A Model Driven Methodology for the Construction of Reliable Concurrent Software

Author: Raúl Nestor Neri ALBORODO
Lic. in Computer Sciences

Supervisor: Julio MARIÑO CARBALLO
Ph.D. in Computer Sciences

A thesis submitted in fulfillment of the requirements for the degree of Doctor of Philosophy in Software, Systems and Computing

September 24, 2019
Declaration of Authorship

I, Raúl Nestor Neri ALBORODO, declare that this thesis titled, “A Model Driven Methodology for the Construction of Reliable Concurrent Software” and the work presented in it are my own. I confirm that:

- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed:

__________________________

Date: September 24, 2019
Abstract

Escuela Técnica Superior de Ingenieros Informáticos
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software

Doctor of Philosophy in Software, Systems and Computing

A Model Driven Methodology for the Construction of Reliable Concurrent Software

by Raúl Nestor Neri ALBORODO

The software development process now requires a working knowledge of parallel and distributed programming. The requirement for a piece of software to work properly over some shared resources is a universal must nowadays. Users want instantaneous and reliable results. In many situations the user wants the software to satisfy a number of requests at the same time. Software must be designed to take advantage of computers that have multiple processors and must be designed to run correctly and effectively.

However, such systems are hard to program, and ensuring quality by means of traditional testing techniques is often useless as errors may not show up easily and reproducing them is hard. Even though testing is one of the most widely used approach to partial system validation, the introduction of concurrency makes exhaustive testing extremely costly, and impossible in some scenarios. Due to that, industry is trying to shift to formal verification techniques.

This thesis proposes a model-driven methodology (guided by Design by Contracts (DbC) technique) for designing and verifying safety-critical concurrent system implementations using shared resources. This methodology is divided into three phases: analysis and design, code generation and testing and verification.

The whole process starts from a mathematical specification of the shared resource defining the precise behaviour of the shared resource using the well-know DbC technique. Then, it is automatically translated into TLA in order to check that some concurrency properties hold, for instance, being free of deadlocks, and that there is no inconsistency in the specification. If no error was detected, the next step is the generation of traceable code by following a set of templates for implementing the shared resource in several languages (like Java, Erlang, Ada, etc.).

Once the model has been validated, a certified code in a variety of programming languages can be generated. This is achieved by dividing the system into client process (light code) and the shared resource (heavy code) where all the concurrency specific constructs are placed. This method is language independent, non-intrusive for the development process, and improves the portability of the resulting system.

Even though the methodology is language-independent, this thesis is mainly focused on the problem of generating concurrent Java code from high-level interactive models (Erlang implementations are also included in this book showing that models can be easily translated even when the language is functional). Using an extension of the Java Modeling Language for specifying shared resources as Java interfaces, the process describes how to translate those formal models into shared memory (using a priority monitors library) or message passing (using the JCSP library) implementations. The code generation process can be fully automatic or semi automatic, but in both cases the obtained code is traceable. Both, the validation of the JML model and part of the code generation process, are constructed over the existing and well-known JML tool-set.

During this phase, a set of well defined proof obligations are added as runtime assertion checking code (RAC), to each of the patterns to ensure that the generated code is correct. After that and because the programmer still retains the power of coding or modifying parts of the code for the resource, an instrumentation process followed by the verification of the instrumented code is presented using KeY verifier.
Finally, in the case that the code of the shared resource is not obtained using this methodology, the presented methodology concludes by presenting a property-based testing technique developed using Erlang and Erlang QuickCheck. This phase is composed of two solutions that are iterative and inclusive: (i) testing an implemented system in order to check if it faithfully conforms to the resource specification on which it is based; (ii) testing a network of shared resources, i.e. a set of shared resources that have to be assembled together to perform a task and communicate with each other following a (well-formed) protocol.
El proceso de desarrollo de software requiere ahora un conocimiento práctico de la programación paralela y distribuida. El requisito de que una pieza de software funcione correctamente en conjunto con algunos recursos compartidos es una necesidad universal hoy en día. Los usuarios quieren resultados instantáneos y fiables. En muchas situaciones, el usuario desea que el software procese varias peticiones al mismo tiempo, con lo cual debe estar diseñado para aprovechar las ventajas de las computadoras que tienen múltiples procesadores y debe estar diseñado para funcionar correcta y eficazmente.

Sin embargo, estos sistemas son difíciles de programar, y a menudo resulta inútil garantizar la calidad mediante técnicas de pruebas tradicionales, ya que los errores pueden no aparecer fácilmente y su reproducción es muy complicada. Aunque metodologías de testing son uno de los enfoques más utilizados para la validación parcial del sistema, la introducción de la concurrencia hace que las pruebas exhaustivas sean extremadamente costosas e imposibles en la mayoría de los escenarios. Debido a ello, la industria está tratando de mudarse a las técnicas de verificación formal.

Esta tesis propone una metodología basada en modelos (guiada por la técnica Design by Contracts (DbC)) para diseñar y verificar implementaciones seguras de sistemas concurrentes críticos que utilicen recursos compartidos. Esta metodología se divide en tres fases: análisis y diseño, generación de código y testing y verificación. Todo el proceso parte de una especificación matemática del recurso compartido definiendo el comportamiento preciso del recurso compartido utilizando la conocida técnica DbC. A continuación, se traduce automáticamente a TLA para comprobar que algunas propiedades de concurrencia se cumplen, como por ejemplo, que esté libres de bloqueos, y que no hay inconsistencias en la especificación en sí. Si no se detecta ningún error, el siguiente paso es la generación de código rastreable utilizando un conjunto de plantillas para implementar el recurso compartido en varios lenguajes (como Java, Erlang, Ada, etc.). Esto se logra dividiendo el sistema en proceso cliente (código ligero) y recurso compartido (código pesado) donde se colocan todas las construcciones específicas para el manejo de la concurrencia. Este método es independiente del lenguaje, no intrusivo para el proceso de desarrollo y mejora la portabilidad del sistema resultante.

Aunque la metodología es independiente del lenguaje, esta tesis se centra principalmente en el problema de generar código Java concurrente a partir de modelos interactivos de alto nivel (las implementaciones de Erlang también se incluyen en este libro, demostrando que los modelos pueden ser fácilmente traducidos incluso cuando el lenguaje es funcional). Usando una extensión del Java Modeling Language para especificar recursos compartidos como interfaces Java, este proceso describe cómo traducir esos modelos formales en implementaciones de memoria compartida (usando una biblioteca de monitores de prioridad) o de paso de mensajes (usando la biblioteca JCSP). La fase de generación de código puede ser totalmente automática o semiautomática, pero en ambos casos el código obtenido es trazable. Tanto la validación del modelo JML como parte del proceso de generación de código, se construyen sobre el conjunto de herramientas JML existentes y conocidas.

Durante esta fase, un conjunto de obligaciones de prueba bien definidas se añaden como código de comprobación (en forma de aserciones en tiempo de ejecución) a cada uno de los patrones para asegurar que el código generado es correcto. Luego, y debido a que el programador aún conserva el poder de implementar o modificar partes del código para el
recurso, se presenta un proceso de instrumentación para su posterior verificación del código instrumentado utilizando KeY Verifier.

Finalmente, en caso de tener código de un recurso compartido que no ha sido generado mediante este enfoque, la metodología presentada concluye con la presentación de una técnica de testing basada en propiedades desarrollada utilizando Erlang y Erlang QuickCheck. Esta fase se compone de dos soluciones que son iterativas e inclusivas: (i) probar que un sistema para comprobar se ajusta fielmente a la especificación y (ii) probar que una red de recursos compartidos, es decir, un conjunto de recursos compartidos que deben ensamblados para realizar una tarea y comunicarse entre sí siguiendo un protocolo (bien definido).
I would like to extend thanks to the many people, in many countries, who have so generously contributed to the work presented in this thesis and eased my stay really far from home.

Firstly, I would like to express my sincere gratitude to my enthusiastic supervisor Julio Mariño Carballo for the continuous support of my Ph.D studies and related research, for his patience, motivation, translations from Argentinian Spanish to Spanish Spanish, and his immense knowledge. His guidance helped me with the researching and writing of this thesis (and in fact, all papers because of my terrible English). I can not have imagined having a better advisor and mentor for my Ph.D studies.

I am also hugely grateful to Ángel Herranz Nievas and Lars-Åke Fredlund for sharing their great expertise so willingly, and for being so focused/dedicated to me as unofficial supervisors.

My sincere thanks also goes to Dr. Ilya Sergey (UCL, UK), Dr. Irene Preti (EIT Digital Trento Node, IT), Ms. Justina Senkus (EIT Digital London Node, UK) and Dr. Alessandro Cimatti (Fondazione Bruno Kessler, Trento, IT) whose provided me with an opportunity to join their teams as intern. They treated me as one of their team members making me feel like home. Without their precious support it would not be possible to conduct the research geomobility that we have planned within the EIT program.

I would like to thank my family for supporting me spiritually throughout my whole stay in Madrid and for giving me through moral and emotional support throughout my life, allowing me to be as ambitious as I wanted supporting me in any step I take, no matter how small or great it was, they were and are always there for me. I am also grateful to Ángel and my other family, my in-laws, for considering me in like their son.

And most importantly, I would like to thank my life partner Vir for everything (especially for helping me while writing in English). Her support, encouragement, (tons of) patience, unwavering love, and tolerance with my occasional moods is a testament in itself of her unyielding devotion and love.

A special mention also goes to Guillermo Golzman for helping me out when I was a resident at ‘Colegio Mayor Argentino’, making me feel like I was at home. Also to those friends who are extremely closed even when you are 11k kms away from home, such as Ezequiel and Marcos among others, and to those whom I had the fortune of meeting during my long stay in Madrid.

And finally, last but by no means least, also to everyone in the Babel Group, EIT and IMDEA Software institute. It was great sharing the workplace with all of you during last years.

Thanks for all your encouragement and the support!
# Contents

Declaration of Authorship iii  
Abstract v  
Abstract in Spanish vii  
Acknowledgements ix  

1 Introduction 1  
\hspace{1em} 1.1 A model driven approach for concurrent systems using shared resources . . . 3  
\hspace{1em} 1.2 Related work .......................................................... 6  
\hspace{1em} 1.3 Preliminary knowledge ................................................. 11  
\hspace{1em} 1.4 Book organization .................................................... 16  

2 Shared resources 19  
\hspace{1em} 2.1 A syntax for shared resources ........................................ 22  
\hspace{1em} \hspace{1em} 2.1.1 A simple semantics for shared resources ..................... 24  
\hspace{1em} 2.2 Shared resources as labeled transition systems ................. 27  
\hspace{1em} \hspace{1em} 2.2.1 LTS definition for shared resources .......................... 29  
\hspace{1em} \hspace{1em} \hspace{1em} Readers&Writers expressed as LTS .......................... 30  
\hspace{1em} \hspace{1em} \hspace{1em} Buffer (1) expressed as LTS ............................... 32  
\hspace{1em} \hspace{1em} \hspace{1em} Bounded Counter as LTS ................................... 32  
\hspace{1em} \hspace{1em} \hspace{1em} Parking slots to LTS ......................................... 32  
\hspace{1em} 2.3 A Client-aware Semantics ........................................... 32  
\hspace{1em} \hspace{1em} 2.3.1 Concurrent Histories for clients ................................ 33  
\hspace{1em} \hspace{1em} 2.3.2 Protocols .......................................................... 34  
\hspace{1em} \hspace{1em} 2.3.3 Adding Scheduling policies .................................. 34  
\hspace{1em} 2.4 Networks of Shared Resources ...................................... 37  
\hspace{1em} \hspace{1em} 2.4.1 Sorting in parallel: MergeSort using shared resources .... 37  

3 From shared resources to TLA+ specifications: model validation 41  
\hspace{1em} 3.1 Structuring the shared resource translation ........................ 41  
\hspace{1em} \hspace{1em} 3.1.1 Describing the inner state of the resource .................... 42  
\hspace{1em} \hspace{1em} 3.1.2 Types and predicates ........................................... 42  
\hspace{1em} \hspace{1em} 3.1.3 Operations as set of TLA Actions ............................. 42  
\hspace{1em} \hspace{1em} 3.1.4 A shared resource becomes a dynamic system ............. 43  
\hspace{1em} 3.2 Validation of the model ............................................... 43  
\hspace{1em} \hspace{1em} 3.2.1 Internal verification ............................................. 43  
\hspace{1em} \hspace{1em} Checking the invariant ........................................... 44  
\hspace{1em} \hspace{1em} 3.2.2 Adding processes to complete the system ................... 44  

4 Code Generation based on patterns 45  
\hspace{1em} 4.1 Extending JML for shared resources ............................... 46  
\hspace{1em} 4.2 Model-based Code Generation Using Synchronized Methods .... 48  
\hspace{1em} \hspace{1em} 4.2.1 Airport Tower Controller implementation .................... 51  
\hspace{1em} 4.3 Model-based Code Generation Using Priority Monitors ......... 53  
\hspace{1em} \hspace{1em} 4.3.1 Trivial Cases ...................................................... 54  
\hspace{1em} \hspace{1em} Readers & Writers Implementation ............................ 54  
\hspace{1em} \hspace{1em} 4.3.2 Parameter Indexation ......................................... 58
D Testing implementations 149
   D.1 The Warehouse Shared Resource Coded in Erlang 149
   D.2 The Permissive Scheduling Policy 150

Bibliography 153
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>An overview of the presented model-driven methodology.</td>
<td>4</td>
</tr>
<tr>
<td>1.2</td>
<td>Air traffic controller system design based on shared resources: processes (airplanes) are shown as circles, shared resources (air traffic controller) by the larger squares (with rounded tops) and operations (asking for a free runway to depart or landing) by the smaller squares which are connected to the aircraft who has requested the permission.</td>
<td>4</td>
</tr>
<tr>
<td>2.1</td>
<td>Key aspects of concurrency from our perspective.</td>
<td>19</td>
</tr>
<tr>
<td>2.2</td>
<td>Concurrent Counter graphically expressed as shared resource withing a set of processes that want to increment o decrement its value.</td>
<td>20</td>
</tr>
<tr>
<td>2.3</td>
<td>Concurrent Counter behaviour illustrated as a labeled transition system (LTS). Notice that if there is no maximum or minimum value, the LTS has infinite states.</td>
<td>20</td>
</tr>
<tr>
<td>2.4</td>
<td>Concurrent Bounded Counter behaviour illustrated in a LTS. Notice that if there is no maximum or minimum value, the LTS is infinite.</td>
<td>20</td>
</tr>
<tr>
<td>2.5</td>
<td>The one element buffer graphically expressed as shared resource and a set of processes.</td>
<td>21</td>
</tr>
<tr>
<td>2.6</td>
<td>Buffer with only one element shown as a transition system. Notice that the machine has as many states as elements of the alphabet (domain).</td>
<td>21</td>
</tr>
<tr>
<td>2.7</td>
<td>Air traffic control behaviour illustrated in a non deterministic automaton. This defines the language of all possible traces of the shared resource based on the specified synchronization conditions. The small values attached to each edge label of <code>askForLanding</code> represents the assigned runway received by the invoker.</td>
<td>22</td>
</tr>
<tr>
<td>2.8</td>
<td>Air traffic controller specification according to the notation introduced in [Her+09]. CADT is an acronym for Concurrent Abstract Data Type and represents a shared resource.</td>
<td>23</td>
</tr>
<tr>
<td>2.9</td>
<td>Readers&amp;Writers specification with preconditions according to the notation introduced in [Her+09].</td>
<td>30</td>
</tr>
<tr>
<td>2.10</td>
<td>Readers&amp;Writers specification with preconditions and using boolean flag for writers according to the notation introduced in [Her+09].</td>
<td>31</td>
</tr>
<tr>
<td>2.11</td>
<td>Air Traffic Controller specification difference according to the notation extended in [Fre+16]. This gives priority to those aircraft that are waiting to land.</td>
<td>35</td>
</tr>
<tr>
<td>2.12</td>
<td>Robots FIFO scheduling for the Warehouse example following the notation presented in [Her+09; Fre+16].</td>
<td>36</td>
</tr>
<tr>
<td>2.13</td>
<td>General FIFO scheduling for a shared resources following the notation introduced in [Her+09; Fre+16].</td>
<td>36</td>
</tr>
<tr>
<td>2.14</td>
<td>MergeSortComp specification according to the notation introduced in [Her+09].</td>
<td>38</td>
</tr>
<tr>
<td>2.15</td>
<td>Sorting an array of four elements using several MergeSortComp shared resources.</td>
<td>39</td>
</tr>
<tr>
<td>3.1</td>
<td>TLC configuration definition of the system to be checked.</td>
<td>44</td>
</tr>
<tr>
<td>4.1</td>
<td>Java generic Interface template with JML annotations for a given shared resources.</td>
<td>47</td>
</tr>
<tr>
<td>4.2</td>
<td>The Air tower controller example specified in JML using the defined extension.</td>
<td>49</td>
</tr>
</tbody>
</table>
4.3 Generic template for implementing an operation of a shared resource using synchronized methods. ................................................................. 50
4.4 Airport Traffic Controller implemented using synchronized methods from the interface in Figure B.2.2. ................................................................. 52
4.5 Generic template for implementing a shared resource using priority monitors when CPRE does not depend on any parameter. ................................. 55
4.6 Readers & Writers implementation using priority monitors with unoptimized unblocking code. JML annotations were only added to write operations (beforeWrite () and afterWrite ()), PRE verification code is also omitted. ........................................ 56
4.7 Readers & Writers unoptimized unblocking code. ........................................... 57
4.8 Readers & Writers implementation using priority monitors with optimized signaling code and readers priority. JML ensures is simplified but equivalent to the one presented in Figure 4.7. .................................................. 59
4.9 Signaling table for the Readers & Writers monitor implementation with writers priority. .................................................................................. 60
4.10 Signaling table for the Readers & Writers monitor implementation with readers priority. ........................................................................... 60
4.11 Generic template for implementing a shared resource using the parameter indexation technique. ................................................................. 61
4.12 An implementation sketch of the Warehouse resource Figure B.3. The invariant is omitted in the specifications to make the specification easier to read. .................................................................................. 62
4.13 Generic template for implementing a shared resource using the client indexation technique. With a slight notational abuse, we have used the outcome of a Java method call (entry.getKey ()) to instantiate the value of one CPRE with a set of actual parameters. A similar “trick” will be used in Figure 4.22. ........ 63
4.14 Shipping Warehouse implementation using JCSP with channel replication technique. ................................................................................... 67
4.15 Graphical representation of our message passing approach: the wavy lines indicate channels permanently present; the dotted lines indicate channels created on demand by an operation method, if needed, and only used during the operation. .................................................................................. 65
4.16 Readers & Writers implementation using JCSP with channel replication technique. ................................................................................... 67
4.17 Generic template for implementing a shared resource using the channel replication technique over JCSP. Channel definition based on the domain of the input parameters. ........................................................................... 69
4.18 Warehouse implementation using the channel replication technique. Channel definition based on the domain of the input parameters. Wrapper (client side) operations. .................................................................................. 70
4.19 Warehouse implementation using the channel replication technique. Server component code. ........................................................................... 71
4.20 Graphical representation of the syncCond array for the warehouse problem with two warehouses (N_WAREHOUSES == 2) and a maximum weight of two (MAX_WEIGHT == 2). Starting from the left corner, the first two positions are used for enterWarehouse CPRE whilst the other for exitWarehouse CPRE. ........................................................................... 71
4.21 Generic Wrapper Component of the deferred requests template. ............... 73
4.22 Generic Server Component of the deferred requests template. ................. 74
4.23 Warehouse implementation using the deferred requests technique using only two channels. Wrapper (client side) operations. ....................... 75
4.24 Shipping Warehouse implementation using deferred requests. Server side message processing. ........................................................................... 76
5.1 Priority Monitor instrumented code to be recognizable and verifiable in KeY. 79
5.2 Condition instrumented code to be recognizable and verifiable in KeY. ....... 80
5.3 FSM representing the behavior of the unblocking code of *Shipping Warehouse* example. An `enterWarehouse (n, w)` invocation can only wake up `enterWarehouse (0, w)` invocations (when `w=0`) or `exitWarehouse (n-1, w)` (when `w>0`).

