second case is concerned with the situation when an interleaving is underway, which is the case in which two consecutive operations are are interleaved. (In case C2 both operations will be annotated with FROM INTERLEAVED.) Finally there is the case when the final operation of an interleaving has occurred, C3. In case C3, \( a.y_a.v_a \) is the final interleaved operation and \( b.y_b.v_b \) is the operation (which has been removed from the set of interleaved operations previously given in C2) that can follow any of the interleaved operations, because each interleave operation enables it (weak annotation interleaving). The parameters
are dropped, in all case but the conditional case, because machine-annotation consistency ensures that they are type consistent. The parameters are left in for the conditional cases because the output is used to decide on the exact case.

Case C1, next, from, next interleaved and from interleaved at the start leaving:

\[
2 \ b \in \text{next}(a) \lor \\
\text{from} \rightarrow \text{any} \rightarrow \text{op}(b) \lor \\
\text{a} \in \text{from} \rightarrow \text{set} \rightarrow \text{op}(b) \lor \\
\text{a} \in \text{from} \rightarrow \text{set} \rightarrow \text{init} \rightarrow \text{op}(b) \lor \\
(\text{condition}(y_a \leftarrow a(v_a, e_a)) \land \\
b \in \text{condition_true}(y_a \leftarrow a(v_a, e_a))) \lor \\
(\neg \text{condition}(y_a \leftarrow a(v_a, e_a)) \land \\
b \in \text{condition_false}(y_a \leftarrow a(v_a, e_a))) \lor \\
(b \in \text{next} \rightarrow \text{interleaved}(a)) \land \\
a \in \text{from} \rightarrow \text{interleaved}(b)
\]

Entirely similar to the proof of theorem 5, page 116, Case (C), from line 2, replacing references to lemma 6, page 60 with lemma 17, page 150

Case C2, interleaved elements

\[
3 \ \exists z \cdot (\forall x \in \{ t_1, \ldots, t_n, a, b \} \Rightarrow \\
(\text{z} \in \text{from} \rightarrow \text{interleaved}(x)) \land \\
\text{tr} \cap \{ a, b \} = \\
\text{tr}_0 \cap \{ z \} \cap \{ t_1, \ldots, t_n \} \cap \{ a, b \})
\]

\[
4 \ \text{[init; tr}_0; z]\text{true} \quad \text{by Case C2 and inductive hypothesis}
\]

\[
5 \ \exists z \cdot (\forall x \in \{ t_1, \ldots, t_n, a, b \} \Rightarrow \\
(\text{z} \in \text{from} \rightarrow \text{interleaved}(x))) \quad \text{by 3 and Case C2}
\]

\[
6 \ \text{tr}_0 \cap \{ z \} \text{is good} \quad \text{by 4}
\]

\[
7 \ \text{tr}_0 \cap \{ z \} \cap \{ t_1, \ldots, t_n \} \cap \{ a, b \} \text{is good} \quad \text{by lemma 15 (below), 5 and 6}
\]
6.2. Annotation Consistency with Interleaving

Case 3, interleaving completed

\[ \exists v \cdot (\forall x \in \{ t_1, ..., t_n, a.y_a.v_a \} \cdot v \in \text{from} - \text{interleaved}(x) \land b \in \text{next}(x) \Rightarrow \]
\[ (tr \cap \{ a.y_a.v_a, b.y_b.v_b \} =
tr_0 \cap \{ v \} \cap \{ t_1, ..., t_n \} \cap
\{ a.y_a.v_a, b.y_b.v_b \})) \]

9 \ [\text{init}; \ tr_0; z] \text{true} \quad \text{by Case C3 and inductive hypothesis}

10 \ \exists z \cdot (\forall x \in \{ t_1, ..., t_n, a.y_a.v_a \} \Rightarrow
(x \in \text{from} - \text{interleaved}(x)) \quad \text{by Case C3}

11 \ \forall y \in \{ t_1, ..., t_n, a.y_a.v_a \} \cdot
(b.y_a.v_a \in \text{next}(y)) \quad \text{by Case C3}

12 \ tr_0 \cap \{ z \} \text{is good} \quad \text{by 9}

13 \ tr_0 \cap \{ z \} \cap \{ t_1, ..., t_n \} \cap
\{ a.y_a.v_a, b.y_b.v_b \} \text{is good} \quad \text{by lemma 16 (below), 10, 11 and 12}

Next-par Annotation Lemma

Lemma 15 (Consecutive actions referenced by a common next-par are good). Two
actions or more in a sequence of actions that all have a common from-par root will terminate:

\[ \exists z \cdot (\{ t_1, ..., t_n, a, b \} \subseteq \text{next} - \text{interleaved}(z) ) \land
tr_0 \cap \{ z \} \text{is good} \Rightarrow
tr_0 \cap \{ z \} \cap \{ t_1, ..., t_n, a, b \} \text{is good} \]

The proof of lemma 15 is developed by constructing a good trace from the action that has
the \text{INTERLEAVED NEXT} annotation.

JULY 6, 2008
Chapter 6. Translating and Refining Annotations

1. $tr_0 \models \{ z \} \text{ is good}$

2. $\exists z \cdot (t_1, \ldots, t_n, a, b) \\ \subseteq \text{next - interleaved}(z)$

3. $\forall x, y \in \text{next - par}(z) \cdot x \neq y \Rightarrow \\ P_x \land P_y \land I \Rightarrow [B_z]P_y$

4. $\forall j \in \{ t_1, \ldots, t_n, a, b \} \\ (P_x \land I \Rightarrow [B_z](P_j))$

5. $\text{[init; } tr_0; (P_a \mid B_z)]\text{true}$

6. $\text{[init; } tr_0; (P_a \mid B_z)]\text{true}$

7. $\text{[init; } tr_0; I]$ \text{by consistency of machine and 1}

8. $\text{[init; } tr_0; (P_z \land I)$ \text{by 6 and 7}

New goal: $\text{tr}_0 \models \{ z \} \models \{ \}$

$\{a, b\} \text{ is good}$ \text{(Base case)}

9. $\text{[init; } tr_0; ([B_z]P_a)$ \text{by 4 and 8}

10. $\text{[init; } tr_0; (P_x \land [B_z]P_a)$ \text{by 8 and 9}

11. $\text{[init; } tr_0; (P_x \mid B_z)]P_a$ \text{by 10 and def. of pre}

12. $\text{[init; } tr_0; ([B_z]P_b)$ \text{by 6 and 12}

13. $\text{[init; } tr_0; ([P_x \land [B_z]P_b)$ \text{by 4 and 8}

14. $\text{[init; } tr_0; (P_x \mid B_z)]P_b$ \text{by 13 and def. of pre}

15. $\text{[init; } tr_0; (P_x \mid B_z)](P_a \land [B_z]P_b)$ \text{by 7, 11 and 14}

16. $\text{[init; } tr_0; (P_x \mid B_z)]([B_z]P_b)$ \text{by 3 and 15}

17. $\text{[init; } tr_0; (P_x \mid B_z)](P_a \land [B_z]P_b)$ \text{by 11 and 16}

18. $\text{[init; } tr_0; (P_x \mid B_z); (P_a \mid B_a)]P_b$ \text{by 17 and def. pre}

19. $\text{[init; } tr_0; (P_x \mid B_z); (P_a \mid B_a)]I$ \text{by machine consistency}

20. $\text{[init; } tr_0; (P_x \mid B_z); (P_a \mid B_a)]I \land P_b$ \text{by 18 and 19}

21. $\text{[init; } tr_0; (P_x \mid B_z); (P_a \mid B_a)][B_b]I$ \text{by 20 and machine consistency}

22. $\text{[init; } tr_0; (P_x \mid B_z); (P_a \mid B_a)][B_b]I \land [B_b]I$ \text{by 18, 21 and machine consistency}

23. $\text{[init; } tr_0; (P_x \mid B_z); (P_a \mid B_a); (P_b \mid B_b)]I$ \text{by 22 and def. pre}

24. $\text{[init; } tr_0; (P_x \mid B_z); \\ \cdots; (P_a \mid B_a); (P_b \mid B_b)]\text{true}$ \text{by 23 and machine consistency}

Induction base case
New goal : $tr_0 \wedge \langle z \rangle \wedge \langle t_1, ..., t_k \rangle \wedge \langle a, b \rangle$ is good

Inductive case

25 $[\text{init}; tr_0; (P_z | B_z); (P_{t_1} | B_{t_1}); \\
...; (P_{t_k} | B_{t_k})]P_a \land P_b$  
by consistency of interleaving actions 100

26 $[\text{init}; tr_0; (P_z | B_z); (P_{t_1} | B_{t_1}); \\
...; (P_{t_k} | B_{t_k})]I$  
by machine consistency

27 $[\text{init}; tr_0; (P_z | B_z); (P_{t_1} | B_{t_1}); \\
...; (P_{t_k} | B_{t_k})](I \lor P_a \land P_b)$  
by 25 and 26

28 $[\text{init}; tr_0; (P_z | B_z); (P_{t_1} | B_{t_1}); \\
...; (P_{t_k} | B_{t_k})][B_a]P_b$  
from new goal
by 27 and lemma 14

Entirely similar to lines 17-24 above

$QED$

---

From Interleaved Annotation Lemma

Lemma 16 (Traces with Consecutive Actions with Overlapping From Interleaved Annotations are good). Two actions in a sequence of actions that:

$$\exists v \cdot (\forall x \in \{ t_1, ..., t_n, a \} \cdot v \in \text{from-interleaved}(x) \land b \in \text{next}(y) \implies (tr \circ \langle a, b \rangle = tr_0 \circ \langle v \rangle \circ \langle t_1, ..., t_n, a \rangle \circ \langle b \rangle))$$

The proof of lemma 16 is developed by constructing a good trace from the action that has the \text{INTERLEAVED NEXT} annotation.

July 6, 2008
Chapter 6. Translating and Refining Annotations

1 \exists v \cdot \forall a \in \{ t_1, ..., t_n, a \} \\
v \in \text{from - interleaved}(x) \Rightarrow \\
(\forall y \in \{ t_1, ..., t_n, a \} \land \\
b \in \text{next}(y)))

Initial assumption

2 \text{[init; ...; } (P_{k_1} \mid B_{k_1}; ... \\
; (P_{k_n} \mid B_{k_n})]P_a

where \( k_1 \) to \( k_n \) are drawn \\
from \( t_1 \) to \( t_n \) line 1.

3 \text{[init; ...; } (P_{k_1} \mid B_{k_1}; ... \\
; (P_{k_n} \mid B_{k_n})]P_a \land I

by 1 and pod of \text{next}

4 \text{[init; ...; } (P_{k_1} \mid B_{k_1}; ... \\
; (P_{k_n} \mid B_{k_n})]I

by 2 and machine consistency

5 \text{[init; ...; } (P_{k_1} \mid B_{k_1}; ... \\
; (P_{k_n} \mid B_{k_n})]P_a \land I

by 3 and 4

6 \text{[init; ...; } (P_{k_1} \mid B_{k_1}; ... \\
; (P_{k_n} \mid B_{k_n})][B_a](P_b)

by 3 and 5

Entirely similar to lines 17-24, lemma 15

QED

Arbitrary Traces are Enabled by Annotations

Lemma 17 (Arbitrary Length Traces of Annotation Consistent Interleaved Controllers are Enabled by Annotations). The actions of arbitrary length traces of annotation-controller consistent controllers that include interleaved operators are enabled by annotations:

\[
R \text{ annotation consistent} \land \text{tr} \models \left( \forall a \right) \left( \forall b \right) \in \text{traces}(R) \Rightarrow \\
b \in \text{next}(a) \land \\
\text{from - any - op}(b) \land \\
a \in \text{from - set - op}(b) \land \\
(\text{condition}(y_a \leftarrow a(v_a)) \land \\
b \in \text{condition - true}(y_a \leftarrow a(v_a))) \lor \\
(\lnot \text{condition}(y_a \leftarrow a(v_a)) \land \\
b \in \text{condition - false}(y_a \leftarrow a(v_a))) \lor \\
(b \in \text{next - interleaved}(a) \land \\
a \in \text{from - interleaved}(b))
\]
Proof of Lemma 17 is as follows (dropping arguments for non-conditional operators):

Cases of controllers:

1. \textbf{case A} \quad c \rightarrow R

\begin{enumerate}
\item \quad c \rightarrow R \text{ step-consistent with } M \quad \text{initial assumption}
\item \quad \text{sub-case: } tr = \emptyset
\item \quad tr \cap (a) \cap (b) \in traces(c \rightarrow R) \quad \text{initial assumption}
\item \quad \langle a \rangle \cap \langle b \rangle \in traces(c \rightarrow R) \quad tr = \emptyset
\item \quad a = c \text{ and } \langle b \rangle \in traces(R) \quad \text{def. of traces}
\end{enumerate}

Either init-par = \{\} (an interleaved operator) or init = \{\} (a non-interleaved operator)

\begin{enumerate}
\item \quad \text{sub-sub-case init-par(R) = \{\}}
\item \quad b \in init(R) \quad \text{by 4, lemma 6, page 60}
\item \quad b \in \text{next}(a) \quad \text{by 1, and 5}
\item \quad b \in \text{next}(a) \quad \text{by 6, and 4}
\item \quad b \in \text{next}(a) \lor \text{from - any}(b) \lor \text{from - set}(b) \lor \text{from - init}(b)
\quad \text{condition}(y_a \leftarrow a(v_a)) \land \text{condition_true}(y_a \leftarrow a(v_a)) \lor \text{condition_false}(y_a \leftarrow a(v_a)) \lor \text{next - interleaved}(a) \land a \in \text{from - interleaved}(b)
\end{enumerate}

QED
Chapter 6. Translating and Refining Annotations

sub-sub-case $\text{init}(R) = \{\}$

9. $b \in \text{init} \cap \text{interleaved}(R)$
   4. lemma 18 below

10. $b \in \text{next} \cap \text{interleaved}(c)$
    1. by 1 and 9

11. $b \in \text{next} \cap \text{interleaved}(a)$
    4. by 10 and 4

12. $b \in \text{next}(a) \lor$
    from - any(b) \lor
    $a \in \text{from} - \text{set}(b) \lor$
    $a \in \text{from} - \text{set} - \text{init}(b)$
    $(\text{condition}(y_a \leftarrow a(v_a))) \land$
    $b \in \text{condition\_true}(y_a \leftarrow a(v_a)) \lor$
    $(\neg \text{condition}(y_a \leftarrow a(v_a))) \land$
    $b \in \text{condition\_false}(y_a \leftarrow a(v_a)) \lor$
    $(b \in \text{next} \cap \text{interleaved}(a) \land$
    $a \in \text{from} - \text{interleaved}(b))$

    QED

sub-case: $tr \neq \{\}$

13. $tr \cap \{a\} \cap \{b\} \in \text{traces}(c \rightarrow R)$
    initial assumption

14. let $tr = \{e\} \cap tr'$

15. $(e) \cap tr' \cap \{a\} \cap \{b\} \in \text{traces}(c \rightarrow R)$
    initial assumption

16. $e = c \land tr' \cap \{a\} \cap \{b\} \in \text{traces}(R)$
    15. def. traces

17. $tr' \cap \{a\} \cap \{b\} \in \text{traces}(R)$
    by 16

18. $b \in \text{next}(a) \lor$
    from - any(b) \lor
    $a \in \text{from} - \text{set}(b) \lor$
    $a \in \text{from} - \text{set} - \text{init}(b) \lor$
    $(\text{condition}(y_a \leftarrow a(v_a))) \land$
    $b \in \text{condition\_true}(y_a \leftarrow a(v_a)) \lor$
    $(\neg \text{condition}(y_a \leftarrow a(v_a))) \land$
    $b \in \text{condition\_false}(y_a \leftarrow a(v_a)) \lor$
    $(b \in \text{next} \cap \text{interleaved}(a) \land$
    $a \in \text{from} - \text{interleaved}(b))$

    QED
6.2. Annotation Consistency with Interleaving

case B \( R_1 \square R_2 \)

Entirely similar to the proof of Lemma 5, page 55, case B.

Substituting for the inductive hypothesis on lines 6 and 7 of the proof of lemma 5 with the following:

\[
\begin{align*}
    b & \in \text{next}(a) \lor \\
    \text{from} - \text{any}(b) & \lor \\
    a & \in \text{from} - \text{set}(b) \lor \\
    a & \in \text{from} - \text{set - init}(b) \\
    (\text{condition}(y_a \leftarrow a(v_a)) \land \\
    b & \in \text{condition\_true}(y_a \leftarrow a(v_a)) \lor \\
    \neg \text{condition}(y_a \leftarrow a(v_a)) \land \\
    b & \in \text{condition\_false}(y_a \leftarrow a(v_a)) \lor \\
    (b & \in \text{next - interleaved}(a) \land \\
    a & \in \text{from - interleaved}(b)) \lor \\
    (b & \in \text{next - interleaved}(a) \land \\
    a & \in \text{from - interleaved}(b))
\end{align*}
\]

case C \( R_1 \triangle R_2 \)

Entirely similar to the proof of Lemma 5, case C.

Substituting for the inductive hypothesis on lines 8, 10 and 14 as in Case B above.

case D \( (R_1 \triangle X R_2) \)

Entirely similar to the proof of Lemma 5, case D.

Substituting for the inductive hypothesis on lines 8, 11, and 13 as in Case B above.

case E \( (R_1 \triangle X^i R_2) \)

Entirely similar to the proof of Lemma 5, case E.

Substituting for the inductive hypothesis on lines 8, 10, 18 and 21 as in Case B above.

case F \( S(p) \)

Entirely similar to the proof of Lemma 5, case F.

Substituting for the inductive hypothesis on lines 5 as in Case B above.
Chapter 6. Translating and Refining Annotations

case G if condition then R1 else R2

1. \( \text{tr} \cap (a) \cap (b) \in \text{traces(} (\text{if } y \text{ then } R1 \text{ else } R2) \text{)} \)
   \begin{align*}
   &\text{initial assumption} \\
   &2. \text{if } y \text{ then } R1 \text{ else } R2 \text{ step-consistent with } M \text{ initial assumption} \\
   &3. R1 \text{ step-consistent with } M \quad 2, \text{ def. step-consistency 100} \\
   &4. R2 \text{ step-consistent with } M \quad 2, \text{ def. step-consistency 100} \\

\text{case 1 } y = \text{true}

5. \( \text{tr'} \cap (a) \cap (b) \in \text{traces}(R1) \)
   \begin{align*}
   &\text{by 1 and } y = \text{true} \\
   &6. b \in \text{next}(a) \lor \\
   &\quad \text{from } \text{any}(b) \lor \\
   &\quad a \in \text{from } \text{set}(b) \lor \\
   &\quad a \in \text{from } \text{set } \text{init}(b) \\
   &\quad \text{condition}(y_a \leftarrow a(v_a)) \land \\
   &\quad b \in \text{condition_true}(y_a \leftarrow a(v_a)) \lor \\
   &\quad \neg \text{condition}(y_a \leftarrow a(v_a)) \land \\
   &\quad b \in \text{condition_false}(y_a \leftarrow a(v_a)) \lor \\
   &\quad (b \in \text{next } \text{interleaved}(a) \land \\
   &\quad a \in \text{from } \text{interleaved}(b)) \\
   \end{align*}

\( \text{QED} \)

\text{case 2 } y = \text{false}

7. \( \text{tr} \cap (a) \cap (b) \in \text{traces}(R2) \)
   \begin{align*}
   &\text{by 4 and } y = \text{false} \\
   &8. b \in \text{next}(a) \lor \\
   &\quad \text{from } \text{any}(b) \lor \\
   &\quad a \in \text{from } \text{set}(b) \lor \\
   &\quad a \in \text{from } \text{set } \text{init}(b) \\
   &\quad \text{condition}(y_a \leftarrow a(v_a)) \land \\
   &\quad b \in \text{condition_true}(y_a \leftarrow a(v_a)) \lor \\
   &\quad \neg \text{condition}(y_a \leftarrow a(v_a)) \land \\
   &\quad b \in \text{condition_false}(y_a \leftarrow a(v_a)) \lor \\
   &\quad (b \in \text{next } \text{interleaved}(a) \land \\
   &\quad a \in \text{from } \text{interleaved}(b)) \\
   \end{align*}

\( \text{QED} \)

The case of interleaving is not included in the proof of lemma 17 as it is relevant only when
interleaving has started which is dealt with in cases C2 and C3, lemmas 15 and lemma 16, respectively.