On the other hand, an `exitWarehouse (n, w)` can wake up `enterWarehouse (0, w)` invocations at any time and `enterWarehouse (n-1, w)` only when `n>0`.

5.4 Unclosed goal verifying `ReadersWritersCSPDeferredRequestBuggyKeY←.java`.

5.5 Unclosed and unreadable goal verifying `ReadersWritersCSPDeferredRequestBuggyKeY←.java`.

6.1 Operational semantics auxiliary functions. .......................................................... 93
6.2 Operational semantics rules for shared resources with scheduling policy. .......... 94
6.3 Operational semantics for systems. ................................................................. 94
6.4 Framework flowchart showing all components of the systems. ......................... 99
6.5 Output from the testing phase. ............................................................... 102
6.6 A Specification of MergeSort using Tool Syntax ........................................... 109
6.7 Modified Shipping Warehouse specification ..................................................... 113
6.8 Gritter: a simple publish-and-subscribe system ............................................. 114

A.1 Concurrent Counter specification according to the notation introduced in [Her+09].
Notice that this transition system has infinite states. ........................................ 121
A.2 Concurrent Counter behaviour illustrated in a minimal LTS. .......................... 122
A.3 Bounded Counter specification according to the notation introduced in [Her+09].
Notice that this transition system has many stages as natural numbers. ............. 122
A.4 Bounded Counter illustrated as an labeled transition system. ......................... 122
A.5 Bounded Counter canonical labeled transition system. .................................. 123
A.6 Parking Lot graphically expressed as shared resource and processes interaction. 123
A.7 Parking Lot specification according to the notation introduced in [Her+09]. .... 123
A.8 Parking Lot illustrated as an labeled transition system. .................................. 123
A.9 Parking Lot minimal labeled transition system. ............................................. 124
A.10 The one element buffer specification according to the notation introduced in [Her+09]. ................................................................. 124
A.11 The one element buffer minimal labeled transition system. .......................... 124
A.12 The bounded buffer graphically expressed as shared resource and a set of processes. ....................................................................................... 125
A.13 The bounded buffer example specified according to the notation introduced in [Her+09]. ................................................................. 125
A.14 The bounded buffer illustrated as an labeled transition system considering only an alphabet composed by 1 and x. ........................................ 125
A.15 The bounded buffer minimal ASM with a maximum length of MAX. ............ 126
A.16 Multibuffer graphically expressed as shared resource and a set of processes. . 126
A.17 Multibuffer example specified according to the notation introduced in [Her+09]. ................................................................. 126
A.18 Multibuffer illustrated as an labeled transition system and only considering an alphabet composed by 1 and 0. ........................................ 127
A.19 Multibuffer minimal ASM with a maximum length of N. .............................. 127
A.20 Readers & Writers graphically expressed as shared resource and processes interaction. ................................................................. 128
A.21 Readers & Writers specification according to the notation introduced in [Her+09]. ................................................................. 128
A.22 Readers & Writers specification difference according to the notation extended in [Fre+16]. ................................................................. 129
A.23 Readers & Writers illustrated in an labeled transition system. Notice that if there is no maximum or minimum value, the ASM is infinite. ........ 129
A.24 Readers & Writers minimal labeled transition system. .................................. 129
A.25 Airport minimal labeled transition system. ..................................................... 130
A.26 Warehouses and robot movements. ................................................................. 130
A.27 Shipping Warehouse graphically expressed as shared resource and its associated processes. .................................................. 131
A.28 Shipping warehouse specification according to the notation introduced in [Her+09]. ................................................................. 131
A.29 Shipping Warehouse labeled transition system. In this case, the graph is simplified by considering two warehouses and \{0, 1\} as weights. ........ 133
B.1 Readers& Writers Java interface with JML annotations using the extension presented in Section 4.1. ............................................. 136
B.2 Airport Java interface with JML annotations using the extension presented in Section 4.1. ......................................................... 137
B.3 Warehouse Java interface with JML annotations using the extension presented in Section 4.1. .................................................. 138
B.4 Readers & Writers implementation using synchronized methods from the interface in Appendix B.2.1. The invariant is omitted in the specifications to make the specification easier to read. ......................... 139
List of Tables

5.1 Summary of the results obtained when verifying several examples using KeY. Major attention is giving to detect which implementations have failures or bugs without considering the (finite and practical) time spent by the prover. 88

6.1 Test results for the students’ implementations of the warehouse example, classified by test rounds. 102
6.2 Results of Gritter Testing 115
I dedicate this thesis to my family for their constant support and unconditional love, and to my friends.
The term concurrency applies to those software systems in which various processes or execution flows interact (i.e., communicate and synchronize with each other) to perform a task, typically complex and from a nondeterministic nature. Such systems are increasingly common nowadays, partly because of the proliferation of distributed applications (e.g., in mobile applications), but also for complex control systems in areas such as automotive systems, manufacturing, medical devices, etc.

This kind of systems, called cyber-physical systems (CPS), are an orchestration of computers and physical systems [AS11; SM12; Raj+10; BG11]. They are composed of a physical part and a software part, whereby the physical part of the system sends information about itself to the software part, and the software sends information, usually in the form of commands, to the physical part. Thus, the development of cyber-physical systems requires the knowledge of several competencies relative to physical systems, and also, software systems. Because of that, any issues on those systems can often harm operators (such as airplanes autopilot, medical devices, or autonomous cars) and/or cost a fortune to repair or fix the problem (such as power grids, infrastructure controllers). Due to the complexity and the continuous usage of these systems, industries and academy are devoting more and more resources to tackle down the most fundamental issues [SAF19; Mon+16; YXL15; Poh18; Sha+14; TLY18; Cho+17]. Therefore, the development of programs that control these kind of systems cannot rely much on "trial and error".

As an intellectual challenge, CPS involves merging engineering models and methods from different areas (such as mechanics, environmental, electrical, industrial engineering) with models and methodology for computer sciences.

Using formal models in these sort of systems is fundamental because they give support in the creation of specifications that describe the true requirements of the user which are not usually identical to the stated requirements (may be ambiguous if written in natural language) and the possible connections between the parts.

By using formal methods, the unambiguity is reduced by means of formal specifications giving the possibility to prove certain properties of the model. During the implementation of the system, models can ensure that the cyber (software) as well as hardware part are consistent and satisfy the requirements specification. The use of models increases trustworthiness of the system in the sense that the system developed is known to be correct if the

---

1This technique is called design by contract (DbC) [Mey88]. Software designers should define formal, precise and verifiable interface specifications for software components, which extend the ordinary definition of abstract data types with preconditions, postconditions and invariants. These specifications are referred to as ‘contracts’.
result system is traceable. Thus, the model acts as evidence ensuring that the system satisfies the requirements regarding security, reliability and correctness.

As it is discussed in [Hal07], error or inconsistencies can be detected in earliest design artifacts, thereby leading to earlier detection and elimination of design defects. Thus, model checkers [CGP99] and other formal analysis tools can process the model and consider all possible execution paths through the system. If there is any possibility of a fault, a model checker will find it [DKW08]. In a multithreaded system where concurrency is an issue, formal analysis can exploit all possible interleavings and event orderings.

Cyber-physical systems are intrinsically concurrent: the cyber and the physical subsystems coexist in time and they could even sharing subsystems or concurrent processes. Models of concurrency in the physical world are far different from models of concurrency in software (arbitrary interleaving of sequences of atomic actions).

Focusing on the software part, concurrency is a key aspect of these (and many others) software systems but it is often one of the most common causes of failures at the same time. This happens due to the fact that programming concurrent systems is significantly more difficult and error-prone than those that are sequential. Almost every stage of programming gets worse when put in a concurrent context: programming languages (the lack of appropriate mechanisms for concurrent programming), libraries (many of them fail to be executed concurrently), debugging, specification, verification, etc. Reasoning about multi-threaded code is intrinsically complex and, as a consequence of the associated nondeterministic execution, traditional approaches to software development do not provide sufficient evidence for correctness, as errors may not show up easily and reproducing them is hard [SL05; Lu+08]. To make things worse, the development of concurrent software is affected by shortcomings in programming language [LS03] and library support, design methodologies, availability of properly trained practitioners, etc.

One of the most important challenges of concurrent programming in critical environment is that practically every well-known programming language incorporates specific mechanisms for concurrent programming but there are still some gaps or considerations to be taken. Despite decades of research, there is no satisfactory concurrency semantics for any general-purpose programming language that aims to support concurrent systems code. The Java Model is unsound with respect to standard compiler optimizations, while the C/C++11 model is nowadays too weak, admitting undesirable thin-air executions [Bat+15]. Even in those languages that give full support for concurrency, design patterns and guidelines, which facilitate the robustness of the code produced, are missing.

Besides, the validation and verification of concurrent system is not trivial\(^2\). The nondeterminism within the operating conditions associated with the industrial and critical environments make traditional testing techniques unreliable, more expensive or, in some cases, inapplicable. Due to that, the use of formal methods techniques to verify software must be encouraged. Unfortunately, most promising tools for formal verification do not provide support for concurrency and, in fact, they are usually limited to subsets of programming languages.

On the other hand, and as it was explain before, critical and CPS pieces of software are often extremely related to complex or embedded hardware. This leads to increasing the risks associated with upgrades or software porting the software from one environment to another one. This makes these systems authentic ‘fossils’ to anyone who dares to adapt them.

Also, chosen and unsuitable programming language just because it is popular (like Ruby, C or Java) can make the scenario worse. Programming safety-critical applications in those programming languages is tempting\(^3\) since there is a large number of trained developers available in the market. However, even when the language and its libraries provide a large number of different concurrency primitives, their limitations are often not well understood. Moreover, these concurrency primitives are generally low-level constructs, primarily targeting efficient, rather than safe execution, thus constituting poor choices for implementing safety-critical systems.

---

\(^2\) Validation is the process of determining whether a design meets the needs of the user, whereas verification is the process of determining whether a design meets a set of requirements, specifications, and regulations.

\(^3\) Java, for instance, cannot be used for targeted system with hard real-time constraints due, e.g. to the presence of automatic garbage collection.
1.1. A model driven approach for concurrent systems using shared resources

For instance, let us consider as an example the case of the International Airport of Son Sant Joan in Palma [Wik18], Majorca, Spain. It is an international airport located at 8 km and it is the third largest airport in Spain with two runways. The air traffic control has to handle all possible landings and take-offs but allowing only two aircrafts at a time.

An aircraft approaching to the airport has to ask the air traffic controller for a free runway to land. Thus, once the controller accepted the petition and give back the authorization (i.e. which runway to use), the aircraft has to do all the procedure required for landing. After that, the pilot must inform the controller that the assigned runway is now available again.

As can be seen, any failure in the system of the air traffic controller could be catastrophic. Assigning the same runway to two different aircrafts can make them crash. Also, ignoring the signal from an aircraft that already landed makes that runway useless because it will be always in use.

1.1 A model driven approach for concurrent systems using shared resources

In a way of alleviating some of the problems associated with developing reliable concurrent software, a complete model-based engineering for developing concurrent systems based on shared resources is proposed.

The presented approach is a model-driven engineering (or MDE for short) which aims to raise the level of abstraction in program specification and increase automation in program development [HT06; Lee08]. The idea promoted by MDE is to use models at different levels of abstraction for developing systems, thereby raising the level of abstraction in program specification. This refers to the systematic use of models as primary engineering artifacts throughout the engineering lifecycle, i.e. Models are considered as first class entities. According to Douglas Schmidt [Sch06], “model-driven engineering technologies offer a promising approach to address the inability of third-generation languages to alleviate the complexity of platforms and express domain concepts effectively”.

Some benefits of the MDE (both in general and, for the specific case of concurrent software) are: (i) formalizing (part of) the requirements reduces ambiguity in the problem statement; (ii) formal models can be the subject of experiments aimed at early requirement validation. That is, a mathematical model can be formally verified for detecting inconsistencies or other flaws; (iii) code is not written from scratch but generated or distilled (semi-automatically) from the model bringing several benefits, such as portability, (specially relevant for concurrent software production due to the volatility of certain languages) robustness against requirement changes, and the reduction of production costs bringing several benefits. One of them is portability, specially relevant for concurrent software production, given the volatility of certain languages. A second benefit is robustness against changes in the requirements – modifying concurrent code by hand may introduce more errors than re-generating it. Finally, the generative approach may reduce production costs at this stage: (iv) models can help in the validation, verification and test case generation of the code obtained from the previous phases.

An overview of the MDE approach is depicted schematically in Figure 1.1. As can be seen as columns in the figure, the construction life-cycle of a shared resource based software is split into three phases: analysis and design plus model validation, code generation and validation and testing.

Considering the airport example, all techniques used are briefly explained below and fully described in the next chapters.

Analysis, design and model validation. The main analysis and design activities are, then, identifying processes and their interactions, refining these interactions into shared resource specifications (i.e. to formally specify the actions interface, how the actions modify state and the synchronization conditions), and validating the model. Of course, this is an iterative process, so these steps should be repeated until a stable design is reached. The core idea is to distinguish active entities, in this case the aircrafts, from passive components, such as the air traffic controller, as shown in Figure 1.2. The latter represent all kinds of interaction among
Chapter 1. Introduction

**Problem**

**Interactions**

**Identify client processes**

**Shared resource model**

**Model validation**

**Code generation**

**Testing & verification**

**Analysis & design**

---

Figure 1.1: An overview of the presented model-driven methodology.

Figure 1.2: Air traffic controller system design based on shared resources: processes (airplanes) are shown as circles, shared resources (air traffic controller) by the larger squares (with rounded tops) and operations (asking for a free runway to depart or landing) by the smaller squares which are connected to the aircraft who has requested the permission.

The former, and are modeled using an abstraction called *shared resource* that contains a clear interface – which can be invoked from processes – and a transactional transition semantics.

As can be seen in Figure 1.1, the shared resource model is the main outcome of the analysis and design phase, as it takes part in most of the relevant activities of the software development process. For that model, the formal mathematical language introduced in [Her+09] is used.

Afterwards, it proceed to translate the model and, thus, validate it using the TLA language. Formal model validation is crucial to prevent certain concurrency issues before the code generation phase. Typically, properties such as liveness, absence of deadlock, etc., are much easier to analyze on a mathematical model than on actual code written in a programming language. In [Her+09] it is shown one way of doing this using model checking techniques.

This phase is described in detail later in Chapter 2.

**Code generation.** Once the model has been validated, a certified code in a variety of programming languages can be generated, e.g. Ada [CHM13], Java [Mar+19], Erlang, etc. The code obtained for the aircrafts (client process code in Figure 1.1) is considered *light* in the sense that it is assumed to be free from concurrency-specific constructs and is, thus, easier to verify, more portable, and does not require specially trained programmers to develop or test. In the figure, the dashed line from the client processes identified to the client processes’ code indicates that we are not considering the use of any formal activity here, i.e. the code is obtained by “traditional” means.

On the other hand, the code from the air traffic controller (interaction code in Figure 1.1) is considered *heavy* code, as it is here where all the concurrency specific constructs are placed.
1.1. A model driven approach for concurrent systems using shared resources

It is convenient then, that this code is carefully distilled from validated designs so that it can be regenerated if requirements change, rather than modified by hand. Code generation is the main formal activity discussed in Chapter 4 describing how to translate the shared resource model into Java code using shared memory mechanisms, such as synchronized methods (Sec. 4.2), or priority monitors [HM12] (Sec. 4.3), and message-passing (Sec 4.4) using JCSF (Java Communicating sequential processes [WAB12]) third-party library. In all cases, the structure of the code is derived from the logical structure of the interaction model, so that simple interaction models produce simple code patterns, while complex interaction models result in more intricate code.

Both the specifications and the code generated are Java files annotated using the Java Modeling Language (JML) [LBR99]. For specific shared resource descriptions, a minimal extension of JML is used devised to express shared resources as annotated Java interfaces that will be introduced in the following chapters.

The general idea of distilling code from specifications can be approached in different ways. A fundamentalist approach is conceivable in which code is automatically obtained from a specification compiler, ensuring correctness by construction and greater maintainability (through code regeneration as changes in specifications occur). For the class of concurrent programs considered in this book such an automatic translation may result in inefficient scheduling of processes, leading to unacceptable process response times. Thus, here a more realistic view is followed in which the translation is only semi-automatic (schemas for code generation) and the programmer still retains the power of coding or modifying parts of the code for the resource.

As the code thus obtained might be less efficient than the code generated by an expert programmer and is not automatically correct, there is still the need to verify its compliance to the original specifications, so supporting this verification task is also a crucial requirement for the translation.

One important property of the translation process is its traceability, i.e. the correspondence between code and specification must be evident. This greatly simplifies the formal verification process, as the structure of the code is not arbitrary – as it would be if the code had been produced by traditional means – but follows a well-known pattern or template. This fact is used to define a set of proof obligations that must be checked at certain points of the code. A set of the proof obligations are forced to be added (for each code template as RAC) and demonstrate how these properties can be checked using an existing program verification tool (KeY [Ahr+05; BHS07; Ahr+14]). It is important to note that KeY is a verification system for single threaded (sequential) Java code that knows nothing about concurrency. Therefore, some tricks are needed in order to use it as a tool for verifying concurrent code.

Model-based testing. Of course, another possibility for assuring the conformance of an implementation with the original specification is to use testing. As part of the presented methodology, the shared resource formal specification is used as a test oracle to decide if a resource implementation behaves correctly or not. As it can be seen in the correspondent experiment section, even if the critical code has been structured according to a shared resource specification, average programmers can still make mistakes that often have to do with subtleties of the target language synchronization primitives. Therefore, the user must at least test, systematically, the resulting implementation. The third component of the methodology is then the extensive use of property-based testing to (i) use a small finite-state model to automatically generate, semi-randomly, intelligible test cases, and (ii) to automatically decide whether test execution is successful (using the declarative specification as a base). The testing tool used, Quviq QuickCheck [Art+06], a variant of the well-known QuickCheck tool [CH00], has excellent support for testing stateful code. Essentially a model of the system is built under test (the shared resource), and use this model both to derive tests, and to judge the correctness of the execution of the system under test by comparing it with the execution of the system model.

Although tests are derived in part randomly, this does not mean that we miss the possibility of deriving good test suites with a desired coverage according to some testing coverage measure. We can semi-randomly search for a set of test cases that fulfill certain coverage criteria, e.g. regarding states and transitions in the model machine which represent a subset...
of the shared resource transition semantics. Indeed, there is experimental evidence that suggests that combining the two approaches, random testing and coverage directed test case generation, yields better results than applying any of them in isolation [Sta+12].

To test a shared resource using this testing framework three components are needed: (i) a model implementation of the resource in the Erlang programming language, (ii) a definition in Erlang of the scheduling policy to use during testing (several standard policies are predefined), and (iii) a QuickCheck state machines that decides which calls to issue to the shared resource under test.

The aforementioned methodology is perfectly applicable to networks of shared resources when each component is considered in isolation. However, ensuring the correctness of the whole network of processes and resources is a different issue.

Because of that, the testing phase was, then, improved by extending the target to a network of shared resources. This is accomplished by the extensive use of property-based testing to (i) use a small finite-state model to automatically generate, semi-randomly, intelligible test cases, and (ii) to automatically decide whether test execution is successful (using the declarative specification as a base). This work meets a functional selection criteria in which the system captures isolated functionalities or requirements within the definition of a environment entity in order to constrain the set of all executions of the model. MBT is hence concerned with the model of both the SUT and the environment.

To evaluate the effectiveness of the approach, it was applied it to the task of specifying and verifying several prototypical concurrent safety-critical systems, such as the already presented airport example, and more. The most interesting ones are presented in Appendix A.

The term “safety” is used throughout this thesis as it is now explained. Some implementation of a shared resource is unsafe when it allows the execution of a resource operation from a state where some of its preconditions do not hold, or conversely, after executing some operation that leaves the resource in a state incompatible with the operation’s postcondition or the resource invariant. Sometimes, it also generates tests aimed at detecting liveness issues, mostly the case where some threads invoking operations on a resource are blocked unnecessarily (lack of concurrency), cannot be given an upper bound on the number of resource operations served before being given access to the resource once the preconditions are met (starvation), or some priority requirement is not met (largest job first, first-come-first-served, etc.).

1.2 Related work

Over the past decades, software researchers and developers have been creating abstractions or modeling techniques that help them focusing on programming only their model. These lead to an abstraction of their work from the complexities and constraints of the execution environments(such as hardware, network devices, etc.). In early programming languages, there was an extreme connection between the code and the machine (Assembly for instance) but little by little this coupling has been reduced, creating high-level languages. However, models are still really useful at in order to obtained a reliable implementation on a system.

As it was pointed out in [Sch06], the evolution of computer sciences and creation of technologies for modeling, increased the level of abstraction used through the software life-cycle. One of the most important milestone was the creation of computer-aided software engineering (CASE) [Kuh89] techniques, in the 1980s. It was focused on the development of methodologies and tools for the software design phase using graphical representations, such as state machines and data-flow diagrams. This brought significant improvements in software quality and productivity, although these have been less than predicted by early advocates of CASE: the improvements that had been achieved are of the order of 40%, based on [Huf92].

These ill results might be derived from ugly models or representations, and due to the fact that these techniques lack support for proving quality-of-service properties, such as fault tolerance and security, among others. Therefore, it was almost mandatory to manually

---

4 As it will be explained in Chapter 2, we consider two kinds of preconditions. Invoking an operation in a state that violates the synchronization precondition (denoted CPRE), blocks the calling thread (until the CPRE holds). Traditional preconditions (denoted PRE) should be treated as exceptions when violated.
1.2. Related work

add non-trivial code to compensate those problems as well as adding all concurrent mechanisms. Thus, the final source code was really hard to debug, to trace to the initial model and evolving over time was extremely complicated even in some simple (but not trivial) scenarios. Also, CASE was unable to scale to production-scale systems and it does not provide for interaction with other team members. As a result, CASE had relatively little impact on commercial software development during the 1980s and 1990s. Nevertheless, since that approach came up, several intents were done in order to enhanced this software development methodology.

As a consequence, a promising approach to alleviate the aforementioned problems (platform dependency, complexity, etc.) was developed: the Model-Driven Engineering (MDE). It describes software development approaches in which abstract models of the systems are created and systematically transformed to concrete implementations. As it is shown in [FR07], MDE offers a way of bridging the problem-implementation gap. This approach combines the following factors:

- domain-specific modeling languages in which the user can quickly formalize the structure, system behavior, and requirements. These, are usually described using metamodels that can be mapped into the elements of the type system of the target language;
- generators or translators that (partially) analyze the obtained models and then synthesize various types of components, such as source code, simulators or just other model representations serving as inputs for other techniques/tools. This (semi) automated transformation process is often referred to as ‘correct-by-construction’;
- methodologies or hints for testing and verifying any piece of code that was filled manually.

MDE can be considered as a relatively new engineering technique with some expectations and challenges that must be addressed in the next years [Mus+14; Sil15; HWR14]. Nevertheless, these techniques have gained the attention of practitioners and theoreticians at a point to claim for a patent [GL01] of usage of this kind of methodologies. Moreover, the huge amount of published work about case studies in both academic and industrial settings is a sign of the relevance of this way of software development. From the academic perspective, several case of studies of successfully MDE application can be found, such us the development of multi-agent systems [GNC12], a model-based performance prediction at software development time [Sim+04], verification of the completeness of requirements [DC16] and its applicability to microgrid models [JCC]. In addition, several examples of the application of these techniques in industrial environments are found in companies such us Motorola [BLW05] or the ones compiled in [HWR14]

What’s more, a variety of tools and techniques have been developed and improved over the past decade for MDE. The most important publications from the academical environment are [AVT06] for integration projects, General Modeling Environment (GME) [Led+01] that supports the creation of domain-specific modeling and program synthesis environments, ProjectIT [Vid07] integrates contributions from requirements and model-driven engineering as well as in [MVT07], for web-applications development in [SS10], a XIS-Mobile domain specific language [RS14], or AtomPM [SYR+] which is an open-source framework for designing and managing domain-specific environments. Creol [JOY06] proposes a formal model of distributed concurrent objects based on asynchronous message passing that can be similar to our CSP model presented in Chapter 4. Also, CoBosex [SP08] unifies active objects with structured heaps giving another example of designing methodologies for concurrent applications. VeriCool [SJPO8] gives yet another approach to the automatic verification of concurrent Java-like small programs based on a set of rules which limits thread inference to synchronization points. From the commercial point of view, some options are available like Obeo Designer5, Eclipse Modeling Project and the JetBrains MPS6, among others.

There is an increasing consideration of the usage of model-driven engineering approach for automatic code generation as can be seen in [KLM15]. As it is explained in the introduction, there are practical (e.g. efficiency) reasons for letting the programmer retain the power

---

5 Official website http://www.obeodesigner.com
6 Some of the plugins are from the academia sector while others are from commercial partners.
Chapter 1. Introduction

to modify the code. However, our approach constrains the programmer’s freedom by fixing
the overall structure of the code (syntactic structure of the template) and the proof obligations
that are verified using KeY. This helps with the traceability from the specification to the
code.

Nevertheless, this methodologies are far from being perfect. One of the most common
issue is the non-standardization of modeling languages that is reflected into a diversity of
philosophies and models of the same problem having no possibility to compare them. It may
seem that an ugly MDE technique is more likely to contribute to the complexity of software
development [Sil15; Sim+04; HT06]. That is why, the formalism showed in this thesis is as
simple as possible (but yet powerful) in order allow translations to any formalism used as an
input of another DE methodology.

Frameworks similar in spirit to the one presented in this work, exist also in large indus-
trial context (such as [BLW05] that follows a similar approach to the one presented in this
work) and in the commercial domain. One example is Dezyne [BV15], developed by the
embedded software company Verum (www.verum.com). It comprises a set of model-driven
software engineering tools that helps developers formalize, create and verify pieces of soft-
dware (sequential or concurrent). They present an overall approach starting from a “mental
model” (as they called it) over which the general properties can be verified, such as a model
being deadlock free, livelock free, and complying to its interface model, towards several
techniques for automatic code generation. Our approach involves the translation of the Java
annotated interface to TLA in order to the check the aforementioned properties. Then, the
methodology focuses on the design of templates to automatically generate code but at this
moment, our templates cover just Java, Ada and Erlang languages as output. However, both
approaches (ours and Verum’s) offer a strong traceability from models to code.

Also, Uppaal [LPY97] offers a tool for modeling and verifying real-time systems using
timed automata extended with integer variables, structured data types, user defined func-
tions, and channel synchronization. It also offers a way of synthesizing the model of the con-
troller component of code. When the model is verified, the user needs to resort to some third
party tool for code generation like the one presented in [Paj+12]. Our approach presents
support during the whole system development life cycle, having different techniques for
modeling, code generation and testing but traceable during the whole process.

As the first part of this work, the modeling part is presented. Modeling is still a dif-
ficult task that requires a certain know-how because of the amount of formalisms and the
no connection with top-used programming languages. This is even more critical in concur-
rent programs. Those involve a collection of processes whose behaviour heavily depends on
their interaction with each others and shared memory, and or their reactions to the environ-
ment stimuli. And also, considering that a minor syntactic change may lead to significantly
different semantic models, the presented approach should be as simple as possible.

Programmers would like to generate implementations in which several properties are
hold [Kin94]. The more important of them are safety (correctness), such us absence of dead-
locks, mutual exclusion or invariant check, and liveness properties (completeness), for ex-
ample starvation-freedom. Those properties can be easily check at the modeling phase.

Taking into account the aforementioned properties and in order to design a more straight-
forward methodology, is mandatory to truly understand concurrency to give a concise syn-
tax and semantic for those systems. From our point of view, concurrency can be easily ex-
pressed as a conjunction between non-deterministic simultaneous tasks execution (threads
that perform some kind of work or computation like workers or actors), and the interaction
between them (the way they exchange data). Thus, in order to coordinate the interaction
of those processes, we should define an interface for shared resources depicting how those
processes will communicate with them, i.e updating the resources inner state through its
interface, using shared memory mechanisms or message passing. In addition, it is also im-
portant to consider how the synchronization of external invocations over those objects will
be done with the propose of avoiding race conditions or invalid invocations. This can be
achieve by implementing semantically non-blocking and local properties like linearizabil-
ity [HW90] or blocking properties as serializability [Pap79; BHG86; WV01]. In terms of
more practical results, blocking properties, such us e.g mutual exclusion or synchronization
by condition [Her+09], are simpler to implement and the most practically used solution.
1.2. Related work

Works in depicting a better way of expressing this system as a model had been published during the last decade in top conferences. In [Sun+09], a way of modeling concurrent CSP-based systems is done as procedural codes. They propose a new language mixing high-level modeling operators with low-level programs. The purpose of this is to simplify the system modeling and its later verification. They modeled data operations as terminating sequential programs, which then can be composed using high-level compositional operators as it was similarly done in this work but keeping state-based specifications. However, they do not show how traceability is achieve during the specification and verification process. In [Xin+11], a technique for identifying the behavioural changes of concurrent programs with LTS-based semantic model is presented. In this work, the robust graph matching techniques is exploited to determine an optimal correspondence relation over the states and their transitions. They claimed to be one of the forerunners of exploiting the model differencing techniques to support the development and verification of concurrent programs, since the modeling phase.

Both works use a starting point LTS as their operational semantic models, which provide the basis for automatic system analysis and verification. But, they focus in calculating the differences between two or more LTS to determine possible inconsistencies on the models. Our methodology presents a better approach by simplifying the connections among the different states of the shared resource and the clients. Besides, it tries to understand the different models in LTS in order to match those whose behaviour are similar.

So that, our methodology is strongly based on the work of Arnold [AN82; Arn94]. His work on synchronization vectors gives us the basis for our semantic model for client aware systems and for those involving more than one resource. This methodology was widely used in modeling several concurrent systems such as the CSP rendezvous channels and objects [Fau87], distributed systems [HF89].

During the thesis, some process calculi were also considered such as CSP [Hoa78]. It is a formal language for describing interactions in concurrent systems using based on message passing via channels. In this specific case, interactions patterns where not taking as much importance as the blocking mechanisms and the history of those blocked threads. The major difference with CSP is that, at this point, our methodology is language and mechanism independent, i.e. concurrent instructions and how processes communicate to each other is unknown.

π calculus [MPW92] is a mathematical model of processes whose interconnections change as they interact and it was born as an evolution of the Calculus of Communicating Systems (CCS)[Mil82]. Another result for a semantic is the one presented in [CSW97]. It is based on the late version of the π-calculus in which any communication is divided into two stages (first, synchronization and then the value is, later, transmitted to the receptor). The model developed can be applied to other subset of the calculus due to the fact that it is purely denotational and it does not required any specific operator in the language. The semantics depicted in this chapter can be easily translated into π calculus using channels as mean of communication by considering each invocation from the client, a creation of a channel. Action input values are values written in the channel and the output can be seen as reads from that channel. However, as it was said before, the programmer will be responsible of choosing the better mechanism based on the nature of the target system.

A finite chart semantic is depicted in [BGP08] as the first try of semantics for the pi-calculus. This uses message sequence charts leading to an easier way of visualization of π calculus programs and causal dependencies that state-based relational semantics omit. Messages sequence charts obliges the user to know before hand the occurrence of some invocations in order to branch the process execution. Besides, adding more components to an existing system (clients for instance) is not straightforward.

Considering trace semantics, the usage of dynamic logic is a cutting-edge methodology as it is shown in [BB13]. This works present a combination of dynamic logic and first-order temporal logic in order to give a trace semantics without taking into consideration the shape of the trace formulae. The major contribution is the ability to express flow properties in a concurrent environment. Notion of bisimilar states/objects in public observations. Our idea of using LTS with synchronization vector is to keep as simple as possible the model and the generation of traces is given for free.
Traces of concurrent objects can be also described as free partial commutative monoids. In [Vas92], they focus on the definition of a simple model where objects communicate via asynchronous messages and with the capability of creating new objects, sending messages and reacting to incoming messages. The theory of traces provided two different and isomorphic ways of viewing concurrent transitions: dependence graph based on events (nodes) and synchronization messages, and equivalence classes of sequences of events.

[Sta96] defines a translation and an interpretation for $\pi$-calculus considering the strong, late semantics for process behaviour. By having simple equations and a sophisticated symmetrical monoidal closed category make any further work, upon this theory, simple and intuitive. [AH01] continues that work by defining a set of concrete definitions of the abstract denotational framework in Stark’s paper. This work points out those richer definitions are necessary for most of the program analysis that relies on the concrete semantics of the $\pi$-calculus.

Trace semantics is it is shown in [HJS07; Jac04] can easily be extracted from any LTS model.

Moving to the verification part, in [Bet+05; YB02], techniques for verifying concurrent controllers are presented by distinguishing two components: concurrent controllers and their clients. Although the approach is similar to ours, their works presents verification over the specification of the concurrent controller component (very much like the use of TLA in [Her+09]) whilst our present proposal discusses black-box testing on a given Java implementation. Moreover, client interleaveings are not tested with the same level of detail as our proposal does. A widely vision on software verification can be seen in [Kro16].

Even considering that introducing formal techniques in the software development process increases its costs, some industrial environments are using them such is the case of Facebook an INFER [Cal+15].

In our methodology, KeY tool is used over dozens of other available tools such as Bogor [RDH03], TACO [Gal+10] and Slam [Bal+04]. All of them use model-checking as underlying verification technique. In principle, this limits the verification to concrete instances of the shared resources (i.e. those with a bound in the number of states of the corresponding transition system) and that is why we preferred to use a tool based on dynamic logic. However, in practice, tools based on model-checking can be a good alternative if formal verification is hard to achieve. We have discarded the first two (due to the lack of support for concurrency) but tried Slam and, according to our (limited) experience, we must say it is a robust tool and the effort required to establish a number of key properties (e.g., $\text{prop\_safe\_signal}$ and $\text{prop\_liveness}$) was similar to the effort spent using KeY.