**Lemma 18 (Elements of Initially-Consistent Singleton Traces are in init(R)).** An element of the singleton trace of controller fragment R is contained in the init of R:

\[
\langle b \rangle \in \text{traces}(R) \Rightarrow (b \in \text{init}(R) \lor b \in \text{init} - \text{par}(R))
\]

case A: \((c \rightarrow R)\)

1. \(\langle b \rangle \in \text{traces}(c \rightarrow R)\) initial assumption
2. \(b = c\) 1, def. traces
3. \(\text{init} - \text{par}(c \rightarrow R) = \{\}\) by def 97
4. \(c \in \text{init}(c \rightarrow R)\) def. of init 96
5. \(b \in \text{init}(c \rightarrow R)\) by 2 and 3

\[QED\]

case B: \((R_1 \sqcap R_2)\)

1. \(\langle b \rangle \in \text{traces}(R_1 \sqcap R_2)\) initial assumption
2. \(\langle b \rangle \in \text{traces}(R_1)\) or
   \(\langle b \rangle \in \text{traces}(R_2)\) 1, def. trace

sub-case B1: \((b) \in \text{traces}(R_1)\)

3. \(\text{init} - \text{par}(R_1) = \{\}\)
4. \(\langle b \rangle \in \text{traces}(R_1)\) initial assumption
5. \(b \in \text{init}(R_1)\) 3 and inductive hypothesis

sub-case B2: As sub-case B1 substituting \(R_2\) for \(R_1\)

July 6, 2008
case C \((R1 \triangle R2)\)

1. \((b) \in \text{traces}(R1 \triangle R2)\) initial assumption

2. \((b) \in \text{traces}(R1)\) or
   \((b) \in \text{traces}(R2)\) 1, def. trace

Entirely similar to the proof of Lemma 18, case B1 and B2 above.

case D \((R1 \Delta_x R2)\)

1. \((b) \in \text{traces}(R1 \Delta_x R2)\) initial assumption

2. \((b) \in \text{traces}(R1)\)

Entirely similar to the proof of Lemma 18, case B1 above.

case E \((R1 \Delta^1_x R2)\)

1. \((b) \in \text{traces}(R1 \Delta^1_x R2)\) initial assumption

2. \((b) \in \text{traces}(R1)\) or
   \((b) \in \text{traces}(R2)\) 1, def. trace

Entirely similar to the proof of Lemma 18, case B1 and B2 above.

case F \((\text{if } y \text{ then } R1 \text{ else } R2)\)

1. \((b, y_b, v_b, e_b) \in \text{traces}(\text{if } y \text{ then } R1 \text{ else } R2)\) initial assumption

   case \(y = \text{true}\)

   Entirely similar to the proof of Lemma 18, case B1 above.

   case \(y = \text{false}\)

   Entirely similar to the proof of Lemma 18, case B2 above.
6.3. Refinement and Translation to Handel-C

Refinaing should be considered where an otherwise cumbersome translation would result. Narrowing down the choice of the next operation reduces the size of the implementation, and avoids the translation process making an arbitrary choice to resolve the choice in the annotations. The first set of refinements, given in the table in Figure 6.3 replaces annotated sets with their subsets: non-determinism is reduced. Operations, like $O_{ij}$, quoted in the table are all sets. In the tables in the figures 6.3, 6.5, 6.6 and 6.7, the B comment braces are left off. The annotations are in some cases extended to deal with sets of operations even though in general they have been defined with individual operations. Extending the annotations will change the way they are translated. Sometimes a choice must be made if a set is to be translated. A choice must be made as to which set to use in the translation. In the tables the environment makes the choice by selecting an option on the command bus. In the tables the INTERLEAVED NEXT annotation is shortened to $|||$, the sequence annotation is shortened to $;$ (semicolon), and the condition next annotation is shortened to $<$. 

NEXT external choice refinement reduces the non-determinism in the choices that are offered in the next step. The INTERLEAVED NEXT refinement reduces the non-determinism in one or more branches of the interleaved execution. The NEXT sequential refinement reduces the non-determinism in one or more sections of the sequence. The NEXT conditional refinement reduces choice in a similar way.

The second refinement table given in Figure 6.5 outlines some algorithmic refinements. Sets of
### Chapter 6. Translating and Refining Annotations

**Annotation Refinement type**

1. **OPi = ...OPj NEXT**
   - **Refinement**: OPi = ...OP'j NEXT
   - **type**: next external choice refinement

2. **OPi = ...OPj || OPk**
   - **Refinement**: OPi = ...OP'j || OP'k
   - **type**: interleaved next refinement
   - **Example**: OPj1 = ...OPj NEXT...
   - **Example**: OPj2 = ...OPj NEXT...
   - **Example**: OPk1 = ...OPk NEXT...
   - **Example**: OPk2 = ...OPk NEXT...

3. **OPi = ...OPj ; OPp**
   - **Refinement**: OPi = ...OP'j ; OP'p
   - **type**: sequential next refinement
   - **Example**: OPj1 = ...OPj NEXT...
   - **Example**: OPj2 = ...OPj NEXT...
   - **Example**: OPp1 = ...OPp NEXT...
   - **Example**: OPp2 = ...OPp NEXT...

4. **OPi = ...OPj < OPp**
   - **Refinement**: OPi = ...OP'j < OP'p
   - **type**: condition next refinement
   - **Example**: OPj1 = ...OPj NEXT...
   - **Example**: OPj2 = ...OPj NEXT...
   - **Example**: OPp1 = ...OPp NEXT...
   - **Example**: OPp2 = ...OPp NEXT...

   where OP'j ⊆ OPj and
   - OP'k ⊆ OPk
   - OP'p ⊆ OPp

---

**Figure 6.3:** NEXT Refinements - Reduction of Non-determinism.
operators are again used in the annotations. When two sets of operations are put in sequence the annotations allow one from each set to be sequenced, but any of the operations in a particular set are equally able to be chosen. In case 1 a new set of operations are introduced \( OP_j \). New operations can be introduced into Event-B in subsequent refinements. In classical B new operations must be introduced beforehand as operations that implement skip. Case 1 refines a simple NEXT operation into a sequence of detailed operations. The refinement sequence must end in the original operation, which signifies the end of the refinement chain.

![Diagram](image)

**Figure 6.4.: Picturing SEQUENCE NEXT Refined Operations**

In case 2, a SEQUENCE NEXT to INTERLEAVED NEXT refinement is depicted. This refinement is possible if the operations that makes up the sequence are independent: they neither read nor write to similar variables. The function \( \text{ VARIABLE—USED} \) is introduced. It operations on sets of operators instead of a set of operations like \( \text{variable—used} \). The annotation allows any of the operations in any of the sets to be executed in parallel.

A translations guide for annotations is given in the table in Figure 6.6. This is a guide because without the knowledge of the control structure, in particular the points of recursion, a translation can not be automated. However, the annotations do differentiate between internal and external B operations, which has an impact on the final structure of the code. The CSP controller is required to get a full picture for translation and the table in Figure 6.18 and to some extent the table in Figure 6.1 illustrates how translation of the control can proceed. As mentioned, the translation of a particular annotated operator is dependent on whether the operation is an internal or external operation. Internal operations can execute immediately after invocation. The execution of an external operation must wait for external stimulus: a change in the command input bus. A wait loop is introduced to poll the appropriate input bus until an external operation invocation is detected: \( \text{wait — on —} \). Some annotated operators have restrictions on their I/O mode. External operators are marked with \( / * \text{ext} * / \). The INTERLEAVED NEXT annotation can only be associated with internal operations next. The SEQUENCE NEXT must have an external operator at the head of the sequence and internal
Chapter 6. Translating and Refining Annotations

**Table:**

<table>
<thead>
<tr>
<th>Annotation</th>
<th>Refinement</th>
<th>type</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 [OP_i \equiv ...OP_X \text{ NEXT}]</td>
<td>[OP_i \equiv ...OP_j; OP_X] [OP_i \equiv ...OP_j \text{ NEXT}\ldots] [OP_j_1 \equiv ...OP_X \text{ NEXT}\ldots] [OP_j_n \equiv ...OP_X \text{ NEXT}]</td>
<td>introduction of new operation (See Figure 6.4)</td>
</tr>
<tr>
<td>2 [OP_i \equiv ...OP_j; OP_K]</td>
<td>[OP_i \equiv ...OP_j \parallel OP_K]</td>
<td>next sequence to interleaved refinement</td>
</tr>
<tr>
<td>VARIABLE-USED(\text{OP}_j)={}</td>
<td>VARIABLE-USED(\text{OP}_K)={}</td>
<td></td>
</tr>
</tbody>
</table>

**Figure 6.5:** NEXT Refinements - Structural Refinements.

operations following. This restriction relates to the way this annotation is used in refinement. The CSP controller does not differentiate between internal and external operations. Hence the tables in figures 6.1, 6.6, 6.7, 6.17 and 6.18 may be required to obtain a translation.

In the table in Figure 6.6 a NEXT annotation with one next operation translates to a sequence of two operations. If the second operation is an internal operation then it is case 1: all its inputs are not ported. If the second operation is an external operation (all inputs are ported) then case 2 is the translation template. In the second case the controller will wait until a new command arrives then execute the external operation if it was requested. Case 3, sequential arrangement of external operations, is restricted to external operations only. A translation of a sequence that starts with one operation then has a choice of several external operations will test each input set and execute the first operation for which the input has changed since its last execution. (The new input values must be latched in.) Interleaved action is only permitted between internal operations (case 4): those that take their input from internal variables. The Handel-C par statement ensures that all the branches when complete wait until the longest (in terms of clock cycles) has completed. The conditional operator can be used for internal or external action. In the table in Figure 6.7 case 5 is the translation of the SEQUENCE NEXT. In the previous section the SEQUENCE NEXT was introduced to support refinement: a basic NEXT is refined into a sequence of SEQUENCE NEXT annotated operations. To refine an operation that has both inputs and outputs, into a sequence of operations the first operation of the sequence must input at the beginning of the sequence and the last operation must output at the end of the sequence. Case 5 does not reflect this requirement because it has been made general: the first operation in the sequence should be an external operation that inputs and the final operation is an internal operation that outputs. In the tables all the annotations are defined with sets. Strictly, the SEQUENCE NEXT annotation should be restricted to single operations as additional controller operators need to be defined to support the synthesis and translation of sets of operations (for example the CSP sequence \(;\)). However, the sequence next annotation is synthesised with CSP sequencing in the tables for illustration purposes.
<table>
<thead>
<tr>
<th>Annotation / Control fragment</th>
<th>Handel-C Translation Fragment</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OP_i \equiv \ldots OP_{j1} \text{NEXT}$</td>
<td>$y_i = OP_i(u_i) ; y_{j1} = OP_{j1}(u_{j1})$</td>
<td>internal single next translation</td>
</tr>
<tr>
<td>$OP_i \equiv \ldots {OP_{j1}} \text{NEXT}$</td>
<td>$y_i = OP_i(u_i) ;$</td>
<td>external single next translation</td>
</tr>
<tr>
<td>$/ \ast \text{ext} \ast / OP_{j1} \equiv \ldots$</td>
<td>$in = \text{wait - on - } OP_{j1} ;$</td>
<td></td>
</tr>
<tr>
<td>$op_{j1}^i y_i^i = u_i \rightarrow (op_{j1}^i y_{j1}^i = u_{j1} \rightarrow \ldots$</td>
<td>$if \ in == \ OP_{j1} \ then \ y_{j1} = \ OP_{j1}(u_{j1})$</td>
<td></td>
</tr>
<tr>
<td>$op_{j}^i y_i^i = u_i \rightarrow (op_{j}^i y_{j}^i = u_{j} \rightarrow \ldots$</td>
<td>$else \ delay ;$</td>
<td></td>
</tr>
<tr>
<td>$/ \ast \text{ext} \ast / OP_i \equiv \ldots$</td>
<td>$y_i = OP_i(u_i) ;$</td>
<td>external multiple next choice translation</td>
</tr>
<tr>
<td>$OP_j \text{NEXT}$</td>
<td>$in = \text{wait - on - } OP_j ;$</td>
<td></td>
</tr>
<tr>
<td>$op_{j}^i y_i^i = u_i \rightarrow (op_{j}^i y_{j}^i = u_{j} \rightarrow \ldots$</td>
<td>$if \ in == \ OP_{j} \ then \ y_{j} = \ OP_{j}(u_{j})$</td>
<td></td>
</tr>
<tr>
<td>$\square$</td>
<td>$else \ ...$</td>
<td></td>
</tr>
<tr>
<td>$op_{j}^i y_i^i = u_i \rightarrow (op_{j}^i y_{j}^i = u_{j} \rightarrow \ldots$</td>
<td>$if \ in == \ OP_{j} \ then \ y_{j} = \ OP_{j}(u_{j})$</td>
<td></td>
</tr>
<tr>
<td>$op_{j}^i y_i^i = u_i \rightarrow (op_{j}^i y_{j}^i = u_{j} \rightarrow \ldots$</td>
<td>$else \ skip$</td>
<td></td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$OP_j \equiv \ldots OP_{X} \text{NEXT}$</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>$OP_k \equiv \ldots OP_{X} \text{NEXT}$</td>
<td>}</td>
<td></td>
</tr>
<tr>
<td>$op_{j}^i y_i^i = u_i \rightarrow (op_{j}^i y_{j}^i = u_{j} \rightarrow \ldots</td>
<td></td>
<td>$</td>
</tr>
<tr>
<td>$</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.6: NEXT Annotation Translation Guide Part 1.
### Chapter 6. Translating and Refining Annotations

<table>
<thead>
<tr>
<th>Annotation/Control fragment</th>
<th>Handel-C Translation</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>$OP_i \equiv ...OP_j ; OP_K$</td>
<td>$y_i = OP_i(v_i)$, $in = \text{wait} - on - OP_j$ if $in = OP_j$ then $y_j = OP_j(v_j)$ else ...</td>
<td>next sequential translation</td>
</tr>
<tr>
<td>/* ext */$OP_{j1} \equiv ...OP_K NEXT ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>/* ext */$OP_{jn} \equiv ...OP_K NEXT ...</td>
<td></td>
<td></td>
</tr>
<tr>
<td>$OP_{k1} \equiv ...$</td>
<td>if $in = OP_{j_n}$ then $y_j = OP_{j_n}(v_{j_n})$ else skip</td>
<td></td>
</tr>
<tr>
<td>$OP_{kn} \equiv ...$</td>
<td>$y_{k1} = OP_{k1}(v_{k1})$, $in = \text{wait} - on - OP_K$</td>
<td></td>
</tr>
<tr>
<td>$op_i?y_i?v_i \rightarrow (op_j?y_j?v_j \rightarrow ...)$</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>else ($op_{k1}?y_{k1}?v_{k1} \rightarrow ...$)</td>
<td></td>
</tr>
<tr>
<td></td>
<td>else ($op_{kn}?y_{kn}?v_{kn} \rightarrow ...$)</td>
<td></td>
</tr>
</tbody>
</table>

Figure 6.7.: NEXT Annotation Translation Guide Part 2.
In the translation a command bus is added to make the appropriate choices. The external environment will make the choice.

The translations of Stepney [Ste03], and Phillips and Stilles [PS04] are given in table 6.1. Only the translation of integer declaration, parametrisable functions, and recursion are used. This is because our source is not CSP (it is annotated B and CSP) as such channels are not being used to synchronise events. In the table the CSP language construct and translation are mapped. A tick indicates if they are supported by Stepney (SS) or by Phillips and Stilles (PS). When an operation is invoked it takes its input from the environment from the port. Internal synchronisation of operations within machines is not dealt with in this chapter. To guide the B translation the table in Figure 6.17, on page 175, has been developed. A discussion of the example is given in section 6.4.

<table>
<thead>
<tr>
<th>Feature</th>
<th>CSPM</th>
<th>Handel-C</th>
<th>PS</th>
<th>SS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Channel Declarations (from use)</td>
<td>channel chan chanin chanout</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Channel Declarations</td>
<td>channel c chan SYNC c;</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Typed Structured Channel Declarations</td>
<td>channel d: T.T chan struct d:DATA d</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Input Channel Operations</td>
<td>in?x</td>
<td>in?x;</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Output Channel Operations</td>
<td>outlx</td>
<td>outlx;</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Integer Declarations</td>
<td>int 8 x;</td>
<td>void(n)...</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Parametrisable functions</td>
<td>p(n) = ...</td>
<td>void(n)...</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>External Choice</td>
<td>[]</td>
<td>prialt ...</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Synchronous Parallel</td>
<td>[ ( ... ) ]</td>
<td>par ...</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Replicated Sharing Parallel</td>
<td>[ Event ] n: { i..j } P(n) par (n=i; n=j; ++n)P(n);</td>
<td>✓</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Recursion</td>
<td>P = ... → P</td>
<td>while(1) ...</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Conditional Choice</td>
<td>if b then P else Q</td>
<td>if (B) then P(); else Q();</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Macros</td>
<td>{- ... -}</td>
<td>...</td>
<td>✓</td>
<td></td>
</tr>
</tbody>
</table>

6.4. Example: Safe Control System

We use the example of a safe locking system to illustrate the ideas introduced in this chapter. The specification outlines the operations offered to the environment. The operations that can be invoked by the environment are indicated with /* ext */ annotations. Both the operation
Chapter 6. Translating and Refining Annotations

and the output can be marked with a /* ext */. All /* ext */ annotation outputs are ported and become part of the Handel-C output interface. All /* ext */ operations are associated with a bus port that has a state of the same name as the operation. Variables intended as input are marked with /* IN */. It is possible to mark the variables as /* IN */ or /* OUT */. Along with the mode the width of the type is given in bits. Operations are invoked in two ways. The first way is via the command bus, which when set to the operator name will invoke the operation when it is enabled by the control flow. Operations not labelled with /* ext */ are internal and are invoked immediately when enabled by the control flow.

6.4.1. The Example’s State and Control Flow

In Figure 6.8 and Figure 6.9 the B Abstract Machine for a safe cabinet design is given. There are three command states Locked, Unlocked, and BrokenOpen which are represented in two bits. The variable Door is drawn from the COMMAND type and initialised to Unlocked. The Lock operation is enabled after initialisation. It is an external operation with externally ported output. After setting the Door state variable to Locked both Unlocked and BreakOpen are enabled. For completeness we introduce two operations that will be used later to develop the detailed functionality of the machine during refinement. These operations are UnlockR1 and UnlockR2. Their bodies are not expanded. The Unlock is an external operation and has externally ported output. It represents an attempt to unlock the safe. It non-deterministically decides to set the Door variable to Unlocked or Locked. The next operator to be enabled depends on the outcome of the Unlock operation. If the door became Unlocked then the next enabled operation would be Lock, otherwise Unlocked or BreakOpen will be offered. The BreakOpen operation sets the Door state to BrokenOpen and offers itself as the next operation available.

The controller CTRL given in Figure 6.10 first performs a Lock and then jumps to the S process where it can perform either an Unlock or BreakOpen. The Unlock event has a single output that is used as the conditional test in the if-then-else following the Unlock event. If the output of the Unlock operation is true then a Lock event occurs and the flow of control is repeated starting again at S, if it is false then control is repeated with S.

6.4.2. A Refined Example

A refinement of the Safe machine, called SafeR, is given in Figure 6.11 and Figure 6.12. The example mimics refinement in Event-B. The operation UnlockR1 and UnlockR2 are introduced to refine Unlock. The laws of refinement of Event-B are not fully justified in this thesis. The Safe refinement, SafeR, breaks down the unlocking process into two stages. Firstly, two new operations are slotted into the control interleaved: UnlockR1(Comb1a, Comb1b) and UnlockR2(Comb2a, Comb2b). Both have a combination parameter which is compared against a stored master code and a second parameter that is used to create a new master key. The UnlockR commands update the master combination if a successful comparison occurs. New input variables are added: Cx1a, Cx2a, Cx1b, and Cx2b. These represent the interface with the environment on which new values are input. These are used to input the combination values and are not used by the B Operations. Checked1, Checked2, Master1 and Master2 are new variables used by the operations. The annotations of the Lock operation
are refined. Two operation are added before the Unlock. The extra proof obligations are discharged automatically. The bodies of the UnlockR1 and UnlockR2 are completed at this level. The body of the Unlock operation is refined to set the output and the Door variable. The annotations of the Unlock are refined: the BreakOpen operation is removed from the next annotation as an option, which means it can not be reached. What was one unlock operation has been expanded into three (two interleaved). Before refinement the Unlock operation has both input and output. The refined version has the input occurring on the first operations in the refined sequence of operations (UnlockR1 and UnlockR2), and the output occurring on the final operation of the sequence (the original Unlock operation).

The controller given in Figure 6.13 begins like the abstract controller with a Lock event then a jump to $S$. There is, in this refined process, no choice to break open only UnlockR1 and UnlockR2 are offered with $Ca1a$ and $Ca1b$, and $Ca2a$ and $Ca2b$ as an input to them, respectively. The Unlock process is refined. The refined sequence starts with an interleaved combination of UnlockR1 and UnlockR2 events then the original Unlock event. As before the outcome of Unlock determines what happens next. If the Unlock is successful the process will be restarted from the beginning. If the current attempt at unlocking fails then another attempt at Unlock will occur. It is noted that the Lock $\rightarrow S$ could have been replaced by CTRL, but the current approach separates it from the main CTRL. The former is easier to translate. The types of the outputs are not illustrated, for example the CTRL process would be:

$$\text{CTRL} = \square y : \{\text{Locked}\} \circ \text{Lock}!y \rightarrow S$$

6.4.3. A Hand Translation using the Guidance

A summary of the hand translations from the refined B specification is given in the table in Figure 6.17. The B provides the details of the types, variables, and functions. The CSP controller provides the executions details that are use later to construct the Handle-C main section. A summary of the hand translation of the controller is given in the table in Figure 6.18.

The SETS clause is translated into an enumerated type. The INVARIANT section is used to create the declarations. Variables annotated with a mode are created as buses of the appropriate I/O type and size. Other variables are created. In the tables only translation of N (or subset of N) or enumerated types are considered. Variables bound to ports are created. To permit external operation calls a bus is created and given the same name as the machine. The width of the bus must be sufficient to handle the number of external operations of the machine. The mechanism for requesting an external operation to execute is to change the data on the command input bus to the same name as the operation required. The last requested operation is latched into variable of the same name as the machine with a -var post fix. Variables are declared for operation outputs. The name of an output bus variable is the concatenation of the operation output name and the operation name. This avoids clashes. Buses are defined for each */IN*/ and */OUT*/ annotation, external operation */ext*/ and operation output. Operations are translated into functions. If an operation has an output the function will return a value. Functions with outputs will have an assignment in them that assigns to the bus output function variable. The function will also return that
output in the final statement of the function. Assigning to the function output variable allows results to be chained (feed into other functions). The details of the translation of the B operators is only given for assignment and if - then - else. There are existing mappings to C from a subset of B0 in the B Method. It is therefore assumed that a translation of at least a subset of B to Handel-C is possible. The bodies are translated in a straightforward manner. Assignments in the operations are put together in a par Handel-C statement. Assignments and the if - then - else B constructs have straightforward translations. The INITIALISATION is translated into a function called Initialisation-fnc. The \(< < B >>\) represents the translation of B fragment.