Finally, the last phase is based on model-based testing (MBT) [JE94]. This is a software testing technique in which the test cases are derived also from a model that describes the functional aspects of the system under test (SUT).

MBT better that unit testing because the latter usually do not suffice to check all possible the functionalities. Thus, these techniques has been adopted as an integrated part of the testing process [EW02; Jor08; Law12]. Besides, higher level of automation can be achieved compared with normal or unit testing making exhaustive testing possible.

Model-based testing [BLK15] is currently a very active research area. A number of recent (mainly theoretical) proposals define frameworks for model-based testing concurrent systems [PHL14; CL13; UK97] based on previous work on testing labeled transition systems [NH84; BT01].

Our proposal tries to be fundamentally practical, and, although the underlying behaviour of a shared resource can be ultimately expressed in terms of a labeled state transition system, as it is shown in Chapter 2, the idea is to express tests at the same level of abstraction of the specifications, that is, coarse-grain resource operations, pre- and postconditions, resource invariants, etc. The semantics of the defined processes was done by means of a structural operational semantics as can be seen in [BG09; BG95; FMS02].

A similar approach is depicted in [Tre08] as he use model based testing theory for labeled transition systems. His work introduces a relation called $\text{ioco}$ to define a formal correctness between a specification and an implementation.

UPPAAL Tron [LMN04] seems to be a good proposal for model-based testing over timed FSMs. One limitation of the tool is the need for (explicitly) describing all reachable states on beforehand in order to express the behavior of a given shared resource. We think that our
1.3 Preliminary knowledge

In this section, some preliminary concepts are introduced in order to make the book self-contained. Below sections are linked with the different phases of the presented methodology.

Design by contract

Design by Contract (DbC) is a programming methodology where the behavior of program components is described as a contract between the provider and the clients of the component. It was firstly introduced by Bertrand Meyer in 1986 with the Eiffel language [NS87].

It is an approach for designing software based on the definition of formal, precise and verifiable interface specifications for each component of the system. These specifications are referred to as ‘contracts’, in accordance with a conceptual metaphor with the conditions and obligations of business contracts.

The DbC approach assumes all client components that invoke an operation on a server component will meet the preconditions specified as required for that operation. Where this assumption is considered too risky (such as distributed computing) the opposite ‘defensive design’ approach is taken, meaning that a server component tests (before or while processing a client’s request) that all relevant preconditions hold true, and replies with an exception/error message if not. The contract is semantically equivalent to a Hoare triple which formalizes the obligations.

The contracts are crucial for the software correctness and they should be part of the design process. In effect, DbC advocates writing the assertions first and then moved to the other phases of the development process. The notion of a contract extends down to the method/routine level.

All class relationships are between client classes and supplier classes. A client class is obliged to make calls to supplier features where the resulting state of the supplier is not violated by the client call. Subsequently, the supplier is obliged to provide a return state and data that does not violate the state requirements of the client. Subsequently, the supplier guarantees to the client that when a delete feature finishes its work, the data item will indeed, be deleted from the buffer. The class invariant guarantees that the state of the class will be maintained within specified tolerances at the end of each method execution.

Design by Contract has become a popular methodology for every object-oriented languages. In the case of Java, the components are the program’s classes. Contracts naturally correspond with the object-oriented paradigm to encapsulate the internal state of an object.

Java Modeling Language

One of the principles of object-oriented analysis and design is that procedural thinking should be postponed as long as possible. That is why it is needed to have some form of describing the behaviour to all defined interfaces of the system.

The Java Modeling Language (JML) is an increasingly popular specification language for Java software, that has been developed as a community effort since 1999 [LBR99]. The syntax and semantics are for the greatest part already settled in [Lea+09].

The benefits of using JML regards in expressing properties about the special aspects of the Java language, e.g., inheritance, object initialization, or abrupt termination. Besides, the specification language itself heavily relies on Java; its syntax extends Java’s syntax and its
Chapter 1. Introduction

semantics extend Java’s semantics. The former makes it convenient to talk about such features in a natural way instead of defining auxiliary constructs or instrumenting the code as in other specification methodologies or languages. The latter can also come in handy since, with a reasonable knowledge of Java, little theoretical background is needed in order to use JML. This has been one of the major aims in the design of JML. It however bears the problem that reasoning about specifications in a formal and abstract way becomes more difficult as even simple expressions are evaluated w.r.t. the complex semantics of Java. Also, assertions in source code to prove correctness of the implementation have already been proposed long time ago [Flo67]. However, assertions were not widely used in practice—the assert statement in Java only first appeared in version 1.4.

Other programming languages adopted assertions earlier such as the introduced concept of Design by Contract as explained before.

With the growing popularity of Java in the 90s, several groups decided to develop a specification language for Java. Gary Leavens, at Iowa State University, used their experience and started working on a DbC specification language for Java in 1998. At the same time that his team proposed a specification language, they developed a JML runtime assertion checker, that could be used to validate the contracts at runtime.

In the same period, Rustan Leino and his team at the DEC/Compaq research centre started working on a tool for static code analysis, ESC/Java [Fla+02], that defines a specification language that was more or less a subset of JML. Its successor, ESC/Java2 [CK05], finally adopted JML as it is now. Therefore, several projects have been targeting formal verification of Java programs since then: LOOP project [BJ01], and KeY [Ahr+05].

Over the years, JML has become a very large language, containing many different specification constructs, some of which are only sensible in a single analysis technique and mostly all tools that support JML support only in fact only a subset of it. Several approaches were considered over the years in order to specify safety spots in a concurrent code [ABL08]. Moreover, JML suffers from the lack of support for current Java versions; for instance, in 2008 there were no specifications for Java 5 features.

This specification language is used to annotate shared resources Java interfaces in order to express the behaviour of each operation as well as defining the class as a shared resource entity. The extension for doing it is presented in Section 4.1. Also, it is used when verifying the code in the KeY tool (Chapter 5).

KeY

The KeY project was started in 1998 at Karlsruhe Institute of Technology as a tool for formal specification and verification of Java programs that would be useful not just for formal methods specialist but every software developers as well. Their long-term goal was to make formal methods easy to apply in a Java program by any programmer. The core of KeY is a software verification tool based on symbolic execution and invariant reasoning. Currently, KeY is developed and maintained by three different sites. In addition to Karlsruhe, these are Technische Universität Darmstadt and Chalmers University of Technology.

However, KeY is not merely a deductive verification tool only usable by specialists in formal methods, but it has developed into a versatile tool for static analysis of programs. Current research covers these areas

- generate test cases automatically that are guaranteed to satisfy certain coverage criteria
- generate counter examples to understand failed proofs
- symbolic state debugging going beyond omniscient debuggers and requiring no fixtures
- information flow analysis and leak detection/exploit generation
- automatic specification generation

Footnote: 7Formerly, University of Karlsruhe.
1.3. Preliminary knowledge

After a long research and development, more than 15 years after a tool which is robust and usable for middle to large scale code is found. KeY lets you specify the desired behavior of your program in the well-known specification language JML, and helps you proving that the given program conforms to its specification without having boundaries, i.e. verification instead of program validation.

Apart from being a formal verification tool, it also includes a debugger with novel features. The Symbolic Execution Debugger (SED) is a platform that allows to interactively debug programs based on symbolic execution. Symbolic execution discovers all possible execution paths simultaneously. This is achieved by the use of symbolic instead of concrete values resulting in a symbolic execution tree. Each node provides the full state and related information such as the symbolic call stack or the path conditions under which the node is reached. Execution can start at any method or at any statement without any prior setting up. Navigation is realized using classic debugger functionality, including stepwise execution or breakpoints.

Furthermore, an automated test case generator is also integrated within the tool. This is extremely useful when checking why the error happened.

This tools is extensively used for the verification of the generated code (see Chapter 5). Even though, KeY works only on sequential programs, a way of verifying concurrent programs is presented and analyzed. Below, some basic notions of the underlying techniques used by KeY are explained.

Symbolic execution

Executing a program symbolically means that the tool interprets the program with unknown, non-initialized start values or parameters. One great feature of symbolic execution is that it can start at any code location. The main drawback is that symbolic execution splits easily, at least at every explicit case distinction, but also, for example, at each object access of the form $o.n$ due to the fact that the behavior of that code differs depending on whether $o$ is null or not.

For instance, consider the following piece of code that swaps the values of $x$ and $y$ without allocating a temporary variable:

```java
// int x, y, z;
x = y - x;
y = y - x;
x = x + y;
z = x / y
```

after swapping the values, the result of dividing $x$ by $y$ is assigned to $z$. After the variable declaration, $x$ and $y$ are initialized with unknown integer values, say $x_0$ and $y_0$, respectively. Symbolic execution of the first line updates the value of $x$ to $y_0 - x_0$. Thus, this is the value that $x$ has on the right hand side. Therefore, the new value for the $y$ variable is $y_0 - (y_0 - x_0) = x_0$, and so on.

When computing the last statement $z = x / y$, an `ArithmeticException` might be thrown depending on whether the initial value was zero or not. As no further information is available, symbolic execution has to consider both possibilities and thus splits the execution into two.

Loop invariants

Consider the following loop that updates every item of the array by its position:

```java
for (int i = 0; i < arr.length; i++) {
    arr[i] = i
}
```

If `arr` is a non null arbitrary array, then it is not possible to know beforehand the length of it (`arr.length`). The loop can be unwound the loop body one or more times, but it is not clear when to stop. KeY uses loop invariants to deal with this kind of situations. A loop invariant is a property that holds (i) when the loop is encountered and (ii) is preserved by
any execution of the loop guard and body. In the example above, the simplest loop invariant written in JML is:

```java
/*@ loop_invariant
@ (i >= 0 && i <= a.length &&
@ \forall int j; 0<= j && j < i; a[j] = j ));
@*/
```

It gives the bounds of the loop counter variable i and specifies that the array has been partially initialized up to the current value of the loop counter. Since at the exit of the loop i==a.length holds, after the loop finishes the array is now fully initialized. This fact can then be used during the analysis of any remaining code that comes after the loop. In simple cases (such as the one above), loop invariants can be derived automatically, but in general the designer must supply a suitable invariant.

Design-by-contract

To render symbolic execution feasible, the code is broken down into manageable pieces that are analyzed separately. In Java the most natural division of analysis is considering each method as a separate portion of the code. KeY follows the specification methodology called “Design by Contract” [Mey88; Mey92] that was explained before.

In this case and because KeY uses Java as the target programming language, the ‘precise’ behaviour of each method is captured in a contract written in JML. Expressions of JML are written in structured Java comments, next to the declaration they relate to. For instance, a contract for a binary search method can be written as follows:

```java
/*@ public normal_behaviour
@ requires
@ \forall int x; (\forall int y;
@ 0 <= x && x < y && y < arr.length; arr[x] <= arr[y]));
@ ensures
@ (\exists int x; 0 <= x && x < arr.length; arr[x] == value) ?
@ arr[\result] == value : \result == -1);
@*/
public /*@ pure @*/ int binarySearch (int[] arr, int value) {
...}
```

The keyword `normal_behaviour` requires the method to terminate normally (i.e., without throwing an exception) if the caller of the method guarantees that at invocation time the method’s precondition (boolean JML expression after `requires`) is satisfied. Here the precondition states that array a is sorted in ascending order. In this case the method contract ensures that the result of the method is -1 if a does not contain element v or otherwise the index of element v in arr. The specification modifier `/*@ pure */` states that the method is not allowed to change the heap.

Formal verification

JML annotations, Java programs, and the rules of symbolic execution are all translated automatically to formulas and rules of the program logic that is used by KeY. From the programmer point of view, it is unnecessary to understand how it is done. Suffice it to say, when loading a class, annotated with JML contracts, into KeY, a proof obligation formula is generated and loaded into its theorem prover. Then the user selects a method and its contract and attempts to show its validity.

If the theorem prover can show the validity (all generated goals are closed) of the proof obligation formula relating to a given contract and method, then we know the following is true: if the method is started in any Java state that satisfies the requires clause, then upon termination, the final state satisfies the ensures clause.

The theorem prover provides a high degree of automation, but allows also interactions by the user in case that proof cannot be found automatically. Most often, interactions with the prover consist in providing additional annotations such as class or loop invariants.
If no proof is found, then the failed proof needs to be analyzed as to whether the specification is too weak or wrong, the program is buggy or the automation is not powerful enough and further interaction by the user is required.

Understanding failed proofs is thus crucial for identifying a problem. To help this process, several options are provided to inspect a proof object:

- theorem prover itself (this requires an understanding of KeY’s program logic);
- in the symbolic execution debugger (SED) which renders a KeY proof situation inside the Eclipse debugger which is clearer for pure developers;
- through counter example generation that gives the concrete input that leads to a violation of the property whose prove attempt failed.

**Erlang**

Erlang is a functional programming language that integrates support for concurrency, high reliability, distribution and fault tolerance. Erlang was originally developed for the large telecommunication systems at Ericsson. The first version was developed by Joe Armstrong, Robert Virding and Mike Williams in 1986 [Arm+96] as part of Ericsson Corporation. It was later released as an open source language in 1998.

In a nutshell, Erlang uses the actor model, and each actor is a separate process in the virtual machine. It reacts only when a message is received. Erlang forces the programmer to write actors (processes) that will share no information with others unless they pass messages to each other. Because of that, every communication is explicit, traceable and safe.

Erlang, as well as QuickCheck, is the language used for implementing the testing tool.

**Quviq QuickCheck**

QuickCheck (QC)[CH00] is a combinator library originally written for Haskell programs to assist in software testing by generating test cases for test suites. For the SUT (system under test), the programmer writes assertions about logical properties that a piece of code must fulfill. Then, QC attempts to create several test cases trying to violate these assertions. If a test case has been found, the system tries to minimize it by simplifying input data.

This tool was then developed for several programming languages like C, Java, Erlang, among others. In this book, we are using Quviq QuickCheck [Art+06].

Quviq QuickCheck uses the Erlang functional programming language [Arm+96; CT09] to express correctness properties and test models. This does not mean that the tested software must be written in Erlang; a good interface library for C code has e.g. permitted the testing of AUTOSAR components and infrastructure on a commercial basis [Sve+12]. In this book, it is shown how to test Java implementations of shared resources in QuickCheck by using the JavaErlang interface library [EF15].

QuickCheck receives a specification instead of test cases. Those specifications, written in Erlang language, express general properties your system should always satisfy. In fact, those sentences are generators. A generator could be seen as a formula which describes the system behaviour in a specified set of cases. Each property can generate many different test cases, i.e, specifications are more concise and maintainable than really test suites. At the same time, many more cases can be generated, so testing is more thorough. QuickCheck uses the power of functional programming to keep specifications concise and readable. QuickCheck has a powerful framework, making it easy to generate complex data with not so complicated properties, with tight control over the distribution of test cases (using frequency operator).

Consider the case in which SUT is a digital clock and the programmer has to generate test-cases for it. The next generator `digital_clocks()` will generate any time of day:

```
digital_clocks () ->
  {choose (1,12),
   choose (0,59),
   oneof ([am,pm])}.
```
However, when the tool call list (digital_clock ()) generates sequences of different times to test the SUT. After that, a controlled random generation of test cases is performed in order to test the SUT against the spec.

Conventional random testing generates failing cases that are loaded with unnecessary details, making the diagnosis of the fault costly to decipher. This tool simplifies test cases to minimal examples automatically or by proposing a simplification strategy.

The simplest approach for modeling the SUT is using a state machine. QuickCheck provides a library to model the state declaratively writing pre and post conditions predicates for each operation declared in the. Based on that, the tool will generate minimal call sequences in order to provoke violations in the specification.

**QuickCheck state machines**

For checking “stateful” code, QuickCheck provides a state machine library. Here the tested “object” is not a simple boolean property, but rather a sequence of function calls each with an associated post condition that determines whether the execution of a call was successful or not. A QuickCheck state machine has a state, obviously, which can be understood as the model state of the system under test. Given a model state, the library generates a suitable next API command, and proceeds to execute the call, checking, after the call has been completed whether the result was the expected one w.r.t the given model state of the state machine. Next, a new model state is computed, and the generation of commands and their execution is repeated, until a test sequence of sufficient length has been generated and tested. In other words, the QuickCheck state machine acts as a model for the program under test.

In order to use the state machine library a user has to supply a “callback” Erlang module providing a set of functions with predefined names. The functions defined in the callback module are called by QuickCheck during test generation and test execution. The functions that should be implemented by a tester are enumerated below.

```erlang
initial_state ()
command (State)
precondition (State,Call)
next_state (State,Result,Call)
postcondition (State,Call,Result)
```

The model state is initialized by the initial_state function, and is updated by next_state. API calls are generated by the function command, which returns symbolic calls of the form [call,ModuleName,Function,Args], which are then executed. The postcondition function checks that the return value of a call is correct, considering the current model test state.

### 1.4 Book organization

The book is divided into two main parts. The first part introduces and develops every core phase of the model-driven methodology as well as presenting several examples of shared resources and solved problems. The second part, the appendices, show all the pieces of code and specifications that are obtained by applying this methodology.

Chapter 2 introduces the definition of a shared resource. Their syntax and semantics are presented and illustrated with examples. An equivalence for them is presented as well based on the blocking states/conditions of each operation. Besides, a simple semantic is depicted that guaranteeing linearization and other properties such as liveness. Then, an extension focused on the two-step operation (one signal when the operation starts execution and one when finishes) is presented. Then, it is extended to described a whole system with an unique shared resources (we aggregate the previous one with the notion of clients). This chapter ends up showing a brief overview of networks of shared resources.

Chapter 3 presents a systematic translation of the given shared resources specification into TLA language. Thus, the generated model can be easily checked using TLA+ tools. Also, an automatization of this process is presented in a tool called ‘RaZoR’.
Chapter 4 shows a set of templates for transforms the shared resource specification into runnable code. Each template is a combination of immutable parts and placeholders that must be manually (or can be derived automatically) filled in manually by the user. This is the starting point for verifying the obtained code in the next phase.