The CSP controller is used to construct the main Handel-C body. A summary of the hand translations made on the CSP controller is given in the table in Figure 6.18. The controller design was structurally limited to facilitate translation: initialisation and setting up operations are performed before a main loop is entered. Hence, the first process definition \(CTRL-fnc\) is not recursive; it is an open process. It translates to a function call \(CTRL-fnc\), which invokes the Initialisation-fnc and lock-fnc functions. On returning to the main program the next function called is the \(S-fnc\), which implements the main loop. \(S-fnc\) is tail recursive and is implemented with a continuously looping while loop; it is a closed process. The first event in the main loop is the UnlockR commands. In the translation the UnlockR-fncs are preceded by wait-UnlockR-fncs as they are external operation. The UnlockR-fnc functions inputs from the \(Cz1a, Cz1b, Cz2,\) and \(Cz2\) input buses. The Unlock-fnc call follows. Unlock-fnc returns a value that is assigned to a variable that is output ported. The value is also used to decide the course of the following if - then - else. Either a Lock-fnc or an UnlockR-fnc is performed after a wait. Then the process recurses. The \(< P >\) represents translation or CSP fragments. The translation of the CSP choice operator is also a refinement.
MACHINE Safe

SETS COMMAND = { Locked, Unlocked, BrokenOpen }/*2*/

VARIABLES Door

INvariant Door \in COMMAND /*OUT2*/

initialisation Door := Unlocked /* { Lock } NEXT */

operations

/\text{*ext*}/ Status \leftarrow /\text{*ext*}/Lock \triangleq 

\text{pre} 

\hspace{1cm} Door = \text{Unlocked}

\text{then} 

\hspace{1cm} Door := \text{Locked}||\text{Status}:= \text{Locked}

\text{end}

/\text{* Unlock, BreakOpen } NEXT */ ;

UnlockR1(\text{Comb1a, Comb1b}) \triangleq 

\text{pre} 

\hspace{1cm} \text{Comb1a} \in \text{NAT} \land \text{Comb1b} \in \text{NAT}

\text{then} 

\hspace{1cm} \text{skip}

\text{end} ;

UnlockR2(\text{Comb2a, Comb2b}) \triangleq 

\text{pre} 

\hspace{1cm} \text{Comb2a} \in \text{NAT} \land \text{Comb2b} \in \text{NAT}

\text{then} 

\hspace{1cm} \text{skip}

\text{end} ;

Figure 6.8.: Safe Machine - Part 1
Chapter 6. Translating and Refining Annotations

Figure 6.9.: Safe Machine - Part 2

/*ext*/ Status ← /*ext*/ Unlock ≡
PRE
Door = Locked
THEN
ANY
dd
WHERE
dd: COMMAND - { BrokenOpen }
THEN
IF
(dd = Unlocked)
THEN
Status := 1
ELSE
Status := 0 || Door := dd
END
END
END
/*{Lock } {UnLock,BreakOpen } CONDITIONAL NEXT */ ;

/*ext*/ Alarm ← /*ext*/ BreakOpen ≡
PRE
Door ∈ COMMAND
THEN
Door := BrokenOpen || Alarm := 1
END
/*{BreakOpen } NEXT */
END
CTRL \(= \Box \text{Lock}_y \rightarrow S\)

\(S = (\Box \text{Unlock}_y \rightarrow (\text{if } y == 1 \text{ then } \Box \text{Lock}_y \rightarrow S \text{ else } S))\)

\((\Box \text{BreakOpen}_y \rightarrow B_{-CTRL})\)

\(B_{-CTRL} = \Box \text{BreakOpen}_y \rightarrow B_{-CTRL}\)

Figure 6.10.: Safe Machine Controller.
REFINEMENT SafeR
REFINES Safe
VARIABLES Door, Cx1a, Cx2a, Cx1b, Cx2b, Master1, Checked1, Master2, Checked2

INVARIANT
Cx1a ∈ NAT/*IN16*/ ∧ Cx2a ∈ NAT/*IN16*/ ∧
Cx1b ∈ NAT/*IN16*/ ∧ Cx2b ∈ NAT/*IN16*/ ∧
Master1 ∈ NAT/*16*/ ∧ Checked1 ∈ NAT/*1*/ ∧
Master2 ∈ NAT/*16*/ ∧ Checked2 ∈ NAT/*1*/

INITIALISATION
Door := Unlocked || Cx1a := 0 || Cx2a := 0 || Cx1b := 0 || Cx2b := 0 ||
Master1 := 67 || Checked1 := 0 || Master2 := 78 || Checked2 := 0 /* { Lock } NEXT */

OPERATIONS

/*ext2*/Status ← /*ext1*/Lock ∪
PRE
Door = Unlocked
THEN
Door := Locked || Status := Locked || Checked1 := 0 || Checked2 := 0
END
/* { UnlockR1(Combla, Comb1b) } UnlockR2(Cx1a, Cx2a, Cx2b) } { Unlock } SEQUENCE NEXT */
/* { UnlockR1(Combla, Cx1b) } UnlockR2(Cx1a, Cx2a, Cx2b) } INTERLEAVED NEXT */

/*ext1*/UnlockR1(/*16*/Combla, /*16*/Comb1b) ∪
PRE
Door = Locked
THEN
IF
(Combla = Master1)
THEN
Checked1 := 1 || Master1 := Comb1b
ELSE
Checked1 := 0
END
END /* { Unlock } NEXT */

Figure 6.11.: Safe Refinement - Part 1.
/*ext*/ UnlockR2(/*16*/Comb2a,/*16*/Comb2b) ≡
PRE
Door = Locked
THEN
IF
(Comb2a = Master2)
THEN
Checked2 := 1 || Master2 := Comb2b
ELSE
Checked2 := 0
END
END /* { Unlock } NEXT */ ;

/*ext*/ Status ← Unlock ≡
PRE
Door = Locked
THEN
Status := Checked2 ||
IF
(Checked1 = 1) ∧ (Checked2 = 1)
THEN
Door := Unlocked
ELSE
Door := Locked
END
END /* { Lock } { UnlockR } NEXT CONDITION */ ;

/*ext*/ Alarm ← /*ext*/ BreakOpen ≡
PRE Door ∈ COMMAND THEN Door := BrokenOpen || Alarm := 1 END
/* { BreakOpen } NEXT */
END

Figure 6.12.: Safe Refinement Part 2
\[ CTRL = □ Locky \rightarrow S \]
\[ S = (UnlockR1?Cz1a?Cz1b \rightarrow \text{skip} || UnlockR2?Cz2a?Cz2b \rightarrow \text{skip}); □ Unlocky \rightarrow (if y then □ Locky \rightarrow S else S) \]

Figure 6.13.: Refined Safe Controller

//set clock = external "Clock"
#define PAL_TARGET_CLOCK_RATE 25175000
#include "pal_master.hch"  
// BreakOpen removed in translation as      
// not used and no command default added
typedef enum {Not_Commanded =  
    (unsigned 2) 0, Locked, Unlocked} COMMAND; //
typedef enum {No_Command =  
    (unsigned 2) 0, Lock, UnlockR1, UnlockR2} SafeR;
unsigned 2 Door; // B variables
unsigned 1 Checked1; //
unsigned 16 Master1; //
unsigned 1 Checked2; //
unsigned 16 Master2; //
SafeR SafeR_Bus_var; // latch input bus values to
// request operation execution
unsigned 1 Status_Unlock; // operation output values
unsigned 2 Status_Lock; //
interface bus_in(unsigned 16 inp) Cx1a(); // IN annotations
interface bus_in(unsigned 16 inp) Cx2a(); //
interface bus_in(unsigned 16 inp) Cx1b(); // IN annotations
interface bus_in(unsigned 16 inp) Cx2b(); //
interface bus_in(SafeR 2 inp) SafeR_Bus(); // ext operations
interface bus_out() Door1 (unsigned 2 OutPort=Door); // OUT annotations
interface bus_out() //
    Status_Unlock1 (unsigned 1 OutPort=Status_Unlock); //
interface bus_out() //
    Status_lock1 (unsigned 2 OutPort=Status_Lock); //

Figure 6.14.: SafeR Translation Part 1.
void wait_on_Lock_fnc(void){
    while (SafeR_Bus.inp != Lock){delay;}
    SafeR_Bus_var = Lock;
}

unsigned 2 Lock_fnc(void){
    par{
        Door = Locked,
        Status_Lock = Locked
    };
    return Locked;
}

void wait_on_UnlockRl_fnc(void){
    while (SafeR_Bus.inp != UnlockRl){delay;}
    SafeR_Bus_var = UnlockRl;
}

void UnlockRl_fnc(unsigned 16 Comb1a; unsigned 16 Comb1b){
    if (Comb1a == Master1) {
    par{Checked1 = 1,
         Master1 = Comb1b}
    } else
    {Checked1 = 0;}
}

void wait_on_UnlockR2_fnc(void){
    while (SafeR_Bus.inp != UnlockR2){delay;}
    SafeR_Bus_var = UnlockR2;
}

void UnlockR2_fnc(unsigned 16 Comb2a; unsigned 16 Comb2b){
    if (Comb2a == Master2) {
    par{
         Checked2 = 1,
         Master2 = Comb2b
    };
    } else
    {Checked2 = 0;}
}

Figure 6.15.: SafeR Translation Part 2.
unsigned 1 Unlock_fnc(void){
    par{
        if (Checked2 == 1) {Door = Unlocked;} else {Door = Locked;},
        Status_Unlock = Checked
    }
    return Checked;
}
void Initialization_fnc(void){
    Checked = 0;Mater = 67;Door = Unlocked; // INITIALISATION
    Status_Lock = 0;Status_Unlock = 0;       // SET OUTPUT DEFAULT
}
void CTRL_fnc(void){
    Initialization_fnc(); wait_on_Lock_fnc();
    if (SafeR_Bus_var == Lock){Status_Lock = Lock_fnc();}else{delay;}
}
void S_fnc(void){
    while(1){par{
        seq{wait_on_UnlockR1_fnc();
            if (SafeR_Bus_var==UnlockR1)
                UnlockR1_fnc(Cx1a.inp,Cx1b.inp);}
            else
                {delay;}
        } // seq
        seq{wait_on_UnlockR2_fnc();
            if (SafeR_Bus_var==UnlockR2)
                UnlockR2_fnc(Cx2a.inp,Cx2b.inp);}
            else
                {delay;}
        } // seq
    } // par
    Status_Unlock = Unlock_fnc();
    if (Status_Unlock){
        wait_on_Lock_fnc();
        if (SafeR_Bus_var==Lock){
            Lock_fnc();
        }else
            {delay;}
    }else
        {delay;}
    } //while
}
void main(void){
    CTRL_fnc();S_fnc();
}

Figure 6.16.: SafeR Translation Part 3.
### 6.4. Example: Safe Control System

<table>
<thead>
<tr>
<th>Feature</th>
<th>B Declaration</th>
<th>Handel-C</th>
</tr>
</thead>
<tbody>
<tr>
<td>set declaration</td>
<td>( \text{SETS SS} = AA,\ldots,XX/\ast n/ )</td>
<td>typedef enum { AA = (unsigned n) 0, \ldots, XX } SS;</td>
</tr>
<tr>
<td>B variable declaration</td>
<td>( \text{INVARIANT} \ V_v \in \text{T} /\ast \text{OUTn}/ )</td>
<td>unsigned n V_v;</td>
</tr>
<tr>
<td></td>
<td>( \text{INVARIANT} \ V_v \in \text{T} /\ast \text{INn}/ )</td>
<td>interface bus_out()</td>
</tr>
<tr>
<td></td>
<td>( \text{INVARIANT} \ V_v \in \text{T} /\ast n/ )</td>
<td>Vv1 (unsigned 2 OutPort=Vv);</td>
</tr>
<tr>
<td>Command Bus Declaration</td>
<td>( \text{MACHINE mname typedef enum} { \text{No_Command=(Unsigned k) 0} \ldots, \text{op_k} } ) \ mname;</td>
<td>\text{interface bus-in(}unsigned \text{n inp) Vv();}</td>
</tr>
<tr>
<td>with k operations</td>
<td>\text{mname mname_Bus_var;} interface bus_in}</td>
<td>\text{unsigned n Vv;}</td>
</tr>
<tr>
<td>Function Declaration</td>
<td>/<em>ext N</em>/ \ Oo \leftarrow /<em>ext</em>/ Cc(/* M */Zz)</td>
<td>\text{interface bus_out() Oo_Cc1 (unsigned N Outport=Oo_Cc);}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>void wait_on_Cc_fnc() {</td>
</tr>
<tr>
<td></td>
<td></td>
<td>while (mname_Bus.inp != Cc) { delay; }</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(mname_Bus_var = Cc;</td>
</tr>
<tr>
<td></td>
<td></td>
<td>}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{unsigned N Cc_fnc(unsigned M Zz){}</td>
</tr>
<tr>
<td></td>
<td></td>
<td>\text{par{..};return exp;}}</td>
</tr>
<tr>
<td>Function Body</td>
<td>\text{&lt;&lt;PRE P THEN B END&gt;&gt;} \par{&lt;&lt; B &gt;&gt;}</td>
<td>\text{if &lt;&lt; b &gt;&gt; { &lt;&lt; c &gt;&gt; }}</td>
</tr>
<tr>
<td></td>
<td>\text{&lt;&lt;IF b THEN c ELSE d END&gt;&gt;} &lt; &lt; b &gt;&gt; = &lt; &lt; c &gt;&gt; ;}</td>
<td>\text{else { &lt;&lt; d &gt;&gt; };}</td>
</tr>
<tr>
<td></td>
<td>\text{initialisation &lt;&lt;INITIALISATION ...&gt;&gt; \ void Initialisation(\text{void}{ &lt;&lt; \ldots &gt;&gt; };}</td>
<td></td>
</tr>
<tr>
<td></td>
<td>\text{main &lt;&lt;OPERATION&gt;&gt;} \ see table 6.18</td>
<td>\text{}</td>
</tr>
</tbody>
</table>

**Figure 6.17.: B to Handel-C Translation Guide.**
### Chapter 6. Translating and Refining Annotations

#### Figure 6.18.: CSP to Handel-C Translation Guide.

<table>
<thead>
<tr>
<th>Feature</th>
<th>CSP</th>
<th>Handel-C</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>initialisation processes</strong></td>
<td>$P = \ldots S$</td>
<td>void Initialisation_fnc(void);...; void CTRL_fnc(void){Initialisation_fnc();...;}</td>
</tr>
<tr>
<td><strong>main loop processes</strong></td>
<td>$S = \ldots S$</td>
<td>void S_fnc(void){while(1){...;}}; void main(void){CTRL_fnc();S_fnc();}</td>
</tr>
<tr>
<td><strong>prefix (internal)</strong></td>
<td>$&lt; e \rightarrow P &gt;$</td>
<td>e_fnc ; $&lt;P&gt;$</td>
</tr>
<tr>
<td><strong>prefix (external)</strong></td>
<td>$&lt; e \rightarrow P &gt;$</td>
<td>wait-on_e; e_fnc ; $&lt;P&gt;$</td>
</tr>
<tr>
<td><strong>choice (external)</strong></td>
<td>$&lt; P_1 \bigcirc P_2 &gt;$</td>
<td>$&lt;P_1&gt;$</td>
</tr>
<tr>
<td><strong>interleaved</strong></td>
<td>$&lt; \text{e}_1 \rightarrow \text{skip} \mid \ldots \mid \text{e}_n \rightarrow \text{skip} \mid P &gt;$</td>
<td>par{$&lt; \text{e}_1 \rightarrow \text{skip} &gt;$; \ldots; $&lt; \text{e}_n \rightarrow \text{skip} &gt;$}; $&lt;P&gt;$</td>
</tr>
<tr>
<td><strong>if-then-else</strong></td>
<td>$&lt;$if $y$ then $P$ else $Q &gt;$</td>
<td>if $y$ {$&lt;P&gt;$} else {$&lt;Q&gt;$}</td>
</tr>
</tbody>
</table>

176

JULY 5, 2008
7. Comparisons with Other Work

7.1. Associated Technologies

The design and implementation of critical systems benefit from development in a formal method such as B. However, B does not support execution specification directly. Event-B [MAV05] and CSP\|B have been proposed as a means to create a formal specification notation that incorporates action specification and B. Each of these approaches has particular benefits. Core to the Event-B approach is refinement, whereas CSP\|B offers a clean separation of control and data manipulations.

Combining model-based formalisms like B [Abr96] and Z [Spi92] with event-based formalisms like CSP [Sch00] is an active research area. The motivation for a closer integration is to obtain a formalism that is as powerful at describing state predicates as it is at describing event behaviour. The CSP\|B [Tre00] approach combines CSP and B so that CSP captures, primarily, the event aspect of the design, whereas the B captures the state evolution. Each CSP controller directs a single B machine via communication channels. Controllers may interact with other controllers. In Treharne [Tre00] consistency between the pre-conditioned B machine and the CSP controllers is established in two ways. Firstly, by showing that operations are always called within their preconditions, which establishes divergence freedom. Guarded controllers present the possibility of controller deadlock. The secondly, consistency condition establishes that controllers are deadlock free. Consistency is investigated using the wps of guarded action [Di76].

A number of tools focus on capturing the designs of safety critical systems using state machines [And03]. State machine designs are in a form amenable to formal reasoning and simulation. They can readily be translated into both software and hardware languages. The Estere\textsuperscript{1} tool produces C, System-C\textsuperscript{2} and VHDL [Ash96]. Object orientation and complex concurrent behaviour are avoided in high integrity designs [MoD99] to ease the burden of verification.

Bridging the semantic gap between software and hardware is important to ensure proven specifications are implemented correctly. Research into formalising Hardware Description Languages (HDL), like VHDL [BKLM97] and Handel-C [BW03] has provided semantic models of HDLs. The translation mappings between FDR2 and Handel-C [Ste03] illustrated the common ground that exists between process algebras and HDLs. Research translating BVHDL to VHDL [ISS01] provided evidence of the commonality of specification languages and HDLs. Event-B has been used to describe hardware circuits [Abr01]. Recent work on using BHDL

\textsuperscript{1}http://www.estere-technologies.com/
\textsuperscript{2}http://www.systemc.org/
to model VHDL [ABD+03] supports the belief that direct compilation to hardware from a formal notation is possible. The "Future Technology" project involves collaborative research into the use of CSP||B to specify co-designs [MS06] [MS07] and to formally investigate systems designs of large scale developments captured in xUML [TSG+07]. The HDL for this work is Handel-C.

Boulanger's et al. [ABD+03] work modelling VHDL in B builds from the bottom up. They develop systems on top of multi valued logic of the IEEE libraries of standard logic vectors specified in B. The problem with this approach is that large circuits are difficult to analyse. Abrial approaches the problem from a higher level, than that of Boulanger et al. Refinement is used to achieve the development of larger components from logic gates. However, no details of mappings to synthesizable logic are given. The work by B-Core and Hill [ISSOl] was aimed at developing larger circuits for industrial applications, so it focused on modelling the structure of the design at the expense of providing a semantic underpinning for BVHDL. Recently, more abstract HDLs have entered the market. SystemC and Handel-C® allow the capture of circuit descriptions at a higher level than VHDL. Stepney's work defines a set of mappings to Handel-C from FDR2, but the mappings are not supported by a common semantics. The work highlights the difficulties of representing state, a very important aspect of hardware designs, in CSP. The aim of this thesis is to extend B with annotations to allow it to capture temporal properties, then to translate directly into Handel-C, but to support the translation process with a common semantics. For simplicity the designs are restricted to finite state machine designs. We note the criticism of Hilton [Hil04] that Handel-C being based on C is fundamentally unsafe. However, work on developing an operational semantics for Handel-C [BW03] may allow it to be used safely. It should also be noted that a safe subset of C [Hat95] have been reported on and used, particularly in the motor industry [Mot04]. So initially, Handel-C for a HDL appears to be a good choice.

From the discussion above it is evident that there are a number of other works that warrant comparison to the findings of this thesis, because of the large span that the thesis covers. In this chapter approaches to hardware verification are considered. These range from semi-formal to fully formal verification. In Section 7.2 the methodologies are outlined and notable verifications are cited, along with a discussion about the limitations associated with the approaches. The research undertaken at the start of the thesis in to the use of BVHDL as a HDL is reviewed in section 7.3. Formal state machine development is considered, namely the approach based on Esterel in section 7.4. Although, this approach is fully formal it does not deal with data processing as well. There are other formal method integrations apart from CSP||B. CSP-Z is an approach based on Z and CSP, and it has been compared to the approach using annotations in section 7.5. Event-B is a mature technology although on its own it has some of the disadvantages that classical-B has when compared to CSP||B. However, Event-B research has covered many areas that can be compared to the work of the thesis, for example there has been work on hardware modelling, variant termination, modalities and state machine modelling, which is discussed in sections 7.6 to 7.7. Annotations are strongly related to temporal logic. B model checking and more recently CSP model checking have moved forward in a similar direction as has the annotations research. A major limitation with the model checking approach is the state space explosion problem. Work on model checking B and CSP using ProB is considered in section 7.8. There have been other approaches to the
translation of CSP to Handel-C that are reviewed and compared to the annotation translation approach in section 7.9. Approaches when targeted at software obtain regulator acceptance; namely those based around Spark Ada. An approach that incorporates both software (SPARK Ada) and hardware that there is not space to detail in full, but deserves a mention, was proposed by Hilton [Hil04]. On the hardware side Hilton proposes that FPGA design should be captured in Synchronous Receptive Process Theory [Joe92] (SRPT). SRPT has a denotational semantics defined in failures-divergence of process and distinguishes between input and output events unlike Hoare’s CSP semantics. Their proof of requirement is in terms of verifying the sequence of events of a FPGA cell (specified in SRPT) against higher-level behavioural requirement sequences. A netlist cannot be obtained directly from SRPT. The SRPT model has to be translated into Pebble [LM98] to obtain a netlist.