Chapter 5 illustrates a novel technique for instrumenting and verifying the obtained Java code. In this phase the well-know KeY verifier is used. For each code generation template, a pattern to instrumented the obtained code is illustrated by example.

Chapter 6 describes a framework for testing a class of safety-critical concurrent systems implemented using shared resource specifications. Then, it is extended to generate test cases for a network of shared resources.

Chapter 7 highlights the novelties of this thesis and discusses the relation with other current developments or techniques being carried out in other institutions.
Shared resources

From a pragmatic point of view, concurrency can be defined as the coordination and management of independent lines of execution (see figure 2.1 for an overview of the most important aspects of concurrency). These executions can be truly parallel or simply be managed by interleaving\(^1\). Those simultaneous execution of processes are done in a non-deterministic pattern but with a composition of interactions between processes. This interaction can be divided into communications mechanism or synchronization techniques over the use of external resources. In this case, that interaction is specified separately as processes or actors and points of interaction that we called shared resources: shared entities used by more than one thread but (usually) can only be operated on by one thread at a time.

\[
\begin{align*}
\text{concurrency} &= \text{simultaneous execution} + \text{nondeterminism} + \text{interaction} \\
\text{interaction} &= \text{communication} + \text{synchronization} \\
\text{synchronization} &= \text{mutual exclusion/serializability} + \text{conditional synchronization}
\end{align*}
\]

**FIGURE 2.1:** Key aspects of concurrency from our perspective.

Basically, a shared resource has an encapsulated state, and defines a set of operations which provides the only mechanisms to modify or access the resource internal state, following [Lam00]. It can be seen as a central controller to serialize conflicting requests. Such operations are guaranteed to be executed atomically from an observational point of view. This means that the concurrent execution of two calls to any action of a certain resource is either forbidden (mutual exclusion is preserved) or, if allowed, the observed behaviour of any concurrent set of calls is equivalent to some sequential execution of them as the linearizability principle states [HW90].

Let us introduce a simple example such as a concurrent counter: an encapsulated integer with only two operations, one to increase the value of the counter and the second to decrease its value\(^2\). As can be seen in Figure 2.2, the presented technique distinguishes processes (light blue circles labeled as $Inc_i$ and $Dec_j$, increasers and decreasers threads, respectively) from the shared resource itself (big yellow rectangle identified with the name of the resource). Each thread/process will only communicate or interact with the shared resource through the atomic operations (gray rectangles labeled as $increment$ and $decrement$).

---

\(^1\) Concurrency is not parallelism, although it enables parallelism. If the host machine has only one processor, the program can still be concurrent but it cannot be parallel.

\(^2\) A growing set of examples, including specifications, documentation and (verified) code is accessible at [AMH14] and in the appendixes.
Chapter 2. Shared resources

The invocation of these actions on resources will have a total effect that will be visible globally and serializable, i.e., any pair of actions executed must be equivalent to an interleaved execution of them.

Following the example, it is easily to see that the communication between the threads, no matter which operation are invoking, happens strictly through the internal value of the counter. From that, a labeled transition system defining the language of the counter can be obtained as shown in in Figure 2.3. The correspondence between shared resources, as formalizations of inter-process interaction, and labeled transition systems will be fully specified in Section 2.2.

Notice that all possible traces or interleavings are valid due to the fact that the only restriction here is the mutual exclusion. The language of paths (i.e. action sequences) for a resource where the only restriction on actions is the linearizability criterion is the free one. In the case of the counter we have $L_{\text{counter}} = (\text{inc}() \mid \text{dec}())^*$. However, in a more realistic situation, not all possible interleavings are valid. In those cases, the shared resource is obliged to synchronize all the petitions received of actions invocations.

Let us consider a bounded counter in which its value cannot be negative. The graphical representation remains the same as in figure 2.2 because from the perspective of the clients, its operations do not change. But now, if a ‘decreaser’ reaches the counter in a value 0, it must wait until a ‘increaser’ is executed as it is shown in figure 2.4.

Notice that there is no communication between the processes (no value is passed as an argument or returned from the shared resource) but there is a synchronization mechanism. The shared resource will not allow to execute a decrement (the invoker is blocked) if the inner value is 0. In this case, it is mandatory to use some formalism similar to a counting

---

3 Notice that this is transparent from the process point of view. All synchronizations is governed by the resource.

4 This example is similar to simulate a parking lot (see Appendix A.3) in which the owner keeps count of the parked cars.
automaton \cite{Kut88} obtaining a more difficult language such as the following:

\[ L_{\text{bounded-counter}} = \{ \alpha | \forall \text{dec}(\) \in \alpha \bullet (\exists \text{inc}(\) \in \alpha \bullet \text{pos}(\text{inc}(\)) < \text{pos}(\text{dec}(\)) \} \]

where \text{pos} is a function that returns the position of the given atom in the string.

Even though there is no communication among threads in the previous example (only synchronization), the most common scenario is when both are needed: the shared resource has to synchronize the calls and the interaction of the processes is made through it. To illustrate this behaviour, let us introduce the classic example of a one element buffer.

A bounded buffer (figure 2.5) of one element is a simplification of the classic example of synchronization problem. A producer generates data to be put in the buffer by calling \text{put}. Analogously, consumers remove the data from the buffer. The problem focuses on ensuring that the producer will not add data into the buffer when it is full and that the consumer will not remove data from an empty buffer (see figure 2.6).

Considering \text{put}, the invoking thread is allowed to perform it if and only if the inner element of the shared resource is empty (or nil). Thus, any call to \text{put} will be blocked until there is free room to place the item. This is an example of conditional synchronization. Section 2.1 will explain how to express conditional synchronization in our formalism.

The communication from consumers to producers is done by modifying the state of the shared resource. In this example, having only one producer or many (analogously for consumers) will be invisible from the point of view of the resource due to the fact that the condition is only based on its internal state.

Let us end this presentation with a more complex example taken from the real world. The International Airport of Sant Joan in Palma \cite{Wik18}, Majorca, Spain, is located 8 kms. from the city and is the third largest airport in Spain with two runways. The air traffic control has to handle all possible landings and takes off but allowing only two aircrafts at a time.

In order to illustrate this example, the air traffic control can be seen as a shared resource as is shown in Figure 1.2). An aircraft that approaches the airport has to ask the air traffic controller for a free runway to land (by calling \text{askForLanding ()}). Thus, once the controller accepts the petition and gives back the authorization (i.e. the runway identifier), the aircraft has to do all the procedure for landing and then tell the controller that the given
runway is available/free, by calling afterLanding (runway). The procedure for taking off is analogous.

Notice that not all interleavings are valid executions. If three aircrafts want to land or take off there must be some sort of scheduling: two aircrafts might proceed but at least one will wait until one runway become available again. Because of that, it is mandatory to define a conditional synchronization as a restriction on the set of interleavings (sequences of uses of a resource) in order to decide if it is possible for an aircraft to land or to take off. The shared resource needs to verify that there is an available runway before authorizing the pilot to take off (analogously for landing). This behaviour is shown in Figure 2.7

Having three aircrafts approaching to the airport leads us to the following valid traces

\[ \text{askForLanding}() / 1, \text{askForLanding}() / 2, \text{afterLanding}(2), \text{askForLanding}() / 2, \text{afterLanding}(1) \]

i.e, the last aircraft that asked for the permission has to do aerobatic maneuver and wait until a runway become free. Therefore, there is no possibility of having the following trace

\[ \text{askForLanding}() / 1, \text{askForLanding}() / 2, \text{askForLanding}() / 2, \text{afterLanding}(2), \text{afterLanding}(1) \]

because the controller cannot assign the same runway to two aircraft without producing an imminent crash.

2.1 A syntax for shared resources

In [Her+09], a shared resource notation is introduced to express inter-process interaction by defining: (i) an encapsulated state (domain), (ii) an interface of observationally atomic actions, and (iii) a state transition semantics. The core idea of having this notation is the definition of a language that is compact, formal and non-ambiguous, and to allow us to express a (potentially infinite) state machine. This technique is language independent and allows the user to precisely define the problem and the behaviour of the component, use patterns for code generation (see Chapter 4), study and analyze the correctness of the solution and its properties (see Chapters 5 and 6). Our specification language is based on first-order logic, which is sufficiently known not to need but a quick brush-up in most cases.

Its core ideas are inspired on a simplification of well-known formal methods (notably VDM [Jon95]) with additional constructions to address concurrency. Following [Lam00], the specifications are state-based, with the state accessible only through a set of public operations.

---

5 The value next to each invocations represents the return value of the operation. In this particular case, it is the identifier of the runway. The communication is specified as the change in the state of the resource, together with the return value of the action.

6 In fact, the language for specifying shared resources is largely parametric on the actual logic used. The one used here is a subset of the Z mathematical toolkit [Spi89].
2.1. A syntax for shared resources

CADT AirTrafficController

OPERATIONS
- ACTION askForLanding: N[0]
- ACTION afterLanding: N[i]
- ACTION askForTakingOff: N[0]
- ACTION afterTakingOff: N[i]

PROTOCOLS:
- Landing : askForLanding; afterLanding
- TakeOff : askForTakingOff; afterTakingOff

CONCURRENCY:

SEMANTICS

DOMAIN:
- STATE: self = Sequence(boolean)
- INVARIANT: Length(self) = N
- INITIAL: ∀ i ∈ 0..N • !self[i]

PRE: true
CPRE: ∃ i ∈ N • !self[i]
- askForLanding(runway)
  POST: self[runway] ∧ (∀ i ∈ 0..N • self[i] = self^in[i] ∨ i = runway)

PRE: !self[runway]
CPRE: true
- afterLanding(runway)
  POST: self[runway] ∧ (∀ i ∈ 0..N • self[i] = self^in[i] ∨ i = runway)

PRE: true
CPRE: ∃ i ∈ N • !self[i]
- askForTakingOff(runway)
  POST: self[runway] ∧ (∀ i ∈ 0..N • self[i] = self^in[i] ∨ i = runway)

PRE: !self[runway]
CPRE: true
- afterTakingOff(runway)
  POST: !self[runway] ∧ (∀ i ∈ 0..N • self[i] = self^in[i] ∨ i = runway)


Figure 2.8: Air traffic controller specification according to the notation introduced in [Her+09]. CADT is an acronym for Concurrent Abstract Data Type and represents a shared resource.

Following the notation introduced in [Her+09], the specification for the air traffic controller is presented in Figure 2.8. It starts by defining a new Concurrent Abstract Data Type (CADT) with name AirTrafficController, i.e. creating a new shared resource with the given name. Then, all operations must be listed. These definitions can be seen as signatures for an interface to the environment, and must include a name, a set of formal parameters and a set of return values, if needed. In this case, askForLanding action returns a natural number (N[0], identifier representing the assigned runway) whilst afterLanding must be invoked by passing a natural number (N[i], runway to be released).

After that, the notation allows the user to define a protocol, i.e. a set of rules and guidelines for communication between the shared resource and the external processes. Protocols specify the valid interactions between the communicating entities. For instance, an aircraft cannot ask for landing and immediately ask for taking off – it must first finish landing and then, ask for a free runway again.

The specification continues by depicting the DOMAIN section. It contains type definitions for the resource and, optionally, an invariant which restricts the values of the resource state to

---

7 More introductory examples of the notation are given in Appendix A.
8 Notice that only types are explicit because the name of the parameters does not matter at this point.
9 Protocols will be defined and explained in detail in Section 2.3.2.
those which are admissible in the problem. Basic types include booleans, naturals, integers, and reals. Complex types are built on them by means of algebraic types (free types). A series of predefined non-basic types, such sequences (indexable flexible-length arrays), sets, and finite mappings, with a complete set of operations on them, are also provided.

In this case, the inner state of the resource consists of just a mapping from the identifier of the runway (an integer value) to a flag telling whether the runway is being used or not (a boolean variable suffices). We follow the convention that the type with the same name as the CADT stands for the type of the whole resource. The special variable “self” acts as a pronoun to refer to the state.

Initially, as specified in the initial clause, all runways are free. The invariant clause describes that number of runways is always the same and it is determined by a constant N.

As explained before, an aircraft that wants to land (analogously, take off) has to ask the air traffic controller for a free runway. It is the task of the (implemented) resource to ensure that the call does not return (i.e., gets blocked) until it is safe for the aircraft to land. The concurrent precondition (CPRE) is specified as a formula that must be satisfied before the action is executed. After landing, a free runway must be available, if not, the aircraft has to wait. The postcondition (POST) condition specifies the change on the resource state provoked by the completion of a call. It is possible to provide preconditions (PRE) for operations too, which specify requirements on the arguments to an operation that every call must satisfy. Once the air traffic controller gives permission to an aircraft to land, the assigned runway is marked as in used.

The return clause is added in order to express the value returned by the operation askForLanding.

2.1.1 A simple semantics for shared resources

This section presents a simple semantics for shared resources. It consists of a static part, in the form of abstract syntax, and an operational semantics based on transition rules. A more flexible approach is presented in the following section, were a translation of the abstract syntax into labeled transition systems is introduced.

Also, we will start by focusing on the behaviour of a single resource, ignoring how it can be constrained by its interactions with the environment. More elaborate semantics dealing with the interaction between resources and their client processes and networks of several resources will be proposed in sections 2.3 and 2.4 respectively.

We will not elaborate on the form of the inner state of a shared resource: it is just a value from some domain $\mathbb{D}$. This is the minimal requirement for any shared resource. On top of that, a semantics for handling concurrent aspects such us blocked threads, joints and for analyzing concurrency properties will be built in the following paragraphs. We have mentioned the use of the Z mathematical toolkit for defining domain types. In order to abstract away from the particularities of notation let us introduce a function $getDomain : Spec \rightarrow \mathbb{D}$ which extracts all possible values defined in the domain of a shared resource specification.

Definition 2.1.1 (Action names). An action name is a tuple consisting of the name of an operation defined in the resource specification plus a list of actual parameters.\(^{11}\) The set of all possible action names is thus defined as the set $\Delta$ containing the pairs of the form $(op, (k_1, \ldots, k_n))$, where $op \in Names$ (the set of operation names in the spec), $n$ is the (input) arity of $op$, $(\mathbb{D}_1, \ldots, \mathbb{D}_n)$ are the domains for $op$’s input parameter types and, for every $i$, $k_i \in \mathbb{D}_i$. We use $\alpha$ to range over a subset of $\Delta$.

Now, auxiliary predicates for pre-, post-conditions and concurrent preconditions are described in order to get a concise definition of a shared resource.

Definition 2.1.2 (Precondition predicate). Consider a given shared resource state and an action name. The predicate expressing the Pre is declared as follows:

$$Pre : \Delta \rightarrow \mathbb{D} \rightarrow \mathbb{B}$$

\(^{11}\)If RETURN clause is not specified in the operation, for instance in finishLanding, that means that the operation does not return any value.

\(^{11}\)The length of the list of arguments is directly proportional to the arity of the operation.
This predicate tells us whether the execution of the operation is legal or not.

**Definition 2.1.3** (Concurrent precondition predicate). Consider a given shared resource state and an operation with its arguments, the predicate expressing the CPRE is declared as follows:

$$C : \Delta \rightarrow D \rightarrow B$$

This predicate tells us whether the execution of the operation is legal or not.

**Definition 2.1.4** (Postcondition). Consider a given shared resource state, the next state by applying the modifications of an operation execution, we declare the postcondition as:

$$Post : \Delta \rightarrow D \rightarrow D^*$$

First, this is a partial function that is only defined for those elements of \(\Delta\) for which both the precondition and the concurrency precondition hold. In that case, the postcondition takes an action name, a resource state, and returns the new state and, possibly, an output. This is represented by a (possibly empty) tuple of elements from the output domains \((D_1, \ldots, D_m)\).

From that, the notion of a shared resource configuration is depicted in the following definition.

**Definition 2.1.5** (Shared Resource Configuration). A shared resource configuration \((shr)\) is an ordered seven-tuple of the form

$$\langle D, \Delta, \text{Pre}, \text{Cpre}, \text{Post}, \text{Init}, \text{Inv} \rangle$$

where:

- \(D\) is the domain of the shared resource;
- \(\Delta\) is a set of operations names defined in the shared resource;
- \(\text{Pre}\) is the specialization of the \(\text{PRE}\) predicate for each resource operation;
- \(\text{Cpre}\) is the specialization of the \(\text{CPRE}\) predicate for each resource operation;
- \(\text{Post}\) is predicate ruling the transformation to the following state of the resource and handling the returned value;
- \(\text{Init}\) is the predicate describing the initial state of the shared resource\(^{12}\);
- \(\text{Inv}\) is the predicate depicting the invariant of the given shared resource.

Consider again the air traffic controller example illustrated in Figures 2.8, 2.7. In this case, \(D = \{ r1 \rightarrow B, r2 \rightarrow B \}\) which represent if the runway is being used or not. \(D\) contains only the identifiers of the runways \(\{r1, r2\}\).

As starting point, both runways are free and ready to use and the invariant is true because \(r1\) and \(r2\) can be seen as boolean variables and all possible combinations of their values are valid scenarios.

As for the precondition for this example, the following set is obtained\(^{13}\)

$$\text{Pre} = \{ (\text{beforeLanding}, \text{nil}), \{r1 \rightarrow X, r2 \rightarrow Y\}, \text{true}),
(\text{afterLanding}, \{r1\}), \{r1 \rightarrow X, r2 \rightarrow Y\}, \neg X),
(\text{afterLanding}, \{r2\}), \{r1 \rightarrow X, r2 \rightarrow Y\}, \neg Y) \}$$

where \(X\) and \(Y\) are boolean values. After that, the predicates for the concurrent precondition can be expressed as below

\(^{12}\)Notice that it is a predicate expressing some restrictions over the initial state. From that, a shared resource may have several allowed initial states.