7.2. Hardware Verification and Hardware Description Languages

In this thesis a formal approach to hardware development is taken. Although formal approaches to hardware development is not new, they remain rare in new developments. Informal approaches are characterised by the development of various models that culminate in an implementation, where the models of the system are not related by mathematical reasoning. Equivalence between high and low level models is demonstrated by testing. Semi-formal methods may employ formal verification between some key levels of abstraction, or may use formal model checking to establish key model properties at some levels of the development. What actually exist today is a large number of approaches across the spectrum that covers fully formal hardware development in research to informal development in industry. No fully formal commercial development approaches exist. All approaches can be augmented with some form of model checking to make them semi-formal.

Hardware development approaches are categorised by their language. Choosing an approach is a matter of choice of hardware description language, simulation tools and synthesis tools. The tools are generally multi-language. Principle Hardware Description Languages (HDL) included VHDL, Verilog [IEEE03] SystemVerilog®, System-C [IEEE05a], C++, C, Handel-C®. VHDL (Very high speed integrated circuit Hardware Description Language) has several active standards in use for different VHDL language variants. The IEEE Standard VHDL Language Reference Manuals IEEE Std 1076 have had issues in 1987, 1993, 2000, 2002 [IEEE02]. Ashenden [Ash96] offers guidance on how to construct VHDL circuits for synthesis. MentorGraphics™ are one of the many EDA suppliers of tools that support the development, debugging, and synthesis of VHDL. One particular tool of note is Modelsim™, which supports developments using the major HDLs: Verilog, SystemVerilog, VHDL and SystemC.

An informal approach requires the development of a system and a test bench. The test bench is used to exercise the design to demonstrate conformity to requirements under simulation by tools like Modelsim. The coverage effectiveness of the test bench can be evaluated with tools

July 6, 2008
like VN-Cover™ developed by Transeda™. Testing is complete when a target coverage is met. MentorGraphics™ support semi-formal design approaches with assertion based verification. Assertion base verification tools dynamically monitor simulations and check property conformance. Large EDA supplies tool's target the major languages. Generally, if the tool can input VHDL it can handle Verilog, and System-C.

A step up in terms of formality is model checking, or as normally referred to in the EDA market property checking. This entails developing a finite model of the HDL circuit and checking the model exhaustively for properties. Property checking is transformed into a problem search that uses both property and circuit reduction techniques to speed up the search. Either the property is established or a counter example is found and displayed. Termination of the checking search is not guaranteed in models that are difficult to reduce or are large. Averant™ have developed a static functional verification tool called Solidify™ [SS06] that performs property checking on VHDL and Verilog clocked designs. Properties can be written in a variety of languages like PSL [IEE05b] (Property Specification Language), or HPL (Hardware Property Language) their own proprietary language. The tool supports a number of automatic checks for dead code, deadlock states (from a non-reset state). Properties may be formulated over a finite clock period to reduce the size of the search space. It is unrealistic to expect properties to be check over a clock period exceeding 80 clock cycles. Property verification is more limited than simulation and the Solidify tool is limited to several thousands of gates [Raw01]. Formal equivalence checking, offered by the Prover eCheck tool[11], between RTL designs and FPGA layout ensures that there is no logical difference between the RTL and FPGAs netlist. Demonstrating the equivalence is a way of showing that the translation from RTL to FPGA netlist is correct.

The success of property checking is probably based on the automatic nature of the proof effort. Property checking and more generally model checking is an approach that augments an informal approach to create a semi-formal approach. Modern property checkers like Solidify and the pioneering model checker like FormalCheck [BG01] analyses the source language directly saving effort translating to an intermediate language to undertake the analysis. In the same way that property checking augments the development process so does formal equivalence checking. Non-commercial model checkers and model checking algorithms are described in a number of texts. The SMV™ [McM92] model checker was the first incarnation of FormalCheck. The SMV model checker does not take VHDL as input instead it takes a C like bespoke input language. Model checking algorithms utilised by SMV and examples SMV programs are available [HR00] [BBF+01] (in depth treatment of algorithms supporting model checking are reported on by Clark [Cla]). The texts introduce temporal logic properties like liveness, safety, deadlock freedom, reachability and fairness; and reasoning frameworks like Binary Decision Diagrams (BDD) [Bry85]. The BDD reduce the state space testing for specified properties. BDD and in particular OBDD [Bry92] (Ordered BDD) were extensively used in symbolic model checking to handle large designs [BCM+90]. If the property is disproved then a counter example is given. NuSMV[12] appeared in 2002 as an open source re-implementation of SMV that included SAT [BCC+99] (Boolean satisfiability) solving. We have dwelled on

8 http://www.transeda.com/
9 http://www.averant.com/
10 http://www.prover.com/
11 http://www.cs.cmu.edu/modelcheck/
12 http://nusmv.irst.itc.it/
SMV for some time now but Berbard et. al. [BBF*01](page 131) explain that it was the first model checker to use BDD and hence handle large state spaces, and examples of its use are published [CYLRO1]. The use of BDDs has its limitations as the representation of reachability in BDD's is PSPACE complete [CGJ*01]. Clark argues that more research into model abstraction is required to combat the state explosion problem. There are other popular model checkers such as SPIN [Hol03]. However, SPIN target software verification. ProB, the B and CSP model checker, is discussed in section 7.8.

In the remainder of this literature review we concentrate on research that has focused on formalising the hardware development process by utilising theorem proving and formal notations. Historically hardware has been modelled in functional languages and higher order logic languages like HOL [GM93]. Surveys of hardware verification techniques in the '90s were dominated with PVS 14, HOL and model checking approaches [KG99] [Sta94] [Gup92]. There were some notable successes with the used of algebraic functional specification languages, like the AAMP5 verification [MS95]. The verification was undertaken in PVS for the NASA space agency. The success of the AAMP5 project in the long term contrasts with the problems encountered by Charter Technologies Ltd., who licensed the VIPER processor, as reported in [Mac05]. The MoD had advertised the VIPER [Coh89] as a formally verified processor. However after typing errors were discovered in the design questions about the context of the VIPER proof of correctness were raised. The question of what constitutes a proof was about to be debated in court as Charter Technologies had taken the MoD to court over the matter, after disappointing sales of the VIPER licences. However, Charter Technologies went bankrupt before the hearing. Model checking and theorem proving has been used successfully to verify the Pentium 4 floating-point multiplier unit and floating-point adder. The model checking for this was undertaken in Forte, which has FL as the interface language. FL is strongly typed and from the same family as ML. Temporal aspects of the specification were specified as Hoare triples. Model checking was carried out via Symbolic Trajectory Evaluation (STE). Kaivola and Narasimhan [KN02] successfully proved that the Pentium4 floating point multiplier `booth_encode` procedure was correct to the IEEE floating-point multiplication specification. They believed that the success of the proof was due to the most part to organising the proof steps and the teams decomposition skills. Jones et. al. [JOS*01] viewed their work verifying the adders as only a first step that awaits industrial strength tools. ACL2 theorem prover, an industrial strength version of the Boyer-Moore theorem prover [Saw00], has been successfully used to demonstrate that pipelined processors can be verified against equivalent sequential machines. Zimmermann and Toma [YZ05] demonstrated that ACL2 can be used to verify that BHDL interface specifications are equivalent to VHDL IP blocks. One limitation of the BHDL approach discussed in section 7.3 is it does not make full use of IP blocks and reuse. Zimmermann and Toma demonstrate that BHDL interface specifications can be flattened and translated into ACL2 and compared with a ACL translation of the VHDL IP blocks. Sivadas et. al. [SRC97] in their work formally verifying microprocessors and various logic circuits highlighted the importance of mechanising the proofs. They made the proof scripts short and concise and developed hardware components that support modular proof. Larch/VHDL 15 is a theorem proving base approach to formal modelling and verification of HDLs. Models are developed in the Larch/VHDL environment, developed by Odyssey Research Associates Inc. Formal verification is carried out by Penelope the interactive theorem

14 http://pvs.cls.sri.com/
15 http://www.stcorp.com/reliablecomputing/hardwareverification.html
Chapter 7. Comparisons with Other Work

Examples of small digital components have been demonstrated. Barbour and Nassif comment that all but the simplest simplifications are require guidance in Penelope [BN97]. Although model checking and theorem proving are powerful approaches few tools have appeared in the mass EDA market to-date. Some possible reasons for the lack of uptake may be due to:

- limits on design size that can be analyzed
- expert knowledge required to formulate properties/proofs
- the safety critical market is still a small market

The approach in this thesis, of using a formal notation during design, seeks to incorporate the generation of functionally correct circuits as part of the development process (correct-by-construction) rather than relying on formulating correctness after the components have been constructed. This does not preclude the use of model checking or theorem proving at the end of the development process to added to the assurance. The remainder of the literature review focuses on approaches that incorporate formal notations in the design process.

Research into formalising Hardware Description Languages (HDL), like VHDL [BKLM97] and Handel-C [BW03] has provided semantic models of HDLs. The translation mappings between FDR2 and Handel-C [Ste03] illustrated the common ground that exists between process algebras and HDLs. Work translating B to BVHDL [ISS01] provided evidence of the commonality of specification languages and HDLs. Event-B has been used to describe hardware circuits [Abr01]. Work on using B to model VHDL [ABD+03] supports the belief that direct compilation to hardware from a formal notation is possible. Lava\(^\text{16}\) has been proposed as a HDL for safety related hardware implementation [Hil04], and Synchronous Receptive Process Theory (SRPT) [Joe92] has been proposed to under-pin the semantics of the mappings. Lava can be used to describe circuits at a netlist level, before synthesis to Xilinx’s Virtex family of FPGAs. It attempts to formalise the step from netlist to FPGA device. To do so requires focusing on a particular FPGA device family.

Co-design is an approach that delays assigning the choice of technology (hardware/software) for the implementation of components until as late as possible in the development cycle, and base that choice on system performance requirements rather than custom practice. Work using CSP||B to formalise the modelling of co-designs is under way at Surrey University. There has been some success modelling hardware components [MS06] using CSP||B. SystemC is a language intended for co-design. It is not formal. However, work proving the operational semantics of the SystemC scheduler in Event-B using refinement has been carried out successfully [DC05]. By providing a formal operational semantics for SystemC Canseil et. al. where able to give an operational meaning to every simulation. They claim it is possible using their approach to validate the correctness of a B models translated into SystemC.

The use of state based and event based formalisms is a deviation from the recent historical approach dominated with algebraic specification hardware notations. Functional languages have the advantage of utilising property preserving transformations and easy of theorem proving. State and algebraic specification languages offer a way to develop co-designs. System

\(^{16}\)http://raintown.org/lava/
on a chip (SoC) developments are becoming common. They encompass a wide range of services including hardware and software support. B is traditional used for software development. Adapting it for hardware development extends its capability and opens up the possibility of using it for co-designs, like SystemC. The annotation approach incorporates the ideas of checking temporal properties directly by starting with temporal properties as a core part of the abstraction.

Hardware modelling in classical B is outlined in section 7.3. In section 7.4 a commercial approach to formal state machine development is reviewed. In section 7.5 an alternative integration of two formal notations is considered to give contrast on the CSP||B integrations. In section 7.6 Event-B approaches modelling are considered, and in section 7.7 Event-B modalities are introduced. A B model checker is evaluated in light of the annotations in section 7.8. Approaches to translation CSP to Handel-C are considered in section 7.9.

### 7.3. Hardware Development in B

#### 7.3.1. BVHDL Hardware Description Language

We begin with the review of approaches to specifying hardware, which was the starting point of this thesis. It was discovered during the initial research phases that what was required was a higher level of abstraction that RTL. The motivation for discussing this work is to show how optimised formal hardware logic can be developed, and to show the limitations of this approach. The term BVHDL is employed to describe a collection of specialised library machines and machine templates, used to construct B descriptions that are structured like VHDL. The major strength of BVHDL is its fixed architecture that mirrors VHDL. Hardware engineers are presented with familiar building blocks, for which they have an intuitive feel. This strength can also be viewed as a weakness in that a design abstract must be introduced within the confines of a VHDL architecture. Annotations do not apply the same level of structural rigidity which allows more abstract formulations of the problem in the initial stages of design. The structural constraint in BVHDL aids translation to VHDL.

A simple example is used to illustrate the BVHDL language features. It is derived from a larger example called Another Example Processor [IF02] (AEP), which was developed at the commencement of this research. The example given in this section concerns the access control to the register bank of the AEP pipelined processor (BVHDL is well suited to the description of low level examples). To keep it small it has been rationalised. The essence of the example design is given in Figure 7.2. Two sub-modules (read and sequence) share access to the register bank, bank. The system invariant asserts that different pipeline states cannot access the register bank at the same time.

Machines can be included (INCLUDES) into other machines to structure a specification. An example of inclusion is given in Figure 7.2, where wz_PR, ss_PR, rr_PR, tt_PR and oo_PK are included. Inclusion is a strong form of ownership. Machines can only be included into another machine once. Included machines may have their OPERATIONS used by the operations of the including machine. Machines may see but not modify another machine's state using the SEES relationship. This a weak form of sharing.
Figure 7.1 introduces a Simple Machine (SM) at the top of the AEP hierarchy. The SM
includes the Process machines (PR) that makes up the simple pipeline. SM compose PR
machines. Process machines detail substitutions like VHDL processes. Inclusion in B gives
access to another machines invariant, and state via the included machine's operations. Shared
type information is provided by tt..PK and oo..PK, shown in Figure 7.2 along with the AEP
SM, aa..SM. Package machine (_PK postfix) translate into VHDL packages, and they are
used to contain shared type information. The SM represents a VHDL entity, a basic VHDL
building block. The SM machine VARIABLES represent entity signals. The machine variables
su, du and ou represent outputs, whereas oO, sO and dO are input signals. The inputs are
latched in by the port operation, aa..port. The architecture operation, aa..architecture,
concurrently executes the pipeline processes producing new output assignment statements.
Variables from the included Process Machines (PRs) are used in the architecture operation to
connect up the processes. An invariant is introduced in the SM which states that a structural
hazard can never exits. Such a hazard occurs when the different pipeline stages attempt to
read and write the same register simultaneously. Hence, s1 (the variable that refers to the
source register) and d1 (the variable that refers to the destination register) must never be
equal during reading and writing. The PR machines are architecturally a level below the SM
machines. There is a higher level BVHDL machine hierarchically superior to SM machines,
which is analogues to the VHDL MAP called the Compound Machine (CM). A MAP is where
instances of component entities are wired together with signals to form new components. The
simple nature of the running example has meant that a CM has not been needed.

The Sequencer PR (ss..PR) provides a behavioural description of the main actions of the
AEP pipeline. It is one of three PRs that model the pipeline action. A BVHDL PR may only
contain one operation. The ss..PR operation has a main case statement, which selects on
the value of the opcode parameter. If a 1dr (load the register) opcode is detected the source
register data (loaded in the last pipeline phase) is loaded into the destination register, ready
for saving in the next cycle of the pipeline.

The remaining two PRs describe the parts of the pipeline that either: (a) read the source
register, from which inputs are taken; or, (b) writes to the destination register, where results
are saved. To simplify the example the rr..PR machine owns all the pipeline registers. Every
time the process is invoked the pipeline registers contents are shifted along: the data in xO

Figure 7.1: AEP Example Essentials
MACHINE aa_SM
SEES Bool_TYPE
INCLUDES ww_PR,ss_PR,rr_PR,tt_PK,oo_PK
VARIABLES o0,s0,d0,su,du,ou

INVARIANT
  o0: Opcode & ou: Opcode & s0: 0..reg_num & su: 0..reg_num &
  d0: 0..reg_num & du: 0..reg_num &

  ((reg_written = TRUE & reg_read = TRUE) => (s1 = d3))

INITIALISATION
  o0:= nop | ou:nop | s0:= 0 | su:= 0 | d0:= 1 | du:= 1

OPERATIONS
  aa_port(xo, xs, xd) =
  PRE
    xo: Opcode & xs: 0..reg_num & xd: 0..reg_num
  THEN
    o0:= xo | s0:= xs | d0:= xd
  END;

  aa_architecture =
  BEGIN
    rr(o0,s0,d0) | ss(o1, sr) | ww(d2, dr, o2) |
    su, du, ou:= s0, d0, o0
  END
END

MACHINE tt_PK
CONSTANTS data_path, reg_num
PROPERTIES data_path:NAT & reg_num:NAT & data_path=3 & reg_num=2
END

MACHINE oo_PK
SETS Opcode = {ldr, nop}
END

Figure 7.2.: AEP Entity Architecture Simple Machine Plus Packages
is read into d1 in the first stage of the pipeline, d1 is read into d2 in the second stage of the pipeline, and d2 is read into d3 in the third stage. The rr_PR machine, depicted in Figure 7.4, SEES the registers owned by the ww_PR machine, and therefore has read-only access to them. The contents of the source registers are read from ww. On detecting a hazard (read and write of the same register at the same time) a no-operation (nop) is inserted into the pipeline instead of the fetched opcode, and the register bank is not read; otherwise the values it was called with are latched in.

The ww_PR machine (Figure 7.5) loads the data xdr into the register bank at xd2 if xo2 is not a nop. The ww_PR machine owns the register bank, which is described in the invariant as a total function that maps register locations to contents. The initialisation is non-deterministic. The pipeline functionality is very low level. The register is written to by the ww operation if the input opcode is not a nop. A register flag reg_written is used to record when the register has been written to. The ss_PR, rr_PR, and ww_PR specifications detail the mechanics of the pipeline.

Low-level specifications are arrived at by a process of formal refinement from a higher level, more abstract, specification. To allow automatic translation to synchronous VHDL we must introduce a clock and this is done using refinement. Figure 7.6 depicts a timing and data refinement. The BToolkit generates the POs that require discharging to ensure that the system refinements holds. These specific hardware refinements can be avoided if they were implemented on translation, which is the subject of chapter 6.

In the refinement r0_num_SG is included into the wwR_PR refinement. r0_num_SG is a BVHDL library machine that provides operations to set and get an encapsulated signal value. This library machine has a corresponding VHDL fragment. The B tool has been extended to allow users to create BVHDL library components with the associated equivalent VHDL to replace it on translation. The philosophy behind allowing library machines is to make use of the VHDL of trusted components. The same approach is used successful in B software development, with the library of implementation data objects. The invariant is used to link the refinement wwR_PR to the more abstract ww_PR. It links the register bank array, reg(), with the individual registers r*_num. This permits the data refinement POs to be discharged. The timing refinement is the introduction of a clock. The refinement satisfies the abstraction when there is a clock event (clockEVENT=true) which is when the clock is high (clock= '1').

To prevent circular references machines can only be included once, but this prevents the sharing of the individual registers. Although the individual registers can be seen by a refinement of rr_PR (rrR_PR) a linking invariant cannot be constructed to discharge the refinement POs (seen state may not be used in the invariant). The abstract specification could be rewritten with registers as a separate machine. A more sensible approach would be to translate to Handel-C (see chapter 6). The VHDL obtained from the translation can be found in Figure 7.7 and Figure 7.8.

There is no proof of equivalence between the semantics of BVHDL and the semantics of VHDL. An informal correctness argument relating the BVHDL PR and the VHDL process is given as follows. BVHDL assignments are made in parallel. The generated VHDL signal assignments all have a fixed delay so like the BVHDL assignments they all happen simultaneously. The VHDL assignments occur in a process when a signal in the sensitivity list changes, equally all BVHDL PR refinements must be guarded by an appropriate clock and reset signals in
MACHINE ss_PR
SEES oo_PK, tt_PK
VARIABLES dr
INVARIANT dr:0..data_path
INITIALISATION dr:=0
OPERATIONS

ss(xo1, xsr) =
  PRE
  xo1:OPCODE &
  xsr:0..data_path
  THEN
  CASE xo1 OF
  EITHER 1dr THEN
    dr := xsr
  OR 0nop THEN
    dr := dr
  ELSE
    skip
  END
  END
END

Figure 7.3.: AEP Sequencer Process Machine

MACHINE rr_PR
SEES ww_PR, oo_PK, tt_PK, Bool_TYPE
VARIABLES o1, o2, s1, d1, d2, d3, sr, reg_read
INVARIANT
  o1:OPCODE &
  o2:OPCODE &
  s1:0..reg_num &
  d1:0..reg_num &
  d2:0..reg_num &
  d3:0..reg_num &
  sr:0..data_path &
  reg_read:BOOL
INITIALISATION
  o1:= nop ||
  o2:= nop ||
  s1:= 0 ||
  d1:= 1 ||
  d2:= 1 ||
  d3:= 1 ||
  sr:= 0 ||
  reg_read:= FALSE
OPERATIONS

rr(xo0, xs0, xd0) =
  PRE
  xo0:OPCODE &
  xs0:0..reg_num &
  xd0:0..reg_num
  THEN
  /* pipe line Stage 1 */
  s1 := xs0 ||
  d1 := xd0 ||
  IF (xs0 /= d2) & /*hazard*/
    /*nop det.*/
    (xo0 /= nop) /*nop det.*/
    THEN
      o1 := xo0 ||
      sr := reg(xs0) ||
      reg_read := TRUE
    ELSE /*Insert bubble*/
      o1 := nop ||
      reg_read := FALSE
    END
  /* pipe line Stage 2 */
  o2 := o1 ||
  d2 := d1 ||
  /* pipe line Stage 3 */
  d3 := d2
END

Figure 7.4.: AEP Read Process Machine

JULY 6, 2008
Chapter 7. Comparisons with Other Work

MACHINE ww_PR
SEES tt_PK, oo_PK, Bool_TYPE
VARIABLES reg, reg_written
INvariant
  reg: (0..reg_num) --> (0..data_path) &
  reg_written : BOOL
INITIALISATION
  reg: (0..reg_num) --> (0..data_path) &
  reg_written := FALSE
OPERATIONS
  ww(xd2, xdr, xo2) =
  PRe
  xd2: 0..reg_num & xdr: 0..data_path &
  xo2: OPCODE
  THEN
    IF xo2 /= nop THEN
      reg_written := TRUE ||
      reg(xd2) := xdr
    ELSE
      reg_written := FALSE
    END
  END
END

Figure 7.5.: AEP Write Process Machine

REFINEMENT wwR_PR
REFINES ww_PR
INCluDES
  clock_EV, r0_num.STG data_path),
  r1_num.STG(data_path),
  r2_num.STG(data_path)
SEES tt_PK, oo_PK, VHDL_PK

INvariant
  reg(0) = r0_num& /*a:{b} => a = b*/
  reg(1) = r1_num&
  reg(2) = r2_num&
  clock_EV& clock= '1'
OPERATIONS
  ww(xd2, xdr, xo2) =
  IF clock_EV&
    clock= '1'
  THEN
    IF xo2 /= nop THEN
      CASE xd2 OF
        EITHER 0 THEN
          r0_num.STO(xdr)
        OR 1 THEN
          r1_num.STO(xdr)
        OR 2 THEN
          r2_num.STO(xdr)
      END
    END
  END
END

Figure 7.6.: AEP Read Process Machine
library ieee;
use ieee.std_logic_1164.all;
package asp_def is -- Definition of data types
constant reg_num : integer := 2 ; constant data_path: integer := 3 ; --
type OPCODE is (ldr,nop); type BOOL is (FALSE,TRUE);
end asp_def;
library work, ieee;
use work.asp_def.all;
use ieee.std_logic_1164.all;

entity aa is
port (oO: in OPCODE; sO, dO : in integer range 0 to reg_num;
      su, du: out integer range 0 to reg_num; ou: out OPCODE);
end aa;
architecture vhdl of aa is
signal rO_nura: integer; signal r1_num: integer ;  signal r2_num: integer;
signal ol, o2: OPCODE ;
signal s i, d1, d2, d3: integer range 0 to reg_num;
signal dr, sr: integer range 0 to data_path;

Figure 7.7.: AEP VHDL - Part 1: Declarations

the SM (adding a reset is a minor amendment to the example). Appropriate guards could
be constructed to restrict the B correctness condition to a rising clock edge as discussed
above. The resulting VHDL process would be sensitive to the clock variable of the correctness
condition, and therefore activate on it. Hence a valid state change in the BVHDL PR equates
to a valid state change in the VHDL process. The SM architectures operations connect PRs,
which does not interfere with the concurrent nature of the assignments within the PRs. The
translation of the SM architecture has the individual processes arranged concurrently with
connecting signals. Hence, the VHDL preserves the concurrent semantics of BVHDL.

The VHDL generated from the BVHDL specifications is given in Figure 7.7. Standard libraries
and custom packages concerning the VHDL data types are referenced and made visible by
the library, use and package clauses. The information in the user defined packages tt_PK
and oo_PK are translated into constant clauses, and enumerated types contained in the asp
package. The entity port and architecture aa are translated from the aa_SM machine. The
port names are derived from the assignments in the aa_port operation. The signals associ­
ated with all the process specifications are introduced into the architecture in the declaration
part. The process sensitivity lists are constructed from the actual parameters and the signal
on the right-hand side of an expression in the processes. Processes are label with their equivalent B operation name. In the first process rr the if-then-else structure of the B operation is mimicked. The function override assignment is translated into a case statement. All assign­
ment time delays are 1 ns. The parallel assignments in the B are translated into sequential
statements in the VHDL. The semantics of the B assignment statements are preserved if the
VHDL assignment statements are scheduled to occur at the same time. The ss process uses
an other catch all clause in the case statement. Only the vv machine is refined and clock
Chapter 7. Comparisons with Other Work

begin
rr : process (o1, o2, d1, d2, o0, s0, d0)
begin
s1 <= s0 after 1 ns; d1 <= d0 after 1 ns;
if s0 /= d2 and o0 /= nop then o1 <= o0 after 1 ns;
case s0 is
  when 0 => sr <= r0_num after 1 ns; when 1 => sr <= r1_num after 1 ns;
  when 2 => sr <= r2_num after 1 ns;
end case;
else o1 <= nop after 1 ns;
end if;
o2 <= o1 after 1 ns; d2 <= d1 after 1 ns; d3 <= d2 after 1 ns;
end process rr;

as : process (o1, dr, sr)
begin
  case o1 is
    when Idr => dr <= sr after 1 ns; when nop => dr <= dr after 1 ns;
    when others => null;
  end case;
end process as;

ww : process (o2, d2, dr)
begin
  if clock = '1' and clock_event = '1' then
    if o2 /= nop then case d2 is
      when 0 => r0_num<= dr after 1 ns; when 1 => r1_num<= dr after 1 ns;
      when 2 => r2_num<= dr after 1 ns;
    end case; end if; end if;
end process ww;

su <= s0 after 1 ns; du <= d0 after 1 ns; ou <= o0 after 1 ns;
end vhdl;

Figure 7.8.: AEP VHDL - Part 2: Processes

190

JULY 6, 2008
introduced in this example. To make this code synthesisable the clock would need to be in the sensitivity list of all processes, and there would need to be an asynchronous reset in each process.

7.3.2. Hardware modelling in BHDL

Boulanger et. al. outlines a methodology for developing functional B models of hardware circuits implemented with components corresponding to VHDL low level libraries [ABM02], called BHDL. In the Boulanger approach low level logic components (multiplexers) are built up from logic gates, which are themselves described by the standard logic libraries represented in B. This approach has a richer hardware data type model than the BVHDL approach, but completing the proofs with the extended types was not possible with the level of tool support when the work was reported [ABM02].

Modelling of circuits

Figure 7.9 is a description of a multiplexer gate (B_Mux_0) in BHDL. It switches one of the inputs to the output depending on the value of the select input. Each gate input pin is modelled independently in a different operation to allow them to change asynchronously. Calling a pin operation recalculates the gate function. The output is applied to the output port when the output operation is called. Boulanger et. al [ABM02] introduce the non-deterministic assignment in the form \( \text{var-list} \in (\text{predicate-list}) \) were the variables in \( \text{var-list} \) are constrained by the \( \text{predicate-list} \). In contrast a BVHDL model has two distinct parts; a part that latches inputs and a part that recalculates the output signals. Each part is captured in a different operation. The BHDL approach allows inputs to change independently as asynchronous changes occur. At the higher level in BHDL a function description is used to represent the circuit (the Compute_ function in Figure 7.9).

The multiplexer can be modelled directly in BVHDL:

\[
\text{IF Select = FALSE THEN out := in_1 ELSE out := in_2 END;}
\]

In Figure 7.9 the input pin in_1 gets a new value when in_1(va)1 is called. The value of the output pin out is selected from the valid states that result from changing the input. The B operations specify the constraints on the outputs. The BHDL approach refines the components, like the multiplexer, by importing lower level models of component using basic gates like the 'not' and 'and' gates. Theoretically if the refinement steps are small enough then they should be provable. The formalised method does assume that the functional model of the gates is a correct representation of the gates. In the example provided by [ABD+03], reworked in Figure 7.10, the implementation of the multiplexer is the lowest level of abstraction. The basic gates are imported. The link between the basic gates is made by the invariant. Local variables are created in the implementation to tie the internal signals together. Each operation has similar logic. The VHDL std_logic_1164 Library is defined although the logic model is not been used in the definition of the library gates.

When a port operation is called the value supplied in the parameter is saved in a global
Machine B_Mux_0

Definitions
Compute_(xx,yy,zz,res)==bool((xx=FALSE)->(res=yy)) & ((xx=TRUE)->(res=zz))

Variables
Select,in_1,in_2,out

Invariant
Select:BOOL & in_1:BOOL & in_2:BOOL & Compute_(Select,in_1,in_2,out)

Initialisation
Operations
in_1(val)=in_1,out:(in_1:BOOL & in_1=val & out:BOOL & Compute_(Select,in_1,in_2,out));
in_2(val)=...;
Gate(val)=Select,out:(Select:BOOL & Select=val & out:BOOL & Compute_(Select,in_1,in_2,out));
val <-- Out = val:out END

Figure 7.9.: BHDL Description of a Multiplexer

Implementation B_Mux_n

Refines B_Mux_1

Imports A1.B_And_0, A2.B_And_0, 03.B_Or_0, NN.B_Not_0

Invariant
select=A2.in2 & select=NN.in & NN.out=A1.in2 & in_1=A1.in1 & in_2=A2.in1 & A1.out=03.in1 & A2.out=03.in2 & out=03.out

Initialisation
Variables xx,yy,zz in
NN.In(FALSE); xx<--NN.Out; A1.In_1(TRUE);
A1.In_2(xx); yy<--A1.Out; A2.In_1(TRUE);
A2.In_2(FALSE); zz<--A2.Out; 03.In_1(yy);
03.In_2(zz); out<--03.Out

End

Operation
...

End

Figure 7.10.: The Multiplexer Implementation
variable, as well as being used along with the other global variables to recalculate the new output. In BHDL no distinction is made between synthesisable and non-synthesisable VHDL.

The extended bit VHDL Library

The library depicted in Figure 7.11 is a portion of the model of the VHDL extended bit standard logic type. There are nine elements in the extendedbit type. The main problem is that B does not allow operator over-loading. Consequently, many more operations are defined in the B than in the VHDL to cover the complete type mapping space. The operations are selected by finding an operation with an enabled precondition. There is no information on how this is applied in practice. In contrast the BVHDL approach obtains realistic models by utilising basic two valued standard logic, which is synthesisable. In BVHDL library machines can be introduced that have the standard logic data type. The typing rules were extended to allow sub-range typing checks. The BHDL approach includes much more of the capabilities of VHDL than the BVHDL approach, but no example of the use of the STD_LOGIC_1164 library has been given in BHDL, and it is not synthesisable to netlist form.

Summary of BHDL Approach

The use of the functional description of circuits is not a standard part of the BVHDL. It is necessary in the BHDL approach because the functionality is repeated in every operation that supports an input pin. The BVHDL approach begins from the assumption that the circuit will receive synchronised input from a clock edge, and a port operation feeds all the inputs into the circuits description on the clock edge synchronously.

The treatment of the STD_LOGIC_1164 Library is much more thought out in BHDL, but in doing so the approach creates descriptions that are not directly synthesisable. BVHDL deals only with synthesisable logic. The key question is how far down is it necessary to model to

July 6, 2008
guarantee a circuit will be synthesised correctly in silicon. The BHDL approach advocates using refinement to basic gates as a form of secure development. However, even basic gates are optimised to a lower level netlist representation before the actual device is produced by the synthesis tools. BVHDL stops at RTL a level above the gate netlist level. The approach introduced in the technical section builds on the BVHDL research. The thesis work includes a hardware description language target that is not VHDL, but drops the requirement to specify a specific clock.

7.3.3. Hardware Design in B

Stefan Hallerstede [Hal03] addressed the development of signal convolution in hardware implementation. The idea was to take space and time trade-offs of hardware design into account when producing a model. Event-B was used to model both hardware and its environment. Hallerstede modelled the input/output as streams in Event-B. In the latter stages of refinement pipelining is used to calculate the sum of the convolution. Pipelining involves introducing several independent stages in the processing stream. Because they are independent they can be performed in parallel. The results of one stage must be registered and fed into the next stage synchronously on each clock cycle. The extra registers introduced in the pipeline increase the size of the final design, but allow the clock speed to increase speeding up the final circuit. Hallerstede makes no real quantitative comparison between the trade off. The abstract convolution algorithm is used to demonstrate the correctness of the refined pipeline design.

Grant and Evans [GE07] reported the approach taken developing hardware. That built on work using Classical B and refinement to generate VHDL code for a microprocessor [ISS01]. Ifill produced a classical B model of the pipelined AEP [Ifi02] [Ifi03] processor to demonstrate how classical B could be used for pipe-line processor development. Fragments of this code are illustrated in section 1.2.2. Grant and Evans later used a similar approach to show how refinement could be used to show the equivalence between the different levels of refinement in the Score processor reference.

7.3.4. Specifying and Synthesising Hardware in B and Bluespec

Research specifying hardware has been undertaken as part of the RODIN project. Ian Oliver [Oli06] reported on the use of B and Bluespec to specify and synthesis hardware. In particular the interest is in constructing a declarative specification of the problem not polluted with implementation details like BHDL. Bluespec is a rule based declarative hardware specification language based on term rewriting, which comes with a compiler that can schedule sets of rules and synthesis SystemVerilog. Bluespec specifications, like BVHDL, have a two part description: interface (port) and module (entity). Models contain the implementation of methods introduced in the interface. Rules are used to describe behaviour. Rules are urgent: they must fire when their guards are true. They are atomic and complete in a clock cycle. All enabled rules execute in parallel on the rising edge of the clock. If two or more rules are not mutually exclusive then only one will be scheduled. A model of the traffic controller in Bluespec is given in Figure 7.12. Four of the five operations are rules that update the system, but do not communicate with the outside world, and do not therefore appear in the
interface. The pedestrian operation has output so it is translated into an ActionValue method and appears in the interface.

```plaintext
typedef enum {Aller, Arret} COMMAND deriving (Bits, Eq)

I1(COMMAND) Moat ← M1(Arret)
I2(COMMAND) Square ← M2(Arret)

interface I1;
    method ActionValue#{(Int#(8)) Aller_pedestrian};
endinterface

modules Interrupting_C_T_C_S(Interrupting_C_T_C_S)
    rule Aller_Moat( Square = Arret ); Moat ← Aller endrule
    rule Arret_Moat( Moat = Aller ); Moat ← Arret ; pv ← 5 endrule
    rule Aller_Square( Moat = Arret ); Square ← Aller endrule
    rule Arret_Square( Square = Aller ); Square ← Arret ; pv ← 10 endrule
    method ActionValue#{(Int#(8)) Aller_pedestrian if (pv > 0)};
        begin
            pv ← pv − 1;
            return pv;
        end
endmethod
endmodule

Figure 7.12: Bluespec Interrupting Traffic Controller Operations

Oliver notes that Bluespec is based on action semantics like B. Bluespec uses atomic interleaving of actions with clock scheduling and rule priority, and the Bluespec compiler schedules the rules so only a subset of the possible states are visited (scheduling puts them in a deterministic order). Although, any given B program can be mapped into Bluespec additional guarding of rules and methods is required to maintain semantic equivalence. A translation of the traffic controller is given in Figure 7.12. The annotations have not been translated over. The consequence is that there are no constraints. Hence, the Aller_pedestrian operation can be scheduled interleaved with either Aller_Square or Aller_Moat rules. Extra guards are required that represent the annotations control. The guard pv = 0 needs to be added to Aller_Square and Aller_Moat rules. B annotations could be used to add these additional guards to guide the translation. The purpose of the annotations is to guide the execution sequence, which is similar in conception to rule/method firing.

JULY 6, 2008

195
7.3.5. VHDL translation to B

It is very unlikely that hardware design engineers will adopt B as a development entry language. Hence, Aljer and Devienne have developed a translator from VHDL to B. Once in B the modules can be formally verified. Aljer and Devienne highlight three important facts. Firstly, the majority of digital circuits have at least one processor on them, which means that the majority of new designs are co-designs. Secondly, increasing chip density and clock speeds means higher random failure rates. Thirdly, the greatest concern to hardware design engineers is the burden functional verification has on projects. This implies that the majority of new designs should be fault tolerant co-designs. In addition more formal verification based around refinement could be used to help alleviate the problems surrounding functional verification. Aljer and Devienne propose a stronger emphasis on architectural design as a co-design entry point. ACME \(^{17}\) is a Architecture Design Language developed by the ABLE group at Carnegie Mellon University, and Dave Wile at USC's Information Sciences Institute. It contains the majority of important feature required to support component development and interconnection, with the power to map architecture to other architecture. The supporting tool, AcmeStudio, has a constraint language to assist in property evaluation. On the hardware side the main advantages for VHDL is that it supports structuring, strong typing, and has an unambiguous semantics. The generation of B from VHDL can be achieved using VGUI a free graphical interface. VHDL outline entities and architectures are generated from VGUI, which can be further refined into VHDL. The outline entities and architecture are derived during architectural design in ACME. During refinement pre-existing library components can be added to facilitate component re-use. ANTLR (ANother Tool for Language Recognition \(^{18}\) is used to generate the B code from the VHDL.

We do not explore in any great depth fault tolerance in the thesis work. However, the annotation framework could be extended with fault tolerance. The next exception annotation is like a NEXT annotation with an additional exception clause.

\[
OP_i \equiv \text{PRE...THEN...END} \quad \{OP_j\}\text{NEXT} \quad \{OP_j\}\text{EXCEPTION}_i \quad */
\]

If a random error occurs in the implemented system during the execution of the current operation \((OP_i)\) the preconditions of the next operation \((OP_j)\) may not be satisfied. The failure to establish \(P_j\) implies that \(P_f\) is established.

The \(P_f\) must be a disjunction of the negation of the conjuncts of \(P_j\). This means that the failure of \(OP_i\) does not have to be catastrophic, as it may failed to establish a single conjuncts of \(P_j\).

\(^{17}\)http://www.cs.cmu.edu/\acme/docs/language_overview.html

\(^{18}\)http://www.antlr.org/

196

July 6, 2008
7.4. State Machine Modelling

We review a state machine approach which adds a level of abstraction to the HDL approaches. The following approach characterises a model using transitions and synchronisations, but does not afford the specification of invariants. The Esterel language [Est05] addresses the need in state machine control specifications to write things once. In particular it provides a notation and semantics to describe synchronicity and interrupt handling in reactive programs. Ignoring the initialisation a mealy machine would require eight arcs and 4 states to describe the state machine below described in Esterel. It generates an output O when both A and B are received, and resets on R. The code fragment is represented diagrammatically in Figure 7.13.

```verbatim
module ABRO :
  inputs A, B, R;
  output O;
  loop
    [ await A || await B ];
    emit O;
    each R
  end module
```

The code fragment represents the parallel combination of the two statements A and B that terminate when both statements within them terminate. The synchronised statement occurs...
when all the actions have terminated. The time taken for synchronisation is 0 seconds. The example shows the interrupt action \( R \), called the preemption operator, which takes the state machine back to the initialisation states. This is strong preemption indicated by the full terminator (\( \cdot \)) in Figure 7.13 and “loop ... each” in the text. This means that A and B are suppressed if they occur during the initialising preemption action. In way of comparison \( \text{CSP} \parallel B \) can be used to synchronise event from different machine controllers. Event-B has no direct mechanism to hold up control flow until a synchronisation occurs. The modelling of reactivity with preemption is a key concept in Esterel programming. Esterel has a data flow capability, which will not be detailed here. It is useful to allude to the behavioural semantics of synchronisation. The synchronisation of \( p \) and \( q \) \( (p; q) \) occurs when \( p \) terminates (termination indicates by 0) and the context environment for \( q \) includes the output event of \( p \).

\[
\begin{align*}
\frac{p: E', q: l}{p': E'} & \quad \frac{q: l}{q': E'} \\
\end{align*}
\]

(seq2)

In the above the context event \( E \) determines which signals are delivered to cause \( p \) to evolve to \( p' \). \( E' \) is the event caused by \( p \) in the context environment \( E \). Berry states that signal information flows from \( p \) to \( q \), because the broadcast invariant implies that \( E' \cup F' \subseteq E \), which implies \( E' \subseteq E \), i.e. \( q \) receives signal emitted by \( p \). The behavioural semantics for parallel execution has a similar premise, except that the return codes are denoted as \( k \) and \( l \). The output event consists of \( E' \cup F' \) and the maximum of \( k \) or \( l \). The continuation program is the parallel combination of the pair.

\[
\begin{align*}
\frac{p: E', k, q: l}{p': E'} & \quad \frac{q: E}{q': E'} \\
\end{align*}
\]

(parallel)

The approach taken by annotated B is to utilise the \text{NEXT INTERLEAVING} annotation to obtain interleaved behaviour between operations of the same machine, in which all the interleaved threads must terminate before the control continues. In the future work an annotation that permits the synchronisation of operation needs to be considered. The sequential specification is obtained with the \text{SEQUENCE NEXT} annotation.

### 7.5. CSP and Z

Mota and Sampaio\[MS01\] researched the model checking of specifications developed in CSP and Z (CSP-Z). Their contribution was to present an approach to model-checking complex specification for deadlock freedom in a compositional way. By reviewing their work the merits of the CSP-Z approach can be presented. CSP-OZ is a semantic integration of CSP and Z (object Z). It is noted that work has been undertaken on this integrated by Fischer \[Fis97\].
The control systems that are the intended domain for the application of this thesis do not require object creation.

A CSP-Z specification may have external and local channel interface declarations. The totality of the interfaces constitute the alphabet ($\Sigma$). Z schemas are used to type the channels and provide the constraints for the event. Event channels have an empty schema type. The main equation details the concurrent behaviour of the specification. Processes are synchronised within the main section, using alphabetised parallel.

Annotated B achieves a form of concurrent behaviour using the INTERLEAVING NEXT annotation as mentioned above. Concurrency and communication between annotated B operations is a future goal. This annotation highlights the synchronisations between different machines operations and is discussed in the future work section 8.

Consider the code fragment in Figure 7.14 adapted from Mota and Sampaio. The beginning of that spec is given as follows:

```plaintext
spec WDT
  channel clockWDT : [clk : CLK]
  channel reset, recover : []
  local_channel timeOut, noTimeOut, failFTR, offWDT : []
...
```

The interface of the WDT specification is made up of \textit{clockWDT} (which can carry \textit{CLK} data in the variable \textit{clk}), \textit{reset} and \textit{recover} are just events signified by the empty type set. The remaining events are not visible to the outside world.