\(^{13}\)Only \text{beforeLanding} and \text{afterLanding} operations are considered. The predicates that correspond to the before\text{-}TakingOff and after\text{-}TakingOff actions are analogous to the ones presented.
\[\mathcal{C}_{\text{pre}} = \{(\text{beforeLanding}, \text{nil}), \{r_1 \mapsto X, r_2 \mapsto Y\}, \neg(X \land Y), (\text{afterLanding}, \{r_1\}), \{r_1 \mapsto X, r_2 \mapsto Y\}, \neg X), (\text{afterLanding}, \{r_2\}), \{r_1 \mapsto X, r_2 \mapsto Y\}, \neg Y)\}\]

The first tuple says that \text{beforeLanding} can proceed if and only if one of the values associated to \(r_1\) and \(r_2\) is false.

Finally, the formulas of the \text{POST} for this example are:
\[\mathcal{P}_{\text{post}} = \{(\text{beforeLanding}, \text{nil}), \{r_1 \mapsto \text{false}, r_2 \mapsto Y\}, \{r_1 \mapsto \text{true}, r_2 \mapsto Y\}, r_1), (\text{beforeLanding}, \text{nil}), \{r_1 \mapsto X, r_2 \mapsto \text{false}\}, \{r_1 \mapsto X, r_2 \mapsto \text{true}\}, r_2), (\text{afterLanding}, \{r_1\}), \{r_1 \mapsto \text{true}, r_2 \mapsto Y\}, \{r_1 \mapsto \text{false}, r_2 \mapsto Y\}, \text{true}, (\text{afterLanding}, \{r_2\}), \{r_1 \mapsto X, r_2 \mapsto \text{true}\}, \{r_1 \mapsto X, r_2 \mapsto \text{false}\}, \text{true}\}\]

Notice that in this case, the first item from the tuple is the state in which the operation is executed while the second one is the resulting state of the result. For instance, the resulting state of \text{beforeLanding} operation marks the runway as in used and return its identifier to the invoker. If the relation is undefined for a given pair of state and action, there is no possible execution of that operation in that state.

In brief, and considering, the shared resource configuration for the air traffic controller is the following
\[\{(r_1 \mapsto \mathbb{B}, r_2 \mapsto \mathbb{B}\}, \Delta, \text{Pre}, \mathcal{C}_{\text{pre}}, \mathcal{P}_{\text{post}}, \neg(r_1 \lor r_2), \text{true}\}\]

where
\[\Delta = \{(\text{beforeLanding}, \text{nil}), (\text{afterLanding}, \{r_1\}), (\text{afterLanding}, \{r_2\}), (\text{beforeTakingOff}, \text{nil}), (\text{beforeTakingOff}, \{r_1\}), (\text{beforeTakingOff}, \{r_2\})\}\]

As it is commonly done, the semantics for this component are built on top of a state-transition system and using interleaving semantics. We define a notion of single-step transition among configurations and, based on that, we will define a multi-step transition function describing all possible reachable states. The domain of this function contains all the possible enabled sequential transitions inside a share resource whilst its codomain contains all the reachable configurations from any state that make the \text{Init} predicate valid.

In an model of interleaving semantics, a system evolves by replacing a configuration by a new one by the execution of an action. When there is more that one possible execution, it is translated in more than one possible transition. We capture the replacement of configurations by the transition function defined below.

**Definition 2.1.6 (Transition Function).** Let \(S = (\mathbb{D}, \alpha, \mathcal{P}_{\text{pre}}, \mathcal{C}_{\text{pre}}, \mathcal{P}_{\text{post}}, \text{Init}, \text{Inv})\) be a valid shared resource configuration. The transition function \(\delta_S\) of \(S\) is a partial function

\[\delta_S : D \times \Delta \to D\]

such that \(\delta_S(d_0, \text{action}) = d_1\) if and only if

- \(\text{Inv}\) holds for \(d_0\) and \(d_1\),
- \(\mathcal{P}_{\text{pre}}(\text{action}, d_0)\) and \(\mathcal{C}_{\text{action}}(d_0)\) holds and
- exists a value \(k\) for which \(\mathcal{P}_{\text{post}}(\text{action}, d_0, d_1, k)\) is defined

Using the define transition function, a shared resource behaviour can be depicted as an abstract state machine. As can be seen in Figure 2.7, its definition is concise and the behaviour is more understandable.

Due to the fact that the same problem can have several representations, a minimization algorithm is presented in Theorem 1. This will lead us to obtain the canonical form for the shared resource representation.

**Theorem 1 (Minimal Shared Resource state machine).** Considering a shared resource configuration \(\mathcal{S}_{\text{shr}} = (\mathbb{D}, \Delta, \mathcal{P}_{\text{pre}}, \mathcal{C}_{\text{pre}}, \mathcal{P}_{\text{post}}, \text{Init}, \text{Inv})\), the canonical (minimal) state machine can be obtained using the following algorithm:
2.2 Shared resources as labeled transition systems

In this section, a shared resource is described as a simple model based on possibly infinite edge-labeled directed graphics where the nodes represent the reachable states of the given shared resource and the edges are the transitions or operations describing how the system evolves. This kind of models are called Labeled Transitions Systems (LTS) as they were first introduced in [Kel76].

**Definition 2.2.1 (Keller Labels).** Let Lab be a countable set of labels (or actions), ranged over by ℓ, possibly indexed, which includes a special action τ denoting an invisible or internal activity.

**Definition 2.2.2 (Keller labeled transition system).** A labeled transition system (LTS for short) is a triple $(Q, A, \rightarrow)$ where:

- $Q$ is the nonempty, countable set of states, ranged over by $q$ (possibly indexed);
- $A \subseteq \text{Lab}$ is the countable set of labels (or actions), ranged over by $\ell$ (possibly indexed);
- $\rightarrow$ is a partial order relation on $A \times Q$.

P := {S};
NewStates := {S};
Edges := ∅
while (NewStates is not empty) do
    choose and remove a set A from NewStates
    for each action in $\Delta$ do and each value in dom(Pre(action)) \cup dom(Cpre(action)) do
        let X be the set of states for which a transition leads to a state in A, i.e.,
        for each set Y in P for which $X \cap Y$ is nonempty and $Y \setminus X$ is nonempty do
            replace Y in NewStates by the two sets $X \cap Y$ and $Y \setminus X$ if Y is in NewStates
            else if $|X \cap Y| \leq |Y \setminus X|$ NewS = $X \cap Y$
            else NewS = $Y \setminus X$
        end;
        add NewS to NewStates
        add {(action, value), A, NewS} to Edges
    end;
end;

Thus, the minimal state machine for the given shared resource shr is the following quintuple $\text{min}(\text{shr}) = (\Sigma, \text{NewStates}, \text{Init}, \text{Edges})$ where $\Sigma = \text{dom}(\text{Edges})$.

**Theorem 2 (SR Equivalence by blocking states/predicates).** Given two shared resource configurations, $\text{shr}_1$ and $\text{shr}_2$, are considered equivalents iff $\text{cannon}(\text{shr}_1)$ behave as $\text{cannon}(\text{shr}_2)$ for all defined operation.

That is,

- the set of states of each configuration are enumerable;
- for each state $s_1$ in $\text{States}_{\text{shr}_1}$ and a state $s_2$ in $\text{States}_{\text{shr}_2}$, if there is an edge $((\text{action}, \text{value}), s_1, s_2)$ in $\text{\Delta}_{\text{shr}_1}$, then there must also exists an edge $((\text{action}, \text{value}), s_2, s'_2)$ in $\text{\Delta}_{\text{shr}_2}$ for some state $s'_2$ in $\text{States}_{\text{shr}_2}$;
- there must exists a predicate $\text{fst}$ such that $\text{fst} \Rightarrow \text{Init}_{\text{shr}_1}$ and $\text{fst} \Rightarrow \text{Init}_{\text{shr}_2}$.

However, when considering a concurrent system, those kind of automata are not enough. It is needed to have a way of expressing those calls that are waiting or could not be executed because its CPRE did not hold at call time. In the next section, the interaction model is presented.

As it was said at the beginning, these definitions work for any shared resource. From those, a client-aware (adding notions of clients calls, see section 2.3) or a system of shared resources (Section 2.4).
→ ⊆ Q × A × Q is the transition relation defined in.

Given a transition (q, ℓ, q′) ∈ →, q is call the source, q′ the target and ℓ the label of the transition, usually abbreviated as q \xrightarrow{ℓ} q′. A rooted labeled transition system is a pair (TS, q₀) where TS = (Q, A, →) is an LTS and q₀ ∈ Q is the initial state (or root). Sometimes it is written TS = (Q, A, →, q₀) for a rooted LTS.

**Definition 2.2.3 (Path).** Lets TS = (Q, A, →), and two states q, q′ ∈ Q, a path of length n from q to q′ is a sequence of n valid transitions q₁ \xrightarrow{ℓ₁} q₁′, q₂ \xrightarrow{ℓ₂} q₂′, ..., qₙ \xrightarrow{ℓₙ} qₙ′ such that q = q₁, qₙ′ = q′, and qᵢ′ = qᵢ₊₁ for i = 1, ..., n denoted by

\[ q₁ \xrightarrow{ℓ₁} q₂ \xrightarrow{ℓ₂} \cdots qₙ \xrightarrow{ℓₙ} qₙ₊₁ \]

Notice that when n = 0, the path is empty and q = q′ = q₁.

**Definition 2.2.4 (Acyclic Path).** Lets TS = (Q, A, →), and a path q₁ \xrightarrow{ℓ₁} q₂ \xrightarrow{ℓ₂} qₙ \xrightarrow{ℓₙ} qᵢ, the path is acyclic if qᵢ ≠ qⱼ for all i ≠ j with i, j ∈ {1..n}.

**Definition 2.2.5 (Reachability Relation).** Let A*, ranged over by σ, be the set of all the strings on A, including the empty string ε. The concatenation of σ₁ and σ₂ yields σ₁σ₂, with the condition that εσ = σ = σε. The reachability relation is defined as →* ⊆ Q × A* × Q the least relation induced by the following axiom and rule:

\[ q \xrightarrow{σ₁*} q₁ \xrightarrow{σ₂*} q₂ \xrightarrow{ℓ} q₃ \xrightarrow{σ₃*} q₃ \]

It is written q₁ \xrightarrow{σ₁*} q₂ to state that q₂ is reachable from q₁ when there exists a string σ such that q₁ \xrightarrow{σ₁*} q₂.

**Definition 2.2.6 (Sort).** Lets TS = (Q, A, →) and a state q ∈ Q, sort of q is defined as the set sort(q) = {ℓ ∈ | ∃ q′* • q →* q′ \xrightarrow{ℓ} x} for some state x. The rooted LTS TS₀ = (Q₀, sort(q), →₀, q₀), called the reachable LTS from q₀, where:

- Q₀ is the set of the states reachable from q₀, i.e., Q₀ = {q′ ∈ Q • q →* q′}, and
- →₀ is the restriction of → on Q₀ × sort(q) × Q₀.

**Definition 2.2.7 (Reduced LTS).** A rooted LTS TS = (Q, A, →, q₀) is reduced if TS is exactly the reachable LTS from q₀, i.e., TS = TS₀.

**Definition 2.2.8 (Classed of LTS).** IMHO, we do not need this.

Now, some equivalences are presented. They are called strong to reflect that there is no internal or inner transaction (same as a shared resource due to the fact that only reacts on externals actions).

An overview and more detailed description can be found in [Gla01; Gla93; San11; GV15].

**Definition 2.2.9 (Isomorphism).** Let TS₁ = (Q₁, A, →₁) and TS₂ = (Q₂, A, →₂) be two LTSs. An isomorphism is a bijection f : Q₁ → Q₂ which preserves transitions:

\[ q \xrightarrow{ℓ₁} q′ \text{if and only if} f(q) \xrightarrow{ℓ₂} f(q′) \]

for all q, q′ ∈ Q₁ and for all ℓ ∈ A. If there exists an isomorphism between TS₁ and TS₂, then we say that TS₁ and TS₂ are isomorphic, denoted TS₁ \cong TS₂.

LTS isomorphism \cong is an equivalence relation, i.e., reflexive, symmetric and transitive.

This definition can be adapted to a rooted labeled transition systems by requiring that the isomorphism f preserve the initial states, i.e., f(q₀) = q₂, if q₁ and q₂ are the initial states of TS₁ and TS₂, respectively.
2.2. Shared resources as labeled transition systems

Definition 2.2.10 (Deadlock). A state \( q \) is a **deadlock** if there is no transition starting from it, i.e., \( \not\exists (\ell,q') \cdot q \xrightarrow{\ell} q' \), usually abbreviated as \( q \rightarrow^\ast \). An LTS \( TS = (Q,A,\rightarrow) \) is considered **deadlock-free** if for all \( q \in Q \), \( q \) is not a deadlock.

Definition 2.2.11 (Simulation and Bisimulation). Let \( TS = (Q,A,\rightarrow) \) be an LTS. A **simulation** is a relation \( R \subseteq Q \times Q \) such that if \( (q_1,q_2) \in R \) then for all \( \ell \in A \)

\[
\begin{align*}
\forall q_1' : q_1 \xrightarrow{\ell} q_1' \quad & \implies \exists q_2' : q_2 \xrightarrow{\ell} q_2' \land (q_1',q_2') \in R \\
\forall q_2' : q_2 \xrightarrow{\ell} q_2' \quad & \implies \exists q_1' : q_1 \xrightarrow{\ell} q_1' \land (q_1',q_2') \in R.
\end{align*}
\]

State \( q \) is simulated by \( q' \), denoted \( q \preceq q' \), if there exists a simulation \( R \) such that \( (q,q') \in R \). Two states \( q \) and \( q' \) are simulation equivalent, denoted \( q \simeq q' \), if \( q \preceq q' \) and \( q' \preceq q \).

A **bisimulation** is a relation \( R \subseteq Q \times Q \) such that \( R \) and its inverse \( R^{-1} \) are both simulation relations. More explicitly, a bisimulation is a relation \( R \) such that if \( (q_1,q_2) \in R \) then for all \( \ell \in A \):

\[
\begin{align*}
\forall q_1' : q_1 \xrightarrow{\ell} q_1' \quad & \implies \exists q_2' : q_2 \xrightarrow{\ell} q_2' \land (q_1',q_2') \in R \\
\forall q_2' : q_2 \xrightarrow{\ell} q_2' \quad & \implies \exists q_1' : q_1 \xrightarrow{\ell} q_1' \land (q_1',q_2') \in R.
\end{align*}
\]

Two states \( q \) and \( q' \) are **bisimilar** (or bisimulation equivalent), denoted \( q \sim q' \), if there exists a bisimulation \( R \) such that \( (q,q') \in R \).

Notice that the previous definition covers also the case of a bisimulation between two LTSs, \( TS_1 = (Q_1,A,\rightarrow_1) \) and \( TS_2 = (Q_2,A,\rightarrow_2) \) with \( Q_1 \cap Q_2 = \emptyset \). Consider one single LTS \( TS = (Q_1 \cup Q_2,A,\rightarrow_1 \cup \rightarrow_2) \), a bisimulation \( R \subseteq Q_1 \times Q_2 \) is also a bisimulation on \( (Q_1 \cup Q_2) \times (Q_1 \cup Q_2) \).

A rooted LTS \( TS_1 = (Q_1,A,\rightarrow,1) \) is bisimilar to the rooted LTS \( TS_2 = (Q_2,A,\rightarrow,2) \) if there exists a bisimulation \( R \subseteq Q_1 \times Q_2 \) containing the pair \( (q_1,q_2) \).

2.2.1 LTS definition for shared resources

The following definitions take into consideration a shared resource configuration presented in Definition 2.1.5 and the mathematical toolset defined for Z [Jac96, Appendix D].

Definition 2.2.12 (Shared resource as LTS). Given a shared resource configuration \( SrC = (D,\Delta,\text{Pre},\text{Cpre},\text{Post},\text{Init},\text{Inv}) \), a rooted labeled transition system for a shared resource \( MSrC \) is a tuple \( (Q,A,\rightarrow) \) where:

\[
Q = D \cup \{X\} \\
A = \{(\text{action}, \text{input}, \text{output}) \mid \text{action} \in \text{Names} \land \text{input} \in D \land \text{output} \in D \cup \{\text{void}\}\} \\
\rightarrow \subseteq Q \times A \times Q \text{ is the transition relation defined in.}
\]

A “legal” state \( q \in Q \) is a state for which \( \text{Inv}(q) \) holds and it is reachable in a finite number of steps. If a transition leads to a state from which it can never escape, this state is called a **trap state** or \( X \). There is no way of reaching the final or the accepting state from this trap or dead state.

A shared resource labeled transition system is a pair \( (TS,Q_0) \) where \( TS = (Q,A,\rightarrow) \) is a LTS and \( Q_0 = \{q_i : \text{Inv}(q_i) \text{ and } q_i \text{ is a legal state}\} \) are the initial states. For economy’s sake, a shared resource LTS with initial states will be written \( TS = (Q,A,\rightarrow,Q_0) \).

In the following, let \( Q,A,\rightarrow \), a label \( \ell = (\text{action},\text{li},\text{lo}) \), a transition \( (q,\ell,q') \) is denoted by \( q \xrightarrow{\ell} q' \) and characterized by one of the following items:

- \( \text{Pre}(\text{li},q) \land \text{Cpre}(\text{action},\text{li},q) \land \text{Post}(\text{action},\text{li},q,q',\text{lo}) \) holds (\( \text{lo} \) could be \( \text{void} \) which represents no value);
- \( q' \equiv X \) if and only if \( \text{Pre}(\text{action},\text{li},q) \) does not hold;
- if \( \text{Pre}(\text{action},\text{li},q) \) holds but \( \text{Cpre}(\text{action},\text{li},q) \) does not, then \( q \equiv q' \) because there is not possible transition in the system.
Next, the definitions of path and reachability for our notion of shared resources are specified.

**Definition 2.2.13** (Shared resource path). Let $TS = (Q, A, →, Q_0)$ and two states $q, q' \in Q$, a shared resource path is a path of length $n$ $q \xrightarrow{a} q_1 \xrightarrow{a} \ldots q_{n-1} \xrightarrow{a} q'$ if and only if is a path for $TS$, $q$ reachable from any state in $Q_0$ and $q_i \not\in X$ for $i = 1, \ldots, n - 1$.