```plaintext
main = Signal ||_{offWDT} Verify
Signal = (reset \rightarrow Signal \sqcap_{offWDT} \rightarrow skip)
Verify = (clockWDT?clk \rightarrow (noTimeOut \rightarrow Verify
  \sqcap_{timeOut} \rightarrow (recover \rightarrow Verify
    \sqcap_{failFTR} \rightarrow offWDT \rightarrow skip)))
```

Figure 7.14.: Example of a CSP Controller for CSP-Z specification

The schema for the \textit{reset} event is prefixed with \textit{com-}, as are all schema definitions that related to a specific event. The schema details the state constraints as follows:

\[ \text{com}_\text{reset} \equiv [\Delta \text{State} \mid \text{cycles'} = 0] \]

In Figure 7.14 the \textit{Signal} and the \textit{Verify} processes share the \textit{offWDT} event. The \textit{Signal} process may \textit{reset} and restart over, or accept the off Watch Dog Timer (WDT) event and shut-down. The \textit{Verify} process initially accepts a clock event. Then times out or doesn't timeout and restarts. If it times out then a choice is made between recovery and failure. If recovery is successful then \textit{Verify} restarts otherwise the Fault Tolerant Recovery (FTR) fail event \textit{failFRT} is signalled and the WDT is turned off before the \textit{Verify} process terminates. In

\textit{July 6, 2008}
way of contrast we observe that currently it is possible to have two control loops within one machine, which share operations. (Only one main loop is permitted currently in annotation B controllers. This restriction was imposed to facilitate the ease of translation.)

The CSP-Z approach associates the Z schemas with CSP events. CSP events that have no corresponding schemas are permitted, which is not the case with annotations currently. The Z part is given a CSP failures-divergence semantics so that they can be put in parallel with the CSP controllers like CSP|B. The final state of a non-empty trace of events described by a CSP-Z process is the composed of the schemas events (provided the preconditions of the schemas are satisfied). If a schema precondition is not satisfied then the process behaviour is STOP (a process that will not engage in any more events). The influence of the state over the process is expressed in guards in the CSP. Functions in CSP are used to model the Z schemas. The schema \( \text{com} \) function types the schema state, tests the precondition function, and produces a set of possible new states with a postcondition function:

\[
\text{com}((v_1, \ldots, v_n), (Ch.\text{In.}\text{Out})) = \\
\{ (v'_1, \ldots, v'_n) | (v'_1, \ldots, v'_n) \leftarrow \text{state}, \\
\pre((v_1, \ldots, v_n), \text{In}), \\
\post((v_1, \ldots, v_n), (v'_1, \ldots, v'_n), \text{In}, \text{Out}) \}
\]

The conversion of the Z part is achieved with the \( Z_{-CSP} \) and other Z functions, where the Interface is the set of internal and external channels:

\[
Z_{-CSP} = \text{let} \\
\begin{align*}
Z(\text{State}) &= \\
\quad \bigcirc (\text{States, Comm}) : \{ (\text{com}(\text{State}, c), c) \mid c \leftarrow \text{Interface} \} \cdot \\
\quad \text{States} := \{ \} \& \\
\quad \bigcirc \text{State}' : \text{States} \cdot \text{Comm} \rightarrow Z(\text{State}') \\
\text{within} \quad \bigcirc \text{iState} : \text{Init} \cdot Z(\text{iState}')
\end{align*}
\]

The pair \( (\text{States}, \text{Comm}) \) is the sets of new states and associated communication offered to the environment. A new state \( (\text{state}') \) is internally selected which will recursively generate more communications from the Z function in the sequence of communications. The Z function begins with a new valid initial state \( (\text{iState}') \) drawn from Init. The states generated can not be empty. The CSP process

\[
\bigcirc \text{State}' : \text{States} \cdot \text{Comm} \rightarrow Z(\text{State}')
\]

is recursively unwound.

The translation function is extended by Kassel and Smith [KS01] to permit divergence \( (DIV) \) when there is no new state. They developed their approach for OZ-CSP. Note however, in this comparison, we do not use any object modelling. A hand translation of the BVHDL pipelined processor described in Section 7.3 in line with the Kassel and Smith approach is
given in Figure 7.15, Figure 7.16 and Figure 7.17. The translation is from B to CSP. In the supporting function section of the model given in Figure 7.15 the channels identify the system inputs and outputs. The state is a global typed set. The \( \text{Init} \) set is the state constrained with the desired initial state settings. An event occurs on one of the defined channels. The values on the channels are drawn from four types of sets: \text{state}, \text{in}, \text{out} or \text{Ops}. The \text{Ops} set in this example has elements that corresponded to the B operations read (r), write (w) and sequence (s). The input and the outputs sets have different definitions depending on the operation: the \text{in} and \text{out} functions generate the appropriate set depending on the operation parameter supplied. The \text{r} operation event produces a \text{r} channel event made up from the elements of the \text{r} in set and two parts of the present state (\text{s1'} and \text{d3'}). The s event passes no data. The \text{w} event puts the \text{w} out set on the \text{w} channel. In the main section of the model, given in Figure 7.16 the \text{effect} is a function that produces outputs and new states.

In the semantic model given in Figure 7.17 it is possible to have dependencies between events. In this model all events are enabled regardless. The semantic function develops the meaning of the B model by producing all possible valid traces that the \text{effect} and \text{event} functions can concurrently participate in. Possible operations and inputs are drawn non-deterministically from their respective sets. Both an \text{effect} and an \text{event} function call will occur in the trace provided the operation is enabled from the present state, and the effect for the inputs and present state is not empty for the selected operation. This differs from the Mota and Sampaio approach which has a trace of the communications. The trace is begun from the initial state. The Kassel and Smith CSP organisation is a sequential representation of the B and all possible sequences are tried. The CSP version of the B model is checked for hazards in the assertions part.

It is slightly easier in the Kassel and Smith model to see the intended control flow paths than it is in the BVHDL model. In the BVHDL model all the operations execute on every visit to the machine. The current state determines when they modify the state. Coding up the CSP model from Z is quite involved. Using annotations is a better approach if the state model is in B. The B model does not need modifying during the development of the CSP model in annotated B. Execution of the joint model (annotated B and CSP controller) is not necessary to show non-divergence in the annotation approach.

### 7.6. Event-B

#### 7.6.1. Introduction

The machine-annotation consistency of guarded actions can be treated in the same way as the machine-annotation consistency of preconditioned operations. Hence, annotations approaches apply in Event-B. In the following both hardware modelling and approaches that add control aspects to Event-B are detailed.
datatype OPCODE = nop | ldr
name type Nats = \{0,1,2,3\}
channel r : OPCODE.Nats.Nats.Nats.Nats
channel s
channel w : Nats.Nats.Nats
state = \{(s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2)\}
\[\begin{align*}
  &s1<-\text{Nats,} o1<-\text{Nats,} o2<-\text{Nats,} \\
  &d1<-\text{Nats,} d2<-\text{Nats,} d3<-\text{Nats,} \\
  &sr<-\text{Nats,} dr<-\text{Nats,} r0<-\text{Nats,} \\
  &r1<-\text{Nats,} r2<-\text{Nats}\}
\end{align*}\]
init = \{(s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2)\}
\[\begin{align*}
  &s1=0, o1=0, o2=0, d1=1, d2=1, d3=1, \\
  &sr=0, dr=0, r0=0, r1=0, r2=0\}
\end{align*}\]
event(r, (s1',o1',o2',d1',d2',d3',sr',dr',r0',r1',r2')) = (r_a,r_b,r_c, out) = r.r_a.r_b.r_c.s1'.d3'
event(s, s_t', in, out) = s
event(w, s_t', in, (w_1,w_2,w_3)) = w.w_1.w_2.w_3
in(r) =\{(o_r,d_r,s_r)\}
in(s) =\{\}
in(w) =\{\}
out(r) =\{\}
out(s) =\{\}
out(w) =\{(w_1,w_2,w_3)\}
Ops = \{r,s,w\}
enable(r)((s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2)) = true
enable(s)((s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2)) = true
enable(w)((s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2)) = true

Figure 7.15.: BVHDLCSP Preliminary Functions
effect(r)((s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2),(s0,s1,s2)) =
{(r),
  (s1','o1','o2','d1','d2','d3','sr','dr','r0','r1','r2'))
(s1',o1',o2',d1',d2',d3',sr',dr',r0',r1',r2')<-state,
  (s0=d2 and
    ( (s0==0) and (sr'=r0) ) or
    ( (s0==1) and (sr'=r1) ) or
    ( (s0==2) and (sr'=r2) )
  ) and
  s1'=s0 and o1'=o0 and d3'=d2
) or
(s0=d2 and
  sr'=sr and s1'=s1 and
  o1'=nop and d2'=d2
  o2'=o2 and d1'=d1 and
  d3'=d3 and d2'=d2 and r0'=r0 and
  r1'=r1 and r2'=r2
)
}

effect(s)((s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2),_)=
{(s),
  (s1',o1',o2',d1',d2',d3',sr',dr',r0',r1',r2'))
(s1',o1',o2',d1',d2',d3',sr',dr',r0',r1',r2')<-state,
  o1'=o1 and (if (o1=ldr) then (dr'=sr) else (dr'=dr))
  and s1'=s1 and o2'=o2 and
  d1'=d1 and d2'=d2 and
  d3'=d3 and sr'=sr and r0'=r0 and
  r1'=r1 and r2'=r2
}

effect(w)((s1,o1,o2,d1,d2,d3,sr,dr,r0,r1,r2),_)=
{(),
  (s1',o1',o2',d1',d2',d3',sr',dr',r0',r1',r2'))
(s1',o1',o2',d1',d2',d3',sr',dr',r0',r1',r2')<-state,
  (if (d2=0)then(r0'=dr and r1'=r1 and r2'=r2) else
    if (d2=1)then(r0'=r0 and r1'=dr and r2'=r2) else
    if (d2=2)then(r0'=r0 and r1'=r1 and r2'=dr) else
    SKIP
  ) and s1'=s1 and o1'=o1 and o2'=o2 and d1'=d1 and
  d2'=d2 and d3'=d3 and sr'=sr and dr'=dr
}

DIV = DIV

Figure 7.16.: BVHDL-CSP Main Section (Effect)
\textbf{Figure 7.17: BVHDL-CSP: Semantic Function}

\begin{verbatim}
semantics(Opss,in,out,enable,
effect,init,event) =

let
  Z_PART(ss) =
  \begin{array}{l}
    \[\text{op : Ops} \& \text{enable(op)}(ss) \&
    \[\text{i : in(op)} \&
      \text{if empty(effect(op)}(ss,i))
      \text{then}
      \begin{array}{l}
        \text{\text{lo} : out(op)} \&
        \text{event(op,ss,i,oo) -> DIV}
      \end{array}
      \text{else}
      \begin{array}{l}
        \text{\text{lo}(oo,ss') :}
        \text{effect(op)}(ss,i) \&
        \text{event(op,ss',i,oo) ->}
        \text{Z_PART(ss')}
      \end{array}
    \end{array}
  \end{array}

  Z_MAIN = \text{\text{in} \& \text{Z_PART(ss)}}
  \text{within Z\_MAIN}

Aep =
\text{semantics(Opss,in,out,enable,}
\text{effect,init,event)}

\text{RUN2(A,B)} = \[\text{a : diff(A,B) \& a -> RUN2(A,B)}

r_no_hazard =
\{(r,b,c,d,e,f) | \begin{array}{l}
  b<\text{OPCODE}, c<\text{Nats}, d<\text{Nats},
  e<\text{Nats}, f<\text{Nats}, e==f
\end{array}\}

\text{assert RUN2(\{r,s,w\},r_no_hazard) \{T= Aep}
\end{verbatim}
7.6.2. Hardware Development in Event-B

Abrial’s approach to hardware modelling captures the notion of a clock as an alternation of control between the environment and the circuit [Abr01]. The switch from one state to another is accompanied by a period of instability when changes within the system are rippling through. Eventually, a stable state will be reached. Only the stable states are modelled.

\[ (\text{mode} = \text{env}) \Rightarrow C((\text{state}, \text{input}), (\text{box}, \text{output})) \]
\[ (\text{mode} = \text{cir}) \Rightarrow D((\text{state}, \text{input}), (\text{box}, \text{output})) \]

When the environment is in control \( (\text{env}=\text{mode}) \) the environment state and the input it supplies to the circuit produces a stable circuit state \( \text{box} \), and stable output. A similar relation holds for the case when control is with the circuit \( (\text{mode} = \text{cir}) \). The system evolves as the mode alternates between environment and circuit, which models the toggling of a system clock. Consistency conditions exist between the static states relations \( C \) and \( D \) across adjacent clock periods. The annotation examples given in this thesis do not included a model of the environment. Abrial’s approach is stylistic and can be accommodated in the annotated B approaches. \( C \) is the predicate that relates the \((\text{state}, \text{input})\) tuple to the \((\text{box}, \text{output})\) when the \text{mode}\ is set to the environment \( (\text{env}) \), and \( D \) is the predicate that relates the tuples when the \text{mode}\ is set to circuit \( (\text{cir}) \).

\[ C((\text{state}, \text{input}), (\text{box}, \text{output})) \land P_1(\text{output, state}) \Rightarrow D(F_1(\text{output, state}), (\text{box, output})) \]
\[ D((\text{state, input}), (\text{box, output})) \land Q_1(\text{input, box}) \Rightarrow C((\text{state}, \text{input}), G_1(\text{input, box})) \]

The first consistency predicate introduces \( P_1 \), a query predicate, and the next state generating function \( F_1 \). When \( P_1 \) holds in \( C \) it is implied that \( F_1 \) generates a new environment that hold in \( D \). A similar argument applies to the second consistency formula, but \( Q_1 \) is a predicate over the circuit configuration, and \( G_1 \) is a function that produces a new circuit and output state.

This approach supports refinement. At the highest levels of abstraction the actions of the environment and reactions of the circuit are specified to preserve invariant conditions. The guards of the circuit are refined such that the refined guards are implied by the more abstract guards. (Refinement is not considered directly in the approach adopted by Treharne [Tre00].) Event-B permits the introduction of operations during refinement and typically the final circuit will be a single machine incorporating the reactive components of the more abstract circuit event machines. B select statements are refined into if – then – else constructs. Logic components have a set relationship to the B. Variables may represent inputs from the environment, outputs to the environment, or internally registered \( \text{out-reg} \) signals. The example of the PULSER1 circuit is extended with output registering and clock synchronisation \( \text{clock-reg} \) in Figure 7.18. Deadlock freedom is guaranteed if it is possible to satisfy at least one guard. Figure 7.18 is given without showing the actual B. It is possible to see from the diagram of the specification that the actual gate logic and clock plays an important role in Abrial’s approach. Modelling approaches that encompass a low-level approach have been
Figure 7.18.: PULSER1 Circuit with Clocking

considered in section 1.2.1 and will not be discussed further here. The annotation approach does not define an explicit clock.

7.6.3. Event-B Manual and VARIANT clause

The Event-B Reference Manual [MAV05] introduces various clauses to prevent divergence not available in the B-Method [Abr96]. The POST construct appears in specifications and refinements. It permits predicates to be formulated using both the current and next values of variables. This could be used to describe a local variant (local to the particular event it is associated with). A variant is employed in classical B to demonstrate that a loop in an implementations will terminate. The global VARIANT clause in Event-B is permitted to appear in refinements and applies globally. Every new event introduced by refinement must reduce the variant. In the future work chapter 8, it is proposed that a variant should be purely localised to the particular execution sequences open to the current operation. The variant detailed in section 8 guards against the execution becoming stuck in a particular circuit.

The RODIN project focuses on the development of large complex fault tolerant systems. The aims of RODIN is to provide a tool set and approach to allow clear modelling of large systems. The project has produced an open source tool set that can be used to develop and verify Event-B specifications. The development of plug-ins was encouraged throughout the project.

One advantage of Event-B is that modalities and a variant clause are supported by extensions

\[http://rodin.cs.ncl.ac.uk/index.htm\]
to it. However, a disadvantage is that the RODIN tool, developed around Event-B, does not support INCLUDES structuring. Two modalities are illustrated in the reference manual: MAINTAIN and ESTABLISH. The MAINTAIN has the following form:

\[
\text{BEGIN } E \text{ MAINTAIN } P \text{ UNTIL } Q \text{ VARIANT } V \text{ END}
\]

In the above \( E \) is an event list. Any event from the list must maintain the predicate \( P \) until such time as the \( Q \) predicate is established. The inevitability of \( Q \) is established by the variant \( V \) which is decreased by each events in \( E \).

### 7.6.4. Distributed State Machine Models in Event-B

In this section two approaches to modelling railways systems will be considered. Both describe the same signalling system.

Papatsaras and Stoddart [PS02] incorporate state machine modelling in their approach in Event-B. They deliver their method as an example of modelling a railway system. A set of functions describing the mappings between railway sections, signal, trains and train speeds are used in machines describing the global system. The refinement separates out the train, section and signal entities into different machines. Operations distributed across several machines have the same name. In a distributed description an event occurs when all the similarly named events are enabled. Papatsaras et al. permit multiple access to the same data structure, providing the access is to different parts of the structure. The interleaved annotation introduces a framework of variable testing to show no interface occurs between variables of a machine.

\[
||f:FUNCTION \text{Operation}\_\text{Label}(i)\_\text{Operation}(\text{parameter}\_\text{List})\]

In the above the parallel application of the set of dot prefixed events of the same name are executed. The distributed parts of the event are synchronised.

In the global model two events were defined: \textit{clear} and \textit{start}. \textit{clear}, depicted in Figure 7.19 moves a train forward if possible and then adjusts the network. Unusually, for Event-B, the events are parameterised. This creates a collection of events for all possible trains in all possible sections. The abstract event is a generic template for each possible combination of train or section. Figure 7.19 shows the legal transitions of the state machine that captures the clearing of trains through sections with respect to the i\textsuperscript{th} track, with the signal states: red (R), amber (A) and green (G). The signal of the i\textsuperscript{th} track protects the next section (see Figure 7.20). The signal machine takes one of the listed actions when the \textit{clear} event given in Figure 7.19 occurs.

The parameter \textit{next}(i) and \textit{next}(\textit{next}(i)) presumes the existence of the related signal machines. The \textit{clear} operation detects the legal transitions that change state with guards. The rest are moped up with a \textit{skip} action. The final distributed system is composed of the signal machines, train machines and controller machines. The train machines record the position of trains, while the controller machines modify the speed of the trains. It is not possible to model the interaction of the labelled signal machines in annotations at present.
clear(train, next(i), signal): \[ R_i \rightarrow A_i \]
clear(train, i, signal): \[ A_i \rightarrow R_i \]
\[ G_i \rightarrow R_i \]
clear(train, next(next(i)), signal): \[ A_i \rightarrow G_i \]

Figure 7.19.: Legal Signal Transitions

Figure 7.20.: Track and Signal Arrangement
Butler reported on a systems approach to specification that involved refinement and Event-B [But02]. We refer to it here because it addresses the problem of action synchronisation. The work was undertaken as part of the MATISSE project (IST-1999-11435). A railway example is used to introduce the approach. The system model introduces the sets SECTION and TRAIN. The derived controller automatically breaks the trains, which removes the need for signals like in the Papatsaras and Stoddart model. The Butler model describes a static network of track sections, and a dynamic switch configuration. In the global system model a check function applies the breaks if the front of the train has no next section to traverse to, or the next section is occupied.

Figure 7.21 portrays the arrangement of a first refinement of Butter's model pictorially. No code is given. In the first refinement the `SendTrainMsg` event is introduced. To reflect the final system a communication process is modelled. The information needed to perform the `Check` is transmitted to the train using the `SendTrainMsg`. This event was introduced in the refinement, a common practice in Event-B. The arrangement of the guards effectively order the execution sequence (an approach avoided by annotations).

The decomposition into parallel interacting machines follows the data refinement. In Butler's model three machines naturally emerge: `TRAIN`, `TRACK` and `COMMS`. Figure 7.21 has the concept of message passing in a communication channel. Synchronisation is achieved via value-passing. The `Check` operation of the `TRAIN` machine is synchronised with the `DeliverMsg` of the `COMMS` machine.

The `Check` operation is illustrated below:
Chapter 7. Comparisons with Other Work

Check(t : TRAIN) =

VAR b ∈ BOOL WHERE
  b ← comms.DeliverMsg(t);
  trains.Check(t, b)
END

In Papatsaras and Stoddart's approach synchronisation was achieved by parallel execution of similarly named events when every guard condition across all events were enabled. Data was not passed between synchronised events. In Treharne data is passed via CSP channels. In Butler's approach synchronisation is achieved by message passing. The annotation approach has yet to consider synchronisation.

7.7. Dynamic Constraints in B

Event-B allows a system to be viewed as a set of events that may spontaneously occur. No explicit dynamic claim outside the use of the invariant could be made globally about Event-B specification, prior to the work of Abrial and Mussat who researched adding dynamic constraints to Event-B [AM98]. Importantly, their framework maintains the strong notion of B refinement. They insist "refinement must not restrict the external non-determinism that is offered by the abstract system at the top level". Abrial and Mussat proposed a number of new clauses for Event-B to support, amongst other things, the temporal notion that every execution from any point in an execution of an Event-B system eventually reaches a state, where a certain property is true/false (leadsto). Temporal clauses such as these are handled in the same way as static constraints. The framework to allow temporal reasoning is discussed. Particular attention is applied to the leadsto temporal assertion which is not directly implementable in the annotation framework, but we demonstrate how the annotation framework can be extended to capture temporal assertion in chapter 8.

In Event-B new events can be introduced during refinement. Such events refine skip. To avoid a new event being introduced during refinement taking control forever POs are introduced that ensures that the new event decreases a global VARIANT V.

\[ V ∈ \mathbb{N} \]
\[ [v := V][B_{se}] (V < v) \]

In the above the new operation \( B_{se} \) must reduce the integer variant \( V \) after it has been executed. In our framework we do not insist that individual operations cannot take control forever. In fact interrupts are used to break into possible infinite execution cycles that could involve a self-looping operation.

Abrial and Mussat introduce the DYNAMICS clause to allow expression of predicates concerning the evolution of certain variables in all executions of the system. It has global range and can use the pre and post versions of a variable the system as following:
DYNAMICS $z \leq z' \land y \leq y'$

The example above illustrates the generation of POs that ensures that every event increases both $z$ and $y$. The POs are constructed by using before-after predicate associated with an event $\varepsilon$. The background to before-after predicate construction for B operations is given in the B Book [Abr96]. The PO associated with the $x$ variable is reproduced below:

$$DYNAMICS \quad prd_x(\varepsilon) \Rightarrow \mathcal{G}(x, x')$$

In the above $prd_x$ returns the before and after predicate of $\varepsilon$. The above predicate says that if a before-after predicate relates the variable $x$ and $x'$ then it implies that the dynamic invariant $\mathcal{G}(x, x')$ must hold (stated in the DYNAMICS clause above). In our framework annotations are used to capture before-after predicates. After the current operation $OP_i$, with a precondition of $P_i$, has completed executing the next operation $OP_j$ will be enabled, if the following annotation exists in the current operation:

$$OP_i \cong (P_i \mid B_i) \ast \{OP_j\} NEXT \ast ;$$

In the above there is an implied DYNAMICS clause in which the predicate in $P_i.x$ hold before execution for the variable $x$ and the predicates in $P_j.x'$, which hold after execution for the variable $x$:

$$prd_x(OP_c) \Rightarrow \mathcal{G}(P_i.x, P_j.x')$$

A particular modality of interest is $P LEADSTO S$: once $P$ holds then eventually $Q$ will hold. It is captured in a system with the following clauses:

ANY $y$ WHERE

$P$

LEADSTO

$Q$

WHILE

$\mathcal{F}_1$ OR ... OR $\mathcal{F}_n$

INARIANT

$J$

VARIANT

$V$

END
Chapter 7. Comparisons with Other Work

In the above $y$ sets up the initial conditions that enables $P$. There after the loop guarded by $\neg Q$ chooses an enabled event from $F_1 \lor \ldots \lor F_n$. The looping action repeats until $Q$ is enabled. All the events $F_1 \lor \ldots \lor F_n$ maintain the next invariant $J$ and reduce the variant $V$. So can only guarantee $Q$ is reached if and only if $F_1 \lor \ldots \lor F_n$ are executed, and nothing else before $Q$ is reached.

The POs for the $\text{leadsto}$ modality is:

1. $\forall y . (P \Rightarrow J)$
2. $\forall y . (P \Rightarrow \forall z . (I \land J \Rightarrow V \in \mathbb{N}))$
3. $\forall y . (P \Rightarrow \forall z . (I \land J \land \neg Q \Rightarrow \forall i : \{1..n\} \rightarrow [F_i]J))$
4. $\forall y . (P \Rightarrow \forall z . (I \land J \land \neg Q \Rightarrow \forall i : \{1..n\} \rightarrow [v := V][F_i](V < v)))$
5. $\forall y . (P \Rightarrow \forall z . (I \land J \land \neg Q \Rightarrow \text{grd}(F_1) \lor \ldots \lor \text{grd}(F_n)))$

All the above POs have an initial non-deterministic choice given to them by the quantification of $y$, and all hold under the assumption that the $P$ holds. The $z$ quantification ranges over the variables that change. The first PO insists that the next invariant $J$ holds, and because $P$ is an assumption the local conditions are set up for the search for the final $\text{leadsto}$ predicate $Q$. The second PO states that under the assumption that the global invariant and next invariant hold the variant must be a natural number. This is necessary as the variant is decremented after each event from the event list has occurred. The third PO states that under the assumption that the local and global invariant hold and the future predicate has not been reached, then each event in the list establishes the next invariant $J$. The forth PO states that the variant is reduced by each event listed. The same assumptions that applied in the third PO apply in the fourth and fifth PO. The fifth PO states that at least one event guard is enabled during the search for the future state $Q$, to prevent deadlock. What is missing is a PO that when $V = 0$ the search will be complete. This can not be stated as $Q$ may be achieved earlier than when $V = 0$. It is implicit from the realisation of the $\text{leadsto}$ modality as a loop that the loop is continued if $\neg Q$, and when $Q$ holds the loop has terminated achieving the goal of the $\text{leadsto}$ operator. In the future work Chapter 8 the $\text{CONDITION NEXT}$ annotation is modified to produce a variant annotation $\text{CONDITION VARIANT}$.

7.8. ProB

ProB\textsuperscript{20} is a model checker [LB03] for B. It can be used to analyse B specifications in two ways. Firstly, it can check consistency, and secondly, as an animator, it can be used to debug B scripts. To obtain an exhaustive state space exploration the variables in the B specification must be given an appropriately small range. There are two forms of consistency checking. To find problems in the specification ProB can either navigate a sequence of operations from the initial condition to achieve a state that invalidates the invariant, or search for a valid state that can be followed legally (precondition satisfied) by an operations with a set of valid inputs that creates a new state that invalidates the invariant. The former check is model checking

\textsuperscript{20}http://users.ecs.soton.ac.uk/mai/systems/prob.html
and a trace that leads to an invalid state will be returned. The later analysis is constraint checking and may return unreachable states. Leusche l insists that proof is not enough\textsuperscript{21} ProB can be used most effectively to highlight incorrect proof obligations, thus saving time otherwise spent trying to discharge them. The animator can be used to explore the state space to look for unspecified behaviour.

The ProB Animator, unlike the B toolkit animator, will offer concrete values where one of a number of values would be valid. The tool offers a set of operations with valid inputs to cover the range of possibilities when stepping through a sequence of operations, provided the argument state space is bounded. This applies to the non-deterministic choice offered by the ANY statement. Like the B toolkit backtracking is possible.

The temporal model checker checks for invariant violation or dead-code. Although B toolkit can prove that every operation when called within its precondition will maintain the invariant the ProB tool automatically checks this. If an violation is found then the shortest trace to that state is indicated to the users. In our approach we ensure that the intended control sequence at all points in the execution maintains the invariant, and we do not have to limit ourselves to finite state. We also insist that every operation has at least one next operation in a B machine to prevent deadlock.

Work to extend the ProB tool to accept the full CSP, has been carried out by Leuschel, it will enable CSP to be model checked by ProB directly. Eventually the aim is to model check CSP$\|B$. At present CSP controllers can be used to direct the operations and therefore restrict the space of possible executions to a manageable state space size. Model checking the CSP$\|B$ will reduce the reliance on the CLI as a means of checking the consistency of control loops in the CSP script. One advantage of using annotated B is that there is less reliance on CLI generation and analysis. One advantage that annotations have over CSP$\|B$ is that annotations do not need to generate or discharge CLI. The introduction of CSP$\|B$ model checking will erode this advantage where the development model uses a small finite set of values.

In other research on model checking of Event-B systems Bellegarde et. al. [BCJ02] notes that that model checking is PSPACE-complete. They suggest undertaking model checking by executing the model in synchronous with the buchi automaton of the negation of the property $P$ for verification. The checking amounts to searching for accepting cycles, which involves the overhead of memorising previous states. The overhead is the production of the negated property buchi automaton. To check $TS$ with $P$ under a fairness hypothesis the complexity is:

$$O((|S| + |\rightarrow|) * 2^{P} + \sum_{h \in H} |h| + 1)$$

The model checking complexity is linear in the number of states and transitions, but polynomial in the number of verification predicates and number of hypotheses raised to the power of the hypotheses. So care has to be taken when developing the properties. We note that the worst case complexity of Computational Tree Logic (CTL) is

$$O((|S| + |\rightarrow|) * |P|)$$

\textsuperscript{21}RODIN Industry day - 10/08/07 - Paris

July 6, 2008
[BBF+01] [HR00]. CTL is not as expressive as LTL and can not express strong fairness properties. An advantage of annotations is that they enables an engineer to capture the execution requirements of a family of controllers during B development, and because it is based on proof it can tackle infinite types during abstract specification. Additionally, annotated B permits the separate refinement of the state operation from the controller.

MACHINE Interrupting_C_T_C_S
SETS COMMAND = {Arret, Aller}
VARIABLES Moat, Square, pv

INVARIANT Moat : COMMAND \land Square : COMMAND \land
(Moat = Arret \lor Square = Arret) \land pv : 0..10 \land
(pv > 0 \Rightarrow Square = Arret \land Moat = Arret)

INITIALISATION Moat, Square := Arret, Arret || pv :: 1..10 /*{Aller_Moat}*/

OPERATIONS
Aller_Moat \equiv \text{PRE} Square = Arret \text{ THEN } Moat := Aller || pv := 0 END
/*{Arret_Moat}*/;

Arret_Moat \equiv \text{PRE} Moat = Aller \text{ THEN } Moat := Arret || pv :: 1..5 END
/*{Aller_Square}*/;

Aller_Square \equiv \text{PRE} Moat = Arret \text{ THEN } Square := Aller || pv := 0 END
/*{Arret_Square}*/;

Arret_Square \equiv \text{PRE} Square = Aller \text{ THEN } Square := Arret || pv :: 5..10 END
/*{Moat}*/;

pv1 \leftarrow \text{Aller_pedestrian} \equiv \text{PRE} pv > 0 \text{ THEN } pv := pv - 1 || pv1 := pv - 1 END
/* NEXT INVARIANT {Moat = Arret, Square = Arret} */
/* FROM SET {Arret_Moat, Arret_Square} */
/* {Aller_Pedestrian}; {Aller_Moat} CONDITION NEXT * /

Figure 7.22.: Traffic Controller Machine for ProB

The ProB analysis of the Interrupting Traffic Controller was undertaken. The necessary changes to the variable ranges are depicted in Figure 7.22 in bold. The original model is given in Figure 8.1. There are no substantial differences between the controller generated from the annotated B, detailed in Figure 8.2 and a controller that would otherwise be developed for a CSP\|B specification in this example. The restriction on the types makes little difference in this example. We would like to check that while the CONDITION NEXT operation, Aller_Pedestrian, is active the traffic is halted. Hence an extra next invariant has been added. This annotation would be checked statically within the annotation framework. The next invariant is discharged by ProB. We expect that an updated ProB tool will be able to check all of the annotations POs if it were modified to accept them as either invariants or
assertions. In general we see our annotation approach as a way of capturing initial execution concepts. Annotation become assertions in a FDR check or a ProB check. Recently, ProB has been extended to undertake Linear Temporal Logic (LTL) checking [LP07]. Temporal logic assertions are added to the machine and they are checked in turn. If an assertion is false then a trace that shows that the assertion is false is shown. The temporal logic operators are given in table 7.1.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>G(\phi)</td>
<td>(\phi) always true in the future execution</td>
</tr>
<tr>
<td>X(\phi)</td>
<td>(\phi) true next</td>
</tr>
<tr>
<td>F(\phi)</td>
<td>(\phi) true at some point in the future</td>
</tr>
<tr>
<td>U(\phi, \theta)</td>
<td>(\phi) until (\theta) true</td>
</tr>
<tr>
<td>R(\phi, \theta)</td>
<td>(\phi) true when (\theta) false</td>
</tr>
<tr>
<td>H(\phi)</td>
<td>(\phi) true for the entire past</td>
</tr>
<tr>
<td>Y(\phi)</td>
<td>(\phi) true in the previous step</td>
</tr>
<tr>
<td>O(\phi)</td>
<td>(\phi) true at least once in the past</td>
</tr>
</tbody>
</table>

Where \(\phi\) and \(\theta\) are temporal formulas

Table 7.1.: The Temporal Logic Operators Used in ProB

Leuschel has experimented with discharging annotations by creating formulas that do the checks of annotations. The NEXT annotation can be coded up as follows:

\[ G[\text{Aller\_Square}] \Rightarrow X(e(\text{Arret\_Square})) \]

The above states that always when Aller\_Square is enabled and executed it is implied that next Arret\_Square will be enabled (e(Arret\_Square)). The FROM annotation can not be modelled directly. It has to be converted into NEXT annotations before it can be modelled in temporal logic.

LTL is not the only kind of temporal logic. CTL, as introduced above, is another kind of temporal logic, which has a branching time logic meaning as opposed to the linear logic meaning of LTL. For completeness the temporal connectives of CTL are informal introduced in table 7.2.

7.9. Translation of CSP to Handel-C

Papatsaras and Stoddart (PS), have constructed CSP to Handel-C [PS04] translation mappings. We compare their findings to the work of S. Stepney [Ste03] (SS). The subset (not including informative comments) of CSPM (machine readable CSP) handled by the translators is reproduced in table 6.1 on page 163.
Chapter 7. Comparisons with Other Work

<table>
<thead>
<tr>
<th>Operator</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGφ</td>
<td>along all paths in all future states φ</td>
</tr>
<tr>
<td>EGφ</td>
<td>along at least one path in all future states φ</td>
</tr>
<tr>
<td>AXφ</td>
<td>along all paths next φ</td>
</tr>
<tr>
<td>EXφ</td>
<td>along at least one path next φ</td>
</tr>
<tr>
<td>AFφ</td>
<td>along all paths at some point in the future φ</td>
</tr>
<tr>
<td>EFφ</td>
<td>along at least one path at some point in the future φ</td>
</tr>
<tr>
<td>A[φUθ]</td>
<td>along all paths φ until θ true</td>
</tr>
<tr>
<td>E[φUθ]</td>
<td>along at least one path φ until θ true</td>
</tr>
</tbody>
</table>

Where φ and θ are CTL temporal logic formulas.

Table 7.2.: CTL Temporal Logic Connectives

The key differences between Papatsaras and Stoddart’s (PS), and Stepney’s (SS) approaches is that the former introduced macros into the Handel-C from CSP comments. The macros are translations of CSP actions which perform data processing. The motivation to move the data processing code into CSP comments and then into macros is to prevent them being analysed in the CSP. Tools like FDR2 and ProB do not manage large state space effectively. Another important distinction is that PS’s approach does not have the capability for the declaration of local variables in functions: no functions are generated in the translation. In the SS approach the process parameters can be used to create an index of processes. In the PS approach channel types are calculated from use. However, having to type the channels, as is the case with the SS approach, provides a richer specification and double check on the use of the channels.

A Handel-C restriction to point-to-point communication restricts the nature of CSP sharing parallel. If in the CSP two events share the same alphabet then they must in the Handel-C be the processes that communicate with one another. The SS approach forces the synchronisation to be over the entire alphabet of the specification. No examples of synchronised parallel are given in the PS approach.
8. Future Work and Conclusion

8.1. Future Work

The main focus of the future work is to demonstrate through example that annotations can be applied to Event-B and develop a plug in to generate either properties for discharging by ProB or proof obligations for discharging with a proof tool. This is by no means not the only thing left to do.

8.1.1. Variant Extensions

In future research the Event-B variant concept could be introduced into annotated B in two ways. Either by introducing a new annotation that mimics the maintain until variant modality (section 7.6.3), or alternatively the current condition next annotation (section 5.3) could be modified.

8.1.2. EVENTUALLY FUTURE Annotation

The work by Abrial and Mussat introduced dynamic constraints to Event-B, which were discussed in Section 7.6.2. The form of the dynamic constraint is given below:

\[
\text{BEGIN } \text{E MAINTAIN } P \text{ UNTIL } Q \text{ VARIANT } V \text{ END}
\]

The variant decreases on every operation call maintaining \( P \) until \( Q \) is asserted. Annotations have the ability to model modalities on an operation by operation basis. Only preconditions are considered here, whereas the Event-B approach considers more general predicates. The operation to be executed when exiting a loop is considered in the annotations approach. In the example below \( OP_f \) follows \( OP_c \) until \( OP_c \) is enabled.

\[
OP_c = \text{PRE } P_c \text{ THEN } B_c \text{ END}
\]

\[
/* \text{EVENTUALLY FUTURE}(OP_f, OP_c, v, OP_r) */
\]

In the above \( OP_r \) is on every execution sequence from \( OP_c \) back to \( OP_c \), where \( OP_r \) reduces the variant \( v \) which ensures that \( OP_c \) is eventually executable from \( OP_f \). No operation on
this path can increase the invariant. (The operations that make up the loop are detailed in
the annotations and controller.) This new annotation would extend the previous work by
Abrial and Mussat. The looping action is not restricted to include only one event, but extend
to include complete execution cycles of events. In future work the POs for this needs to be
deduced.

8.1.3. CONDITION VARIANT Annotation

A self looping operation that reduces a variant until it exits from the self loop is a much simpler
proposition to reason about than an EVENTUALLY FUTURE annotation. The CONDITION NEXT
annotations could be extended to capture the notion of variants and alternative action when
the variant is zero. The extended annotation is called CONDITION VARIANT. In this section
the necessary proof to show that the annotation and step-consistency controllers are non-
divergence are not undertaken. The framework is illustrated only. The proof obligations must
check the following:

1. The output must be a positive integer and each time when invoked the value must be
   reduced by exactly 1 (in this simplified version of the variant annotations the variant
   operation would have to output the variant value)
2. The precondition of the operations must be of the form \( v > 0 \) where \( v \) is the variant
3. The output must be the operation variant value reduced by exactly 1

The above framework needs some refinement but serves to illustrate the annotation approach.
However, given the above POs structure it should be possible to prove that an operation
annotated with CONDITION VARIANT cannot be continuously called without breaking the
precondition. The following changes are needed to the current version of the annotation
given in chapter 5.

The definition of step consistency changes to:

\[
\text{if } b \neq 0 \text{ then } R1 \text{ else } R2 \\
\text{is step-consistent with } M \text{ if } b \in \mathbb{N} \text{ and } R1 \text{ and } R2 \text{ are step-consistent with } M \text{ and} \\
\forall c \in \text{init}(R1) \Rightarrow c \in \text{condition_true}(a?y?z!x) \text{ and} \\
\forall d \in \text{init}(R2) \Rightarrow d \in \text{condition_false}(a?y?z!x)
\]

The controller \( \text{if } \_ \_ \text{ then } \_ \_ \text{ else } \_ \_ \) fragment decides on which path to traverse next by testing the
\( b \) value. The \( b \) output is updated in the operation with the CONDITION NEXT annotation, \( b \)
is the value of the variant. Effectively our approach distributes the checking of the leadsto
modality to individual operations events. The advantage with this approach is that not all
operations need to reduce the variant. Also the operation executed on exiting the looping
8.1. Future Work

MACHINE Interrupting_C-T_C-S
SETS COMMAND = {Arret, Aller}
VARIABLES Moat, Square, pu

INVARIANT Moat : COMMAND \land Square : COMMAND \land 
   (Moat = Arret \lor Square = Arret) \land pu : N \land 
   (pu > 0 \Rightarrow (Square = Arret \land Moat = Arret))

INITIALISATION Moat := Arret \parallel Square := Arret \parallel pu := N / \{Aller_Moat\} ! * /

OPERATIONS

Aller_Moat \equiv PRE Square = Arret THEN Moat := Aller \parallel pu := 0 END
   / \{Arret_Moat\}! * /;

Arret_Moat \equiv PRE Moat = Aller THEN Moat := Arret \parallel pu := N \parallel pu := 0 END
   / \{Aller_Moat\}! * /;

Aller_Square \equiv PRE Moat = Arret THEN Square := Aller \parallel pu := N END
   / \{Aller_Square\}! * /;

Arret_Square \equiv PRE Square = Aller THEN Square := Arret \parallel pu := N \parallel pu := 0 END
   / \{Aller_Square\}! * /;

pu1 \leftarrow Aller_pedestrian \equiv
PRE pu > 0 THEN pu := pu - 1 \parallel pu1 := pu - 1 END
   / \{NEXT INVARIANT\{Moat = Arret \land Square = Arret\}\} * /
   / \{Aller_Pedestrian\}; \{Aller_Moat\} CONDITION VARIANT * /

ENDP

Figure 8.1.: Interrupting Traffic Controller Machine with the Variant Annotation