**Definition 2.2.14** (Shared resource state reachability). The reachability relation for a shared resource is defined as $\Rightarrow^* \subseteq Q_0 \times A^* \times Q$ the least relation induced by the following axiom and rule:

$$
\begin{align*}
q \xrightarrow{ε} q & \quad \text{empty step} \\
q_1 \xrightarrow{a} q_2 & \Rightarrow q_2 \Rightarrow q_3 & q_2 \not\in X
\end{align*}
$$

A state $q_3$ is reachable if $q_1 \xrightarrow{a}^* q_3$.

Readers&Writers expressed as LTS

In order to illustrate the previous definitions, let's consider a slightly modified version of the Readers&Writers example\(^{14}\). In this case, some preconditions were added in order to prevent stop writing where there is no writer in the system as can be see in Figure 2.9.

**CADT ReadersWriters**

**OPERATIONS**

ACTION BeforeRead:
ACTION AfterRead:
ACTION BeforeWrite:
ACTION AfterWrite:

**SEMANTICS**

DOMAIN:
STATE: $\text{self} = (\text{readers} : \mathbb{N} \times \text{writers} : \mathbb{N})$
INVARIANT:
$(\text{self.readers} > 0 \Rightarrow \text{self.writers} = 0) \land$
$(\text{self.writers} = 1 \Rightarrow \text{self.readers} = 0) \land$
$(\text{self.writers} = 0 \lor \text{self.writers} = 1)$
INITIAL: $\text{self.writers} = 0 \land \text{self.readers} = 0$

**POST:** $\text{self} = (1 + \text{self.writers}^{<0n}), 0$
**CPRE:** $\text{self.readers} > 0$
**PRE:** $\text{self} = (0, 0)$
**CPRE:** $\text{self.writers} = 0 \lor \text{self.writers} = 1$
**PRE:** $\text{self.writers} = 0 \lor \text{self.writers} = 1$

**Figure 2.9:** Readers&Writers specification with preconditions according to the notation introduced in [Her+09].

Following this example and considering $\text{getDomain} (Spec_{\text{RW}}) = (\mathbb{N} \times \mathbb{N})$ (the first item represents the active readers whilst the second the writers) and $\text{getIOValues} (Spec_{\text{RW}}) = \text{void}$, the definition of a rooted LTS $TS_{\text{RW}} = (Q_{\text{RW}}, A_{\text{RW}}, →_{\text{RW}}, \{(0, 0)\})$ is:

$$
Q_{\text{RW}} = \{(r, w) \in \mathbb{N} \times \mathbb{N} | (r > 0 \Rightarrow w = 0) \land (w = 1 \Rightarrow r = 0) \land (w = 0 \lor w = 1)\} \cup \{x\}
$$

$$
A_{\text{RW}} = \{(\text{beforeRead}, \text{void}, \text{void}), (\text{afterRead}, \text{void}, \text{void}),$
\text{(beforeWrite, void, void)}, (\text{afterWrite, void, void})\}
$$

Considering $r > 0$, the transition function is defined as follow:

\(^{14}\)The unmodified problem/example, using protocols, can be found in Appendix A.7.
2.2. Shared resources as labeled transition systems

\[ \rightarrow_{RW} = \{ \]
\[
(r, 0), (beforeRead, void, void)), (r + 1, 0)),
(r, 1), (beforeRead, void, void)), X),
((r, 0), (afterRead, void, void)), (r - 1, 0)),
((r, 1), (afterRead, void, void)), X),
((r, 0), (beforeWrite, void, void)), X),
((r, 1), (afterWrite, void, void)), X),
((0, 0), (beforeRead, void, void)), (0, 1)),
((0, 0), (afterWrite, void, void)), X),
((0, 1), (afterWrite, void, void)), (0, 0)),
\}

In order to give an example of the isomorphism definition, another version of the Readers\&Writers example is presented in Figure 2.10 in which the writer set a boolean flag instead of a counter.

<table>
<thead>
<tr>
<th>CADT</th>
<th>ReadersWriters</th>
</tr>
</thead>
</table>

**OPERATIONS**

<table>
<thead>
<tr>
<th>ACTION</th>
<th>BeforeRead</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION</td>
<td>AfterRead</td>
</tr>
<tr>
<td>ACTION</td>
<td>BeforeWrite</td>
</tr>
<tr>
<td>ACTION</td>
<td>AfterWrite</td>
</tr>
</tbody>
</table>

**SEMANTICS**

<table>
<thead>
<tr>
<th>DOMAIN:</th>
<th>self = (readers : N × writers : B)</th>
</tr>
</thead>
</table>
| INVARIANT: | (self.readers > 0 \iff self.writers) ∧
| | (self.writers → self.readers = 0) |
| INITIAL: | \( \neq self.\text{writers} \land self.\text{readers} = 0 \) |

| PRE: true |
| CPRE: \( \neq self.\text{writers} \) |
| BEFORE READ |

| POST: self = (0, false) |

| PRE: true |
| CPRE: self = (0, false) |
| AFTER WRITE |

| POST: self = (0, false) |

**Figure 2.10**: Readers\&Writers specification with preconditions and using boolean flag for writers according to the notation introduced in [Her+09].

Following this example and considering getDomain(Spec_{RWb}) = (\(N \times B\))(the first item represents the active readers whilst the second the writers) and \(TS_{RWb} = (Q_{RWb}, A_{RWb}, \rightarrow_{RWb})\), \(((0, false))\) is

\[ Q_{RWb} = \{(r, w) \in \mathbb{N} \times \mathbb{B}|(r > 0 \Rightarrow \neq w) \land (w \Rightarrow r = 0) \land (\neq w \lor w = 1)\} \cup \{X\} \]

\[ A_{RWb} = A_{RW} \]

Considering \( r > 0 \), the transition function is defined as follow:
Chapter 2. Shared resources

Let us considered the bounded buffer example (Figure 2.5) but with the particularity that it can only contain two elements \{x, y\}.

The LST \(TS_{B1} = (Q_{B1}, A_{B1}, \rightarrow_{B1}, \{\text{void}\})\) for this example can be defined as follows:

\[Q_{B1} = \{\{\text{void}\}, \{x\}, \{y\}\} \cup \{X\}\]

\[A_{B1} = \{(\text{put}, \{x\}, \text{void}), (\text{put}, \{y\}, \text{void}), (\text{get}, \text{void}, \{x\}), (\text{get}, \text{void}, \{y\})\}\]

Considering \(r > 0\), the transition function is defined as follow:

\[\rightarrow_{B1} = \{(\{\text{void}\}, \{\text{put}, \{x\}, \text{void}\}, \{x\}),\]

\[\ (\{\text{void}\}, \{\text{put}, \{y\}, \text{void}\}, \{y\}),\]

\[\ (\{x\}, \{\text{get}, \text{void}, \{x\}\}, \{\text{void}\}),\]

\[\ (\{x\}, \{\text{get}, \text{void}, \{y\}\}, X),\]

\[\ (\{y\}, \{\text{get}, \text{void}, \{x\}\}, X),\]

\[\ (\{y\}, \{\text{get}, \text{void}, \{y\}\}, \{\text{void}\})\}\]

Bounded Counter as LTS

Parking slots to LTS

2.3 A Client-aware Semantics

In this section, a notion of clients, or agents, is added to complement the previous definition of LTS for shared resources. But before that, some definitions are presented and then applied to our shared resources.

Considering a path \(p\), each transition takes places in a well-defined interval. However, one cannot tell the time at which each transition takes place. One can only state that transition \(t_{i+1}\) took place in an unbounded amount of time after transition \(t_i\) (which preceded it in the path \(p\)). This manner of interpreting the sequence of transitions that make up a path is standardized as asynchronous interpretation as it is depicted in [Arn94].

**Definition 2.3.1** (Free Product of LTS). Consider \(n\) transition systems \(TS_i = (Q_i, A_i, \rightarrow_i, Q_{0i})\). The free product \(TS_1 \times \ldots \times TS_n\) is the transition system \(TS = (Q, A, \rightarrow, Q_0)\) where:

\[Q = Q_1 \times \ldots \times Q_n\]

\[A = A_1 \times \ldots \times A_n\]

\[\rightarrow = \rightarrow_1 \times \ldots \times \rightarrow_n\]

\[Q_0 = Q_{01} \times \ldots \times Q_{0n}\]
Notice that if the transition is in only one of the transition systems, the rest of the actions are marked with $\tau$ meaning an inner or invisible transition.

$$(s_1, \ldots, s_n) \in Q_1 \times \ldots \times Q_n$$ is called a global state and $$(t_1, \ldots, t_n) \in \to_1 \times \ldots \times \to_n$$ is called a global transition. The free product assumes that in a global state, all components can execute their actions simultaneously and without any restriction. However, in the case of shared resources and their clients, the must conform to the CPRE of each action before executing the action. That is why, it is needed to introduce the notion of synchronous product. Besides processes need to interact to each other, so it is mandatory to add communication and synchronization constraints.

**Definition 2.3.2** (Synchronization Constraints). Consider $n$ transition systems $TS_i = (Q_i, A_i, \to_i, Q_{0i})$, a synchronization constraint is a subset $I$ of the Cartesian product of all actions of the LTS, $I \subseteq A_1 \times \ldots \times A_n$. Each vector of $I$ represents the set of actions that can be executed simultaneously.

Notice that a synchronization constraint set is constant.

**Definition 2.3.3** (Synchronous System). A synchronous system is a vector of $n$ transitions systems together with a synchronization constraint $I \subseteq A_1 \times \ldots \times A_n$.

**Definition 2.3.4** (Synchronous Product [AN82]). Consider $n$ transition systems $TS_i = (Q_i, A_i, \to_i, Q_{0i})$, the synchronous product under $I$, written $(A_1, \ldots, A_n; I)$, is the transition subsystem of the free product of all $A_i$ containing only the global transitions that belongs to $I$.

If $I = A_1 \times \ldots \times A_n$, then the synchronous product is the same as the free product of the systems. Now, and based upon the previous definition, a client aware system can be expressed as follows:

**Definition 2.3.5** (Client-aware System). Consider $n$ transition systems $TS_i = (Q_i, A_i, \to_i, Q_{0i})$ of clients and $TS_{shr} = (Q_{shr}, A_{shr}, \to_{shr}, Q_{0shr})$ of a given shared resource. The client-aware LTS is the synchronous product $(TS_{shr}, TS_1, \ldots, TS_n)$ with a synchronization set created following the rule below (considering a vector $(I_{shr}, I_1, \ldots, I_n)$):

- if the transition $t_{shr} \neq \tau$, then it must exist a $I_i = (\text{name}, \text{input}, \text{output})$ such that $I_i = t_{shr}$ and, for all $j \neq i, I_j = \tau$;
- if $t_{shr} = \tau$, any product of transition of the clients is valid.

Notice that an $\tau$ transition in the client means no transition at all in that particular client LTS. If more than one shared resource is considered, the first rule will force a one-one transition association allowing concurrent execution over the involved shared resources.

### 2.3.1 Concurrent Histories for clients

Even though the previous definition characterized mostly every possible system using one or more shared resources, it does not consider the notion of concurrent histories (some transition can be partially blocked if the CPREs not valid). For that, a more detailed definition is needed:

**Definition 2.3.6** (Client coupling transition set). Consider a client transition systems $TS_i = (Q_i, A_i, \to_i, Q_{0i})$ and $TS_{shr} = (Q_{shr}, A_{shr}, \to_{shr}, Q_{0shr})$. The coupling transitions is a set of transition $T$ where:

$$T = \{ (x, (\text{action}, \text{input}, \text{output}), y) |$$

$$q, q' \in Q_{shr} \land x \in Q_{shr} \land$$

$$q, (\text{action}, \text{input}, \text{output}), q' \in \to_{shr} \land$$

$$(x, (\text{action}, \text{input}, \text{output}), y) \in \to_i$$

$$\}$

where $\text{\textbullet}$ is an special value called connector value from the shared resource to the client LTS.
Then, a shared resource based system with histories is defined as follows:

**Definition 2.3.7** (Histories-aware system). Consider \( n \) client transition systems \( TS_i \), a shared resource \( TS_{shr} \) and \( n \) coupling transition sets \( T_i \). The client-aware with histories LTS is the synchronous product \( (TS_{shr}, TS_1, \ldots, TS_n) \) with a synchronization set created following the rule below (considering \( t_{shr} = (q, (action, input, output), q') \)):

- if the transition \( t_{shr} \neq \tau \), then for each pair of transitions \( t_s = (x, (action, input, \Box), q), t_b = (q', (action', \Box output), y) \in T_i \) for the client \( k \) (as explained in definition 2.3.6), \( t_s = (x, (action, input, \Box), q) \) and \( t_b = (q', (action', \Box output), y) \), a brand new shared resource transition \( t'_{shr} = (q, (action', input, output), q) \) is created. Thus, two synchronization vectors are added to the set \( \{t'_{shr}, \ldots, t_s, \ldots\} \) and \( \{t_{shr}, \ldots, t_b, \ldots\} \) such that for all \( j \neq k, t_j = \tau \);

- if \( t_{shr} = \tau \), any product of transition of the clients is valid.

Besides, some restrictions are made over those synchronization vectors. The vector containing the transition \( t_b \) should be preceded by the one containing \( t_s \).

### 2.3.2 Protocols

The sequence of calls that a process is allowed to make, to shared resources and environment entities, in a well-defined order, is called here a protocol. For instance, as explained in the warehouse problem, a process controlling a robot should first ask the shared resource for permission to enter a warehouse, and then, once permission is granted, issue orders to the robot hardware to physically make the robot move. The sequences of such permitted call sequences forms a protocol. In a shared resource based system, each process forming part of the system has conceptually such an associated protocol specification.

Formally a protocol can be viewed as a non-deterministic state machine, where state transitions correspond to permitted calls to resources and environment entities.

Considering the airport example (Figure 1.2) there is an implicit protocol. A plane need to ask for permission to land first and then tell the air traffic controller that it is doing the taxi phase and the runway is free again (analogously for taking off procedure). Thus, only two patterns of calls from planes are permitted:

- \( \text{askForLanding}(); \text{afterLanding}() \)
- \( \text{askForTakingOff}(); \text{afterTakingOff}() \)

The set of those list of sequential invocations is what we called protocols.

### 2.3.3 Adding Scheduling policies

In [Fre+16], an extension of the declarative specification of a shared resource was presented in order to allow the specification of scheduling policies, i.e., to decide given multiple competing calls whose concurrency preconditions are true, which call should be executed first. As it will be explain later, having a formal and precise definition of such scheduling policies is crucial.

Considering this extension, a shared resource specification now is composed by two parts: a safety specification which provides the \( \text{PRE}, \text{POST} \), and \( \text{CPRE} \) clauses are described above, and a scheduling specification which, for each operation of the operation, defines constraints on the scheduling of calls to the method. Such constraints are specified in three auxiliary clauses that deal with the new scheduling state of the resource. These are: \( \text{ENABLED} \) describes when a call may be scheduled, provided the call, the scheduling control information associated with the call, and the scheduling state. By default, we assume the scheduling state to contain an ordered sequence of pending calls (what we call a history) and a number of predefined functions to access histories that match a certain pattern.

The airport specification in Figure 2.11 shows an example of this. The \( \text{ENABLED} \) clause specifies the additional requirement that there is no pending call to \( \text{askForLanding} \) in order to serve a call to \( \text{askForTakingOff} \). This is a way of giving priority to those aircrafts that
2.3. A Client-aware Semantics

CADT AirTrafficController

... SEMANTICS DOMAIN:

CPRE: \( \exists i \in \mathbb{N}, \text{self}[i] \)

ENABLED(call, tag): \( \text{waiting}(\text{askForLanding}) = \emptyset \)

\( \text{askForTakingOff} \)

POST: \( \text{self}[\text{runway}] \land (\forall i \in 0..N \cdot \text{self}[i] = \text{self}^{\text{th}}[i] \lor i = = \text{runway}) \)

RETURN: runway

... Figure 2.11: Air Traffic Controller specification difference according to the notation extended in [Fre+16]. This gives priority to those aircraft that are waiting to land.

want to land. For a finer control over scheduling states, two other clauses are available: WAITING which modifies the scheduling state right after a call (whose precondition is true) is invoked; and POSTWAITING which describes possible effects on the scheduling state right after executing a call.

These clauses can be seen more naturally in the warehouse example\(^{15}\). A warehouse complex is serviced by a set of autonomous robots. The problem involves a set of robots, a fixed number of warehouses and corridors, and the warehouse controller that supervised the movements of the robots. The specification can be seen in Figure A.28. A robot must first enter warehouse 0, then it may load a charge, and next it exits warehouse 0 and enters the corridor between warehouse 0 and warehouse 1. Then, it continues by entering warehouse 1, etc, until it finally exits the warehouse complex by exiting the last warehouse (warehouse 2 in the figure). Each robot has an specific weight, and the total weight of a robot and its cargo increases monotonously as it moves around in the warehouse complex.

In this example, there are two obvious variations on scheduling: to impose a FIFO policy in the advance of robots (this might reduce the overall throughput of the system), or to give preference to robots with the heaviest loads (with the risk of lighter robots starving). Assuming the existence of a function \( \text{older} \) on histories such that \( \text{older}(\text{pat}) \) returns the sub-history of pending calls matching \( \text{pat} \) with an arrival time older than the invoking one, the FIFO strategy can be expressed as:

\[
\begin{align*}
\text{PRE}: & n \in \{0 \ldots \text{N\_WAREHOUSES} - 1\} \\
\text{CPRE}: & w + \text{weight}(n) \leq \text{MAX\_WEIGHT} \\
\text{ENABLED}(\text{call, tag}): & \text{older}(\text{enterWarehouse}(n, \_)) = \emptyset \\
& \text{enterWarehouse}(n, w) \\
\text{POST}: & \ldots
\end{align*}
\]

whilst the largest robot first policy as:

\[
\begin{align*}
\text{PRE}: & n \in \{0 \ldots \text{N\_WAREHOUSES} - 1\} \\
\text{CPRE}: & w + \text{weight}(n) \leq \text{MAX\_WEIGHT} \\
\text{ENABLED}(\text{call, tag}): & \forall \text{enterWarehouse}(n, w') \in \text{waiting}(\text{enterWarehouse}(n, \_)) \cdot w' \leq w \\
& \text{enterWarehouse}(n, w) \\
\text{POST}: & \ldots
\end{align*}
\]

Another possible policy is to give priority to calls which correspond to robots which smaller weights, as they will move more easily through the warehouse complex. In terms of the shared resource definition, the FIFO scheduling policy is illustrated in Figure 2.12. where \( \text{waitings}(\text{pat}) \) contains the set of \textit{blocked} calls (input events) matching call pattern \( \text{pat} \) and \( \text{older}(\text{pat}) \) is the set of \textit{blocked} calls matching call pattern \( \text{pat} \) which are older than the \textit{present} one. The key idea of the presented methods is to give a minimal framework to reason about invocations and the order in which they were issued to set priorities (or even be used.