behaviour is defined. The example given in Figure 7.22 on page 214 is rewritten in Figure 8.1, page 219 with the CONDITION VARIANT annotation.

In Figure 8.1 the fair traffic controller machine is reproduced with the addition of the variant variable $pu$, which limits the number of times the pedestrian operation can be called consecutively. The pedestrian operation is extended with the CONDITION VARIANT annotation for illustration. In Figure 8.2, for completeness, the controller for the modified traffic light controller is illustrated. It has been made ProB compatible. Although the number of invocations made during the looping of a pedestrian crossing is limited by the CONDITION VARIANT, the pedestrian loop could infinitely interrupt everything else, making it unfair.

8.1.4. Event-B Annotation Plug-In

Bringing the different strands of the work together is a goal for the future. The BVHDL route should not be disregarded in favour of the newer development route with Handel-C.
Instead the routes should be brought together so that there are diverse route to hardware. The future route is illustrated in Figure 8.3. In the future tools support will have to be more centralised on the open source RODIN tool set, because the B-Core BToolkit has not had any significant development work in several years. This will mean developing BVHDL and Handel-C generator plug-ins. A plug-in for annotations could be developed that would generate the POs associated with the annotations or call the ProB tool to discharge them. POs would be generated in the cases where ProB could not handle the data type under analysis or I/O was being used in the annotations.

### 8.1.5. Extension of the interleaving next Annotation

An extension is required to include full synchronised parallel as suggested in Section 7.4

### 8.2. Conclusion

This PhD. research had three main aims, which were stated in the abstract and are reproduced in the following discussion. They are the introduction of control directives, proving the approach will guarantee non-divergence, and show that refinement and translation is possible.

The first aim was to introduce a set of annotations to describing control directives to permit controller development in B. However, before launching into annotations some space in this thesis is dedicated in the chapter 1 and chapter 7 to the technical background and the motivation for annotations. BVHDL and CSP||B were introduced. BVHDL is, as explained in chapter 7, a specialisation of the BToolkit. Templates to support hardware design and a VHDL generation are described. However, the templates and requirements for VHDL code generation are prescriptive. For example the introduction of a clock is necessary to obtain a translation. The thread of execution is controlled by adding control flags to the processes obscuring the control flow specification. Abstraction in BVHDL is difficult because of the templates. The methodology needed to be extended in two areas. The introduction of a more abstract language was required to capture abstract specifications with control specifications, and a HDL that was free of clock details was required. Similar criticisms can be levelled at BHDL. However, the great draw back with the BHDL approach in particular is the reliance on very low libraries. Change from using B to CSP||B as the formal notation has its lim-
8.2. Conclusion

Figure 8.3.: Future Annotated B Handel-C development Methodology

... Equivalence Check...
is limited when the model or properties become large. In the related work chapter, in section 7.8, ProB the LTL model checker is investigated. Whether temporal logic requirements captured in annotations or LTL formulae can be viewed as a difference in the choice of syntax. ProB is based on LTL, which has a worst case complexity of $O((|S| + |\rightarrow|) \times 2^{|P|})$, where $S$ is the size of the state, $|\rightarrow|$ is the number of transitions and $|P|$ is the number of the property to be checked. The annotations are introduced to capture general execution requirements not execution path specifications. Hence, CTL is adequate to capture their requirements, worst case complexity $O((|S| + |\rightarrow|) + |P|)$. LTL is more expressive and is used to verify systems whereas CTL is used to specify systems. The expressiveness and use of annotations is likened to CTL. We also note that CTL can not express fairness properties. However, in the current annotations there is no provision for fairness, which again leads to the conclusion that the expressiveness of the current annotations is correct. In fact with the current annotations it is possible to model interrupts (from annotations), and an annotated B model is viewed as an open reactive system. The fairness of an open system is dependent on the choices that are made by the external environment.

The query annotations was introduced to allow the guarantees set up by NEXT annotations to carry over operations that do not change the state. For example if $op_1$ is proven to enable $op_2$, but $op_3$ is inserted in the control flow between $op_1$ and $op_2$ provided $op_3$ is a QUERy operation (chapter 3) then the arrangement is still machine-annotation consistent. This idea was generalised with the introduction of the NEXT INVARIANT, which allows state predicates to flow across operations. An example of its use is given in chapter 4. The idea of next invariants that apply on particular execution path is new. They are used to discharge annotations POs that involve several operations applications. Normally, extra preconditions or state variables would be added to the operations during the discharging of the PO in the proof environment. We record the requirements of execution associated with annotations separately from the static precondition of predicates.

The annotations can deal with I/O. Event-B specification generally introduce I/O into events using the ANY clause. CSP||B introduces I/O using the CSP as an interface between the environment and the B machine operations. In annotated B the model of I/O, introduced in chapter 4, is more sophisticated than the current CSP||B model. This assists with implementation, because it is possible to differentiate between internal and external communication. The model of communication was simplified in chapter 6 to allow the introduction of refinement and translation. Future work is necessary to develop the model of I/O. The underlying model of communication makes annotated B distinctively different to CSP||B as the B and CSP models are different views of the specified system. The CSP does not control the B. The B and CSP are different views of the same system. Both views are required for an efficient translation. The remaining sections built on the basic annotations. The complete set of annotations are given in table 8.1 and the glossary B.

Adding conditional annotations like CONDITION NEXT and the generalisation of it in the SELECT NEXT annotation, allows execution branching based on either the returns of the operation or some B state. The current set of annotations are enough to be able to begin development. Importantly the framework for the annotations has shown to be extendible.

Secondly, we have proven that CSP controllers that are consistent with the annotations will preserve the non-divergence property established between the machine and the annotations.
Table 8.1: Annotations and Related Control Operations

<table>
<thead>
<tr>
<th>Name</th>
<th>Control Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>NEXT</td>
<td>(\rightarrow) and (\square)</td>
</tr>
<tr>
<td>FROM-ANY</td>
<td>(\triangle)</td>
</tr>
<tr>
<td>FROM-SET</td>
<td>(\triangle_x)</td>
</tr>
<tr>
<td>QUERY</td>
<td>none</td>
</tr>
<tr>
<td>NEXT INVARIANT</td>
<td>none</td>
</tr>
<tr>
<td>SELECT NEXT</td>
<td>if - the - else</td>
</tr>
<tr>
<td>CONDITION NEXT</td>
<td>if - then - else</td>
</tr>
<tr>
<td>SEQUENCE NEXT</td>
<td>(\rightarrow)</td>
</tr>
<tr>
<td>INTERLEAVED NEXT</td>
<td>(\mid\mid)</td>
</tr>
<tr>
<td>FROM INTERLEAVED</td>
<td>(\mid\mid)</td>
</tr>
</tbody>
</table>

The idea of demonstrating non-divergence comes from the CSP||B work. It is important to that community because the CSP controller drives the B specifications. Proving non-divergence demonstrates that the CSP controller will not drive the B to diverge. In the annotated B approach the CSP and B models are different views, but it is just as important to show non-divergence to show the views are compatible. The proof work also covers showing that operations with distinct state space can be safely interleaved, and that operations that do not modify the state space will carry across the guarantees set up by NEXT annotations. In future work we need to show that the translations preserve the annotated B and CSP model semantics.

Thirdly, we show how annotation refinement is possible, and show a range of mappings from the annotated B and the consistent controller to Handel-C. Refinement within the B specification is always possible. We have additionally shown that the annotations support refinement, and that operations can be introduced providing they met the rules of annotations refinement. More work on refinement can be undertaken in the future. The refinement introduced in chapter 6 is not formally justified in terms of either Classical B, Event-B or CSP refinement. In relations to the translation work the current translation to Handel-C needs justification as mentioned above. Also other languages need to be considered as possible candidates for target languages.

There were some specific points that were reported in the introduction that at the start of the research were considered aims of the research. These and additional aims that arose during the research are visited below.

- To work with existing tools - This is true to a certain extent, but in order to automatically generate the PO associated with the annotations the BToolkit or Rodin tool will need to be extended. This point has been discussed above in relation to plug-ins.
- To have one design entry tool and language - B is the central design language and
the ultimate aim is to synthesis at least the framework for the controllers from the annotations. Having B as the front-end tool allows the state specification and the control framework requirements to be captured in one place.

- **Simplify the development of correctness arguments** - The annotations, once shown to be machine-annotation consistent with the B machine, can be used independently from the machine to assess the correctness of CSP controllers developed to detail the control behaviour.

- **Opt for simplicity to accommodate analysability** - The structured way the annotations are introduced allows others to be introduced in a straightforward and robust fashion. In the future different sets of annotations developed for different purposes should be merged provided they follow the same framework.

- **Support code generation** - Code generation has been supported and the introduction of clocks for state machine designs has been eliminated in hardware development.

- **Permit hardware or software implementation** - We have demonstrated that hardware can be translated from annotated B. Producing software should not be any harder. It is possible to generate software already from B implementations.

- **Avoid operating system** - Operating systems have been avoided with the hardware implementation route. However, it may not be possible to avoid operating systems if a software route is selected.
A. The Comparison Between CSP\textvert|\textbar B's Translations and B Annotations

Here we restate the CSP translation mappings established in [TS00] and show the relationship to the annotation in table A.1 (in the table the next annotation (chapter 2) is shortened to X and the \textsc{condition next} annotations (chapter 5) is shortened to the infix \textlangle \rangle). A summary of this table without the relationships to annotations is given in Figure 1.1, page 19. The first column gives the CSP fragment to be translated. The second column gives the translation of the fragment to B. The third column presents the annotations that are associated with the particular fragment. $\rho$ signifies the translation with respect to the translation environment mappings.

The details of the B translations are given in [TS00]. The first four CSP fragments involve action prefix, which associate with the \textit{next} annotations. Discharging the \textit{next} POs establishes that transition are non-divergent for all paths. The CSP choice operator is again represented with the \textit{next} annotation. The CSP if – then – else is represented with the \textsc{condition next} annotation. There is no equivalent annotation syntax for the recursive call, but in chapter 5, Figure 5.3, page 125 recursive calls are represented by B definitions.
### Appendix A. The Comparison Between CSP||B’s Translations and B Annotations

<table>
<thead>
<tr>
<th>CSP</th>
<th>Translation</th>
<th>Annotation</th>
</tr>
</thead>
<tbody>
<tr>
<td>${E_n \rightarrow P}_p$</td>
<td>$n; {P}_p$</td>
<td>$n / X init(P) * /$</td>
</tr>
<tr>
<td>${E_n ? x_c \rightarrow P}_p$</td>
<td>$n(x_c) / X init(P) * /$</td>
<td></td>
</tr>
<tr>
<td>${E_n \lfloor w_c \rightarrow P}_p$</td>
<td>$w_b \leftarrow n;$</td>
<td>$w \leftarrow n / X init(P) * /$</td>
</tr>
<tr>
<td>${E_n \lfloor w_c ? x_c \rightarrow P}_p$</td>
<td>$w_b \leftarrow n(x_c);$</td>
<td>$w \leftarrow n(x_c) / X init(P) * /$</td>
</tr>
<tr>
<td>${E_n \rightarrow P \boxdot Q}_p$</td>
<td>$n;$</td>
<td>$n / X init(P) \cup init(Q) * /$</td>
</tr>
<tr>
<td>${E_n \lfloor w_c \rightarrow$ \textbf{IF} $w_c$ \textbf{THEN} {P}_p \cup {w_c \rightarrow w_b}$</td>
<td>$w_b \leftarrow n;$</td>
<td>$w_c \leftarrow n / X init(P) \cup init(Q) * /$</td>
</tr>
<tr>
<td>${E_n \lfloor w_c \rightarrow$ \textbf{ELSE}${Q}_p \cup {w_c \rightarrow w_b}$</td>
<td>$\textbf{IF} w_b$</td>
<td>$\textbf{ELSE}$</td>
</tr>
<tr>
<td>${S(p)}_p$</td>
<td>$c_b := \rho[p]$</td>
<td></td>
</tr>
</tbody>
</table>

where: $P$ and $Q$ are processes

- $n$ is the $n^{th}$ operation
- $E_n$ is the action relating to the $n^{th}$ operation
- $x_c$ is an input
- $w_c$ is an output
- $\rho[x_c]$ returns the value of $x_c$
  in the environment
- $\rho \oplus \{w_c \rightarrow w_b\}$ overwrites $w_c$
  with $w_b$ in $\rho$

Table A.1.: The Relationship of B Translations of CSP Fragments to Annotations
B. Glossary

B.1. Annotations Syntax

**NEXT** - A defined set of operations enabled by the current operation.

\[ \text{OP}_i \triangleq (P_i \mid B_i) / \{ \text{OP}_j, ..., \text{OP}_k \} \text{ NEXT} /; \]

**FROM-ANY** - The initialisation and any operation may precede the current operation.

\[ \text{OP}_i \triangleq (P_i \mid B_i) / \* \text{ FROM - ANY} * /; \]

**FROM-SET** A defined set of operations that may precede the current operation.

\[ \text{OP}_i \triangleq (P_i \mid B_i) / \* \text{ FROM - SET} \{ \text{OP}_j, ..., \text{OP}_k \} * /; \]

**FROM-SET-INIT** - The initialisation and the defined set of operations may precede the current operation.

\[ \text{OP}_i \triangleq (P_i \mid B_i) / \* \text{ FROM - SET - INIT} \{ \text{OP}_j, ..., \text{OP}_k \} * /; \]

**QUERY** - The current operation gives an output, but does not change the state of the machine.

\[ \text{OP}_i \triangleq (P_i \mid B_i) / \* \text{ QUERY} * /; \]
NEXT with I/O - A defined set of operations with or without I/O enabled by the current operation.

\[ y_i \leftarrow O_{P_i}(v_i, e_i) \equiv (P_i \mid B_i) / \ast \{ O_{P_j}(v_j, e_j) \} \text{NEXT} * /; \]

NEXT INVARIANT - A predicate that holds before and after the current operation completes that is used to establish the next operation precondition hold.

\[ O_{P_i} \equiv (P_i \mid B_i) / \ast \{ \text{predicate} \} \text{NEXT INVARIANT} * /; \]

FROM-ANY WITH I/O - The initialisation and all operations enable the current operation with or without I/O.

\[ y_j \leftarrow O_{P_j}(v_j, e_j) \equiv (P_j \mid B_j) / \ast \text{FROM - ANY} \{ ?V \mid E \} * /; \]

FROM-SET WITH I/O - All defined operations enable the current operation with or without I/O.

\[ \text{/* FROM - SET} \{ OP_1, \ldots, OP_i \} \{ ?V \mid E \} * /; \]

FROM-SET-INIT WITH I/O - The initialisation and all defined operations enable the current operation with or without I/O.

\[ \text{/* FROM - SET - INIT} \{ OP_k, \ldots, OP_i \} \{ V_j, E_j \} * /; \]

SELECT NEXT - The given operations are enabled by the current operation. There selection in the implementation will depend on the criteria specified in the annotation.

\[ O_{P_i} \equiv (P_i \mid B_i) / \ast \{ \text{cond}_{j_1} : O_{P_{j_1}}, \ldots, \text{cond}_{j_{m-1}} : O_{P_{j_{m-1}}} \text{ELSE} O_{P_{j_m}} \} \text{SELECT} \text{NEXT} * /; \]
**CONDITION NEXT** - All defined operations are enable by the current operation. The first set of operations are enabled when the current operation output is true (not 0). Conversely, the second set of operations are enabled when the current operation is false (0).

\[ y_i \leftarrow OP_i \equiv (P_i \mid B_i); \quad */ \text{OPJ OPK CONDITION NEXT} */ \]

**INTERLEAVED NEXT** - The defined set of operations are all enable by the current operation and may execute next interleaved.

\[ y_i \leftarrow Op_i(v_i, e_i) \equiv (P_i \mid B_i); \quad */ \text{OPJ INTERLEAVED NEXT} */ \]

**SEQUENCE NEXT** - The first operation from the defined sequence of operations is enable by the current operation, and in turn each operation in the sequence enables its right neighbour.

\[ y_i \leftarrow Op_i(u_i) \equiv (P_i \mid B_i); \quad */ \{ Op_j, \ldots, Op_k \} \text{ SEQUENCE NEXT} */ \]
Bibliography


[GE07] N. Grant and N. Evans. Towards the formal verification of a java processor in Event-B. In McEwan et al. [MSIW07], pages 425 – 442.


Bibliography


[MS07] A. McEwan and S. Schneider. Modelling and analysis of the AMBA bus using CSP and B. In McEwan et al. [S1W07].


Bibliography


harne, S. King, M. Henson, and S. Schneider, editors, ZB 2005: Formal
Specification and Development in Z and B, pages 279 — 298, Guildford,
Index

grd, 14
EVENTUALLY FUTURE, 218
VARIANT, 217
good TRACES, 30
FROM-SET-INIT
   PICTURING, 62
CONDITION NEXT
   ANNOTATION LISTING, 111
   FALSE, 112
   TRUE, 111
   EXAMPLE CONTROLLER, 121
   EXAMPLE MACHINE, 121
   INITIAL-CONSISTENCY, 113
   OPERATION
   PO, 110
   SYNTAX, 110
CONDITIONAL NEXT
   CONTROLLER, 112
FROM ANY
   EXAMPLE CONTROLLER, 64
   EXAMPLE MACHINE, 63
FROM ANY
   CONTROLLER, 44
FROM INTERLEAVED
   ANNOTATION LISTING, 137
   PO, 136
FROM-ANY
   INITIALLY
   PO, 42
   LISTING FUNCTION, 46
   OPERATION
   PO, 42
   SYNTAX, 41
   PICTURING, 62
FROM-ANY WITH I/O
   LISTING FUNCTION, 90
   OPERATION
   PO, 87
   SYNTAX, 87
FROM-SET-INIT
   LISTING FUNCTION, 46
   OPERATION
   PO, 44
   SYNTAX, 43
FROM-SET-INIT WITH I/O
   LISTING FUNCTION
   OPERATION NAME, 91
   OPERATION SET, 91
   OPERATION
   PO, 89
   SYNTAX, 89
FROM-SET
   EXAMPLE CONTROLLER, 64
   EXAMPLE MACHINE, 64
   LISTING FUNCTION, 46
   OPERATION
   PO, 43
   SYNTAX, 43
   PICTURING, 62
FROM-SET WITH I/O
   LISTING FUNCTION
   OPERATION NAME, 90
   OPERATION SET, 90
   OPERATION
   PO, 88
   SYNTAX, 88
I/O
   CONTROLLER, 91
INTERLEAVED NEXT
   ANNOTATION LISTING, 137
   CONTROLLER, 138
   PO, 133
   SYNTAX, 133
NEXT IN Variant
   ANNOTATION-CONTROLLER CONSISTENCY, 1
   EXAMPLE MACHINE, 100
   MACHINE-ANNOTATION CONSISTENCY, 101
   OPERATION
   SYNTAX, 79
NEXT

July 6, 2008
INDEX

ANNOTATION CONSISTENT, 68
STEP-CONSISTENCY, 68
SYNTAX, 68
I/O
SYNTAX, 91
IF-THEN-ELSE
TRACE, 112
CSP, 10, 177
DIV, 200
ACTION PREFIX, 11
CHANNELS, 11
CSPM, 10, 213
DIV, 11, 16
EXTERNAL CHOICE, 11
FAILURE-DIVERGENCE, 200
Handel-C TRANSLATION, 160
HIDING, 11
INDEX EXTERNAL CHOICE, 11
INTERLEAVING, 11
INTERNAL CHOICE, 11
INTERRUPT, 11
RECURSIVE DEFINITION, 11
RENAMING, 11
SKIP, 16
SKP, 11
STOP, 15
TRACEs, 10
CSP, 4, 13, 177, 225
CSP-Z, 178, 198, 200
Esterel, 177, 197
Event-B, 177
DYNAMICS, 210
ESTABLISH, 207
HARDWARE MODELLING, 201
MAINTAIN, 207
REFINEMENT, 205
VARIANT, 206, 210
FDR2, 177
FPGA, 4, 179
Guarded Substitution Language (GSL), 6, 23
Handel-C, 5, 177
CHANNELS, 216
SYNTAX, 12
HDL, 4
I/O
GUARDED FUNCTION, 92
INIT FUNCTION, 92
INITIAL-CONSISTENCY, 92
STEP-CONSISTENCY, 93
IF-THEN-ELSE
GUARDED FUNCTION, 113
INIT FUNCTION, 112
INITIAL-CONSISTENCY, 113
STEP-CONSISTENCY, 114
SYNTAX, 112
INDEX INTERNAL CHOICE, 11
INITIAL-CONSISTENCY, 22
INITIALLY-CONSISTENT, 24
INTERLEAVING
ANNOTATION CONSISTENCY, 140
INIT FUNCTION, 139
INIT-PAR FUNCTION, 139
STEP-CONSISTENCY, 141
INTERRUPT
GUARDED FUNCTION, 45
INIT FUNCTION, 45
INITIAL-CONSISTENCY, 47
STEP-CONSISTENCY, 48
SYNTAX, 44
TRACE DEFINITION
ANY, 46
SET, 46
SET-INIT, 46
LEMMA
CONDITION NEXT
ANNOTATION ENABLED TRACES, 118
INITIAL ACTION, 120
FROM
ANNOTATION ENABLED TRACES, 55
INITIAL ACTION, 60
SINGLETON TRACES, 51
INTERLEAVED NEXT
ANNOTATION ENABLED TRACES, 150
DISJOINT OPERATIONS, 134
ENDING INTERLEAVING, 147
INTERLEAVING, 149
SINGLETON TRACE, 155
NEXT
240
July 6, 2008
INDEX

ANNOTATION ENABLED TRACES, 34
INITIAL ACTION, 35
SINGLETON TRACE, 33
QUERY
good, 70
TERMINATION, 71
I/O
ANNOTATIONS ENABLED TRACES, 97

LTL, 215

MACHINE, 7
ANY, 213
COMPOUND MACHINE, 184
INCLUDES, 183, 207
INITIALISATION
PO, 9
INVARIANT, 184
PROCESS MACHINE (PR), 184
REFINEMENT, 186
SEES, 183
SELECT, 14
SIMPLE MACHINE (SM), 184
SYNTAX, 23
VARIABLES, 184
MACHINE CONSISTENT, 24
MACHINE DEFINITION, 21
MACHINE-ANNOTATION CONSISTENCY, 22
MACHINE-ANNOTATION CONSISTENT, 24
MACHINE-CONSISTENCY, 8, 24
MACHINE-CONTROLLER CONSISTENT, 24
MODEL CHECKING
SMV, 180
MODEL CHECKING, 181
SOLIDIFY, 180

OBDD, 180
OCCAM, 6, 12

PROB, 212
PROCESSOR
AAMP5, 181
PENTIUM4, 181
PIPELINED, 194
VIPER, 181
PROPERTY CHECKING, 180

REFINEMENT
ANNOTATION, 157

ALGORITHMIC, 157
EXAMPLE CONTROLLER, 165
EXAMPLE MACHINE, 164
MACHINE PO, 131
REFINEMENT CONSISTENT, 24
STEP-CONSISTENCY, 22
STEP-CONSISTENT, 24
SYNCHRONOUS RECEPITIVE PROCESS THEORY, 179
SYSTEM-C, 177

THEOREM
CONDITION NEXT NON-DIVERGENCE, 116
FROM NON-DIVERGENCE, 50
NEXT INTERLEAVED NON-DIVERGENCE, 143
NEXT NON-DIVERGENCE, 30
QUERY NON-DIVERGENCE, 69
I/O NON-DIVERGENCE, 95
THEOREM PROVING, 181
ACL2, 181
HOL, 181
Penelope, 181
PVS, 181

TRANSLATION
EXAMPLE HANDEL-C, 174
GUIDE, 176

VERILOG, 179
VHDL, 179
MAP, 184
PACKAGES, 184
PROCESSES, 9
SENSITIVITY LIST, 9
SIGNAL ASSIGNMENT, 9
SIMULATION, 9
TESTING, 179

WATCH DOG TIMER, 199
WEAKEST PRECONDITION (WP), 13

Z, 177
SCHEMA, 199

July 6, 2008