\(^{15}\) See Appendix A.9 for a full description and specification of the problem.
Chapter 2. Shared resources

CADT WarehouseAccessControl

... SEMANTICS
DOMAIN:

\[
\begin{align*}
\text{PRE: } & n \geq 0 \land n < \text{N\_WAREHOUSES} - 1 \\
\text{CPRE: } & w + \text{weight}(n) \leq \text{MAX\_WEIGHT} \\
\text{WAITING}(\text{call}): & \forall w \in 0..\text{MAX\_WEIGHT} \cdot \bigcup_{n=0}^{w} \text{waiting}(\text{enterWarehouse}(i, w)) \\
& \bigcup_{n=0}^{w} \text{waiting}(\text{exitWarehouse}(i, w)) = \emptyset \\
\text{enterWarehouse} & (n, w) \\
\text{POST: } & \text{weight} = \text{weight}^{\text{in}} + \{ n \mapsto \text{weight}^{\text{in}}(n) + w \} \land \\
& (n > 0 \Rightarrow \text{occupied} = \text{occupied}^{\text{in}} + \{ n \mapsto \text{false} \}) \land \\
& (n = 0 \Rightarrow \text{occupied} = \text{occupied}^{\text{in}}) \\
\text{exitWarehouse} & (n, w) \\
\text{POST: } & \text{weight} = \text{weight}^{\text{in}} + \{ n \mapsto \text{weight}^{\text{in}}(n) - w \} \land \\
& (n < \text{N\_WAREHOUSES} - 1 \Rightarrow \text{occupied} = \text{occupied}^{\text{in}} + \{ n + 1 \mapsto \text{true} \}) \land \\
& (n = \text{N\_WAREHOUSES} - 1 \Rightarrow \text{occupied} = \text{occupied}^{\text{in}})
\end{align*}
\]

Figure 2.12: Robots FIFO scheduling for the Warehouse example following the notation presented in [Her+09; Fre+16].

CADT SharedResourceName

... SEMANTICS
DOMAIN:

\[
\begin{align*}
\text{STATE: } & \text{self} = (\ldots \times \text{time} : \text{N} \times \text{calls} : \text{Set}(\text{call} \times \text{N})) \\
\text{INVARIANT: } & \ldots \land \text{time} \geq 0 \\
\text{INITIAL: } & \ldots \land \text{time} = 0 \land \text{calls} = \emptyset
\end{align*}
\]

\[
\begin{align*}
\text{PRE: } & \ldots \\
\text{CPRE: } & \ldots \\
\text{WAITING}(\text{call}): & \text{self\_time} = \text{self}^{\text{in}}.\text{time} + 1 \land \text{self\_calls} = \text{self}^{\text{in}}.\text{calls} \cup \{ (\text{call}, \text{time}) \} \\
\text{ENABLED}(\text{call}, \text{tag}): & \exists \text{time} \cdot ( (\text{call}, \text{time}) \in \text{self}^{\text{in}}.\text{calls} \land \\
& \forall (\text{call}', \text{time}') \in \text{self}^{\text{in}}.\text{calls}. \text{time} \leq \text{time}' \lor \neg \text{cpre}(\text{call})) \\
\text{operationName} \\
\text{POSTWAITING}(\text{call}, \text{tag}): & \text{self\_time} = \text{self}^{\text{in}}.\text{time} \land \text{self\_calls} = \{ (\text{call}', \text{time}') \in \text{self}^{\text{in}} : \text{call} \neq \text{call}' \}
\end{align*}
\]

Figure 2.13: General FIFO scheduling for a shared resources following the notation introduced in [Her+09; Fre+16].

as part of \text{CPRE} predicates). This is what we called \text{resource histories} which are a sequence of call events \((\text{op}_{\text{in}}, \text{op}_{\text{out}})\) having the possibility of referring them to the \text{past}, the \text{future}, or both. The machinery needed to implement histories, scheduling states and the helper functions is described in Section 6.1.

As can be intuited many scheduling policies are general, i.e., they do not depend at all on the particular operations available in a resource whereas other scheduling policies can depend on the parameters in calls to the resource. Using these scheduling primitives both general scheduling policies and policies specific to a particular example can be defined. We can for instance define a general FIFO policy for all operations of a resource, i.e., that the oldest call with a concurrency precondition true should always be scheduled first. This scheduling specification is presented in Figure 2.13

Note that the normal state of the shared resource is extended with the scheduling state. The
2.4 Networks of Shared Resources

A resource based system is composed of a set of shared resources, a set of processes, and a set of entities which forms the execution environment of resources and processes.

Processes (light code) are sequential programs that make normal sequential, blocking, calls to both shared resources and to environment entities. In this article we will not assume that processes are written in any specific programming language, as almost any language (with restrictions to forbid use of concurrent constructs) will do. Indeed, one of the advantage of the approach advocated in this thesis is that the troublesome aspects of concurrent programming are isolated in well-defined components (shared resources) of the resulting system, and does permeate the whole system design.

Processes execute in an environment, with which they interact. In the warehouse example, for example, the environment consists of the actual robots and the actual goods that are being carried around in the warehouse complex (appendix A.9). A process wanting to physically move a robot around, for instance, should first ask the warehouse shared resource for permission to do so, and once permission has been granted, it can issue orders to the environment (to the software of the physical robot) to effect the change. Moreover, the process must likely also interact with various sensors in the warehouse complex. For want of a better word we refer to the diverse set of entities (be they processes, threads, or hardware) which a process needs to interact with to solve its purpose, and which are not shared resources, as environment entities.

In this work the model of such process-to-environment interactions is made by normal operation calls as well, such that a call from a process to an environment entity blocks until a reply is received. Should a more asynchronous API be needed, i.e., to model sensor interrupts from the environment to a process, this has to be programmed using multiple processes and/or shared resources.

2.4.1 Sorting in parallel: MergeSort using shared resources

As an example of how multiple shared resources can be composed, a sorting parallel algorithm is presented. This is to provide a new useful and more complex control behaviour of simple shared resources.

The MergeSort algorithm ‘divide-and-conquer’ technique to sort a collection of elements (of any kind of comparable nature). This algorithm divides the n-element sequence to be sorted into two subsequences of \( n/2 \) elements each. Then, it sorts the two subsequences recursively using merge sort. After that, the merge algorithm combines the two sorted subsequences to produce the sorted answer. The recursion ‘bottoms out’ when the sequence to be sorted has length 1, in which case there is no work to be done, since every sequence of length 1 is already in sorted order.

A shared resource MergeSortComp which implements the merge task of a mergesort algorithm is defined in Figure 2.14.

The behaviour of this shared resource is rather straightforward. Its state is represented by two items of ANY kind\(^{16}\), left and right, and a number of operations are defined to manipulate those attributes: SetLeft (and analogously SetRight) changes the value of the left (right) state member on the condition that no value is waiting to be merged. Initially, both variables are equal to nil. The third operation, GetResult, returns the minimal value among left and right. Notice that there is a special value EOD (short for End-Of-Data) which indicates that the process calling SetLeft (or SetRight) has run out of values to merge. The EOD value is greater than any value in the ANY type. If both left and right are equal to EOD the shared resource forwards the end of data indication via the GetResult operation. Note the subtle interplay between RETURN and POST clauses in the definition of GetResult: the RETURN clause chooses the value to return (the minimum

\(^{16}\) Notice that in fact, the type has an spare value that indicates that the last element has been read, i.e. a termination special token.
Chapter 2. Shared resources

CADT MergeSortComp

OPERATIONS
ACTION SetLeft: ANY − {nil}
ACTION SetRight: ANY − {nil}
ACTION GetResult: → ANY ∪ {EOD}

SEMANTICS
DOMAIN:
STATE: self = self = (right : ANY ∪ {EOD} × left : ANY ∪ {EOD})

INITIAL: self = (nil, nil)

CPRE: self.left = nil
SetLeft(obj)
POST: self = (obj, self^next.right)

CPRE: self.right = nil
SetRight(obj)
POST: self = (self^next.left, obj)

CPRE: self.right ≠ nil ∧ self.left ≠ nil
getResult(r)
POST: (r = EOD ⇒ self^next = self = (nil, nil)) ∧
(r = self^next.left ⇒ self = (nil, self^next.right)) ∧
(r = self^next.right ⇒ self = (self^next.left, nil))
RETURN: min(self^next.left, self^next.right)

Figure 2.14: MergeSortComp specification according to the notation introduced in [Her+09].

of the left and right state components) while POST uses the return value (the minimum) to update the state.

Note that for the MergeSortComp shared resource to work correctly incoming data using the SetLeft and SetRight operations needs to be sorted. Moreover, after merging two input sources, the shared resource is able to begin a new phase of merging the new two input sources, as long as every phase is correctly terminated with an end-of-data indication EOD (special token of ANY type).

In this example, three cases can be distinguished depending on the state of the resource: (i) when both values left and right are EOD, the operation (returns EOD); (ii) if one of the attributes (say left) is equal to EOD but the (other) right does not, the returned value is the content of right analogously to left; (iii) if none of the values is EOD, the operation will return the smallest of the two values stored for this reason, all the items must be comparable with each other.

The MergeSortComp shared resource is capable of merging elements from two sources (SetLeft and SetRight), but it cannot by itself merge n input sources, where n > 2. However, by composing multiple instances of the MergeSortComp shared resource which are connected by processes, a system capable of merging n input sources can be constructed.

In Figure 2.15 a composed system is presented. It consists of three shared resources MSC₁, MSC₂, and MSC₃, and two interconnecting processes E and F, which can merge four input sources. It is assumed that the process E repeatedly calls GetResult on MSC₁ to obtain a value v, and then calls SetLeft on MSC₃ with the value as a parameter. The F process behaves analogously.

As can be seen, the base cases (in which a process reads from a stream and then passes the value to the shared resources) are similar to the base cases of the well-known algorithm itself.

First, processes A and B (similarly C and D) read the first two values from the array and then pass them to the shared resource MSC₁ using SetLeft and SetRight operations, respectively. At the same time, Processes E, F and G call GetResult method from
MSC₁, MSC₂, and MSC₃, and get blocked. These invocations leave the system as follows: calls from threads A to D are allowed and executed due to the fact that all shared resources are empty (MSC₁ and MSC₂ contain both values), but the invocations of processes E and F are blocked until the other threads finish their execution, i.e., the other shared resources must contain two values. Process G is also blocked until MSC₃ is filled in with data.

After that, CPRE of GetResult operations (from MSC₁ and MSC₂) holds, so, E and F are unblocked and receive values 2 and 3. Those values are passed then to MSC₃ shared resource, so threads E and F are enabled to invoke GetResult operation again (they are blocked until processes B and C put EOD value to the shared resource). This scenario enables process G to get the minimal value (2) of the given array (CPRE of GetResult of MSC₃ holds). The system continues working as explained until thread G get one by one all elements sorted. Notice, as specified, that any value is always smaller than EOD value.

This example illustrates how even a really simple shared resource can be part of a complex whole solution. Considering MergeSortComp, it can be verified and tested easily due to its responsibilities and its behaviour. However, if we consider the whole system, the whole system and environment must be taken into account in order to perform a correct verification and validation of the system.

Consider now the merge sort solution using shared resources which implements a four-way merge operation. To ensure the correctness of the operations, all involved processes in the figure must respect the requirement that data sent to a shared resource (e.g., as the argument to a call to SetRight or SetLeft) must be sorted (up to the reception of an end-of-data indication). Moreover processes E and F make strictly alternating calls between retrieving a new value (via calls to GetResult and forwarding the received value via calls to SetLeft or SetRight).
From shared resources to TLA+ specifications: model validation

Let us dismiss the question, “Have you proven that your model is valid?” with a quick NO. Then let us take up the more rewarding and far more challenging question: “Have you proven that your model is useful for learning more…”
—James B. Mankin, Jr.

Once the model for the shared resource is obtained, a systematic translation to TLA+ [Lam15; EGL92] is presented. It includes a combination of a linear-time temporal logic (The Temporal Logic of Actions [Lam94]) and Zermelo-Fränkel set theory. Firstly, a linearization semantics is presented in an informal way, introducing some general information about the translation.

### 3.1 Structuring the shared resource translation

TLA+ specifications are organized into modules that can be extended into other modules to use their functionality. Although the TLA+ standard is specified in typeset mathematical symbols, existing TLA+ tools use LaTeX-like symbol definitions in ASCII to facilitate the translation.

Briefly, in TLA+ a ‘state’ is an assignment of values to variables and a ‘behaviour’ is a sequence of states. Thus, a ‘step’ is defined as a pair of successive states in a behaviour and the ‘next-state’ relation is a relation describing how variables can change in any step from one step to the other.

Roughly speaking, each shared resource is translated into a TLA module. It contains a TLA+ specification, i.e a formula $S$ written as a conjunction of a TLA predicate $I$ that states the initial value of TLA variables $x, y, \ldots$ representing the state of the system and a next-state relation $N$ (TLA action) that specifies valid value changes of the variables: $S \triangleq I \land \Box[N](x, y, \ldots)$. For now on, the translation is based on the air traffic controller example (see Figure 2.8). The signature of the module is defined as follow

```
---------------------------- MODULE AirTrafficController -----------------------

EXTENDS Naturals, Sequences

The name of the module must be the same as the filename. Then, all modules needed are included in order to use, for instance, mathematical operations. EXTENDS incorporates the declarations, definitions, assumptions, and theorems from the modules listed into the current module. If any constant is needed, they must be defined using the keyword CONSTANTS. This must be follow by a set of constraints or properties that those constants must satisfy (mainly type information).
```
CONSTANTS
  MAX_RUNWAYS_ID,

ASSUME
  \( \langle /\ \text{MAX\_RUNWAYS\_ID} \in \text{Nat} /\ \text{MAX\_RUNWAYS\_ID} > 0 \rangle \)

Following the example, it is mandatory to described that the last identifier for a runway must be a natural number and it must be grater that zero, i.e. there is at least one runway in the airport. Constants can be seen as the parameters of the problem. Each constant \( C_j \) is declared to be constant parameters (rigid variables): it is either an identifier or has the form \( C(\_ , \ldots , \_ ) \), the latter form indicating that \( C \) is an operator with the indicated number of arguments.

3.1.1 Describing the inner state of the resource

Each component of the domain of the specification in the presented notation is translated into a TLA variable. Those depict the resource domain.

VARIABLES
  runways

3.1.2 Types and predicates

Since TLA has almost no predefined types\(^1\) are translated into sets and predicates (initial state predicate, invariant, preconditions, and concurrency preconditions) into TLA predicates. In this case, a function from natural numbers to boolean is defined in order determine if a specified runway is being used or not.

\[
\text{Types} == \text{runways} \in [\text{Runways\_Id} \to \text{BOOLEAN}]
\]

Once, types are defined, a TLA predicate that represents the initial state is depicted. A state satisfying this predicate and the invariant predicate, is the starting point for TLC. Invariant and Init predicate for the air traffic controller can be seen below

\[
\text{(* Resource invariant *)} \\
\text{Invariant} == \text{TRUE}
\]

\[
\text{(* Initial state of the resource *)} \\
\text{Init} == \langle /\ A \ i \in 1 .. \text{Len (runways)}-1 : \text{runways}[i] = \text{FALSE} \rangle
\]

3.1.3 Operations as set of TLA Actions

Every operation is translated into a TLA action which is the conjunction of the predicates collecting type information, precondition, concurrency precondition, and postcondition of the operation. This action will represent an atomic, valid transition of the system.

Consider the operation \( \text{afterLanding} (r) \) in which an airplane set free the runway \( r \). Let us first define the type of the parameter \( r \) as follow

\[
\text{TYPE\_After\_Landing} \ (r) == r \in \text{Runway\_Id}
\]

which describes that \( r \) must be a valid runway id. Then, other two predicates are needed: one describing the precondition, and another to describe the concurrent precondition. Thus, the following definition is obtained:

\[
\text{PRE\_After\_Landing} \ (r) == \text{landing}[r] = \text{TRUE}
\]

\[
\text{CPRE\_After\_Landing} \ (r) == \text{TRUE}
\]

\(^1\)Notice that even natural numbers are defined as an external module.
3.2 Validation of the model

However, post-conditions cannot be depicted with a simple predicate because it expresses a change variable values. That’s why postconditions are translated into TLA actions\(^2\). The value of a variable before an action is represented using the variable name, and its value after the action is represented with the same name primed; so we replace \(x\) by \(x\) \(x\) and \(x\) by \(x\).

\[
\text{POST\_After\_Landing} (r) == \\ Slash\ a\ \text{runways}[r]' = \text{FALSE} \\ Slash\ \forall \ a \ i \ in \ 1 .. \text{Len (runways)}-1 : \text{runways'}[i] = \text{runways}[i]
\]

Combining all together, a full description of a shared resource operation is as follow:

\[
\text{After\_Landing} (r) == \\ Slash\ \text{TYPE\_After\_Landing} (r) \\ Slash\ \text{PRE\_After\_Landing} (r) \\ Slash\ \text{CPRE\_After\_Landing} (r) \\ Slash\ \text{POST\_After\_Landing} (r) \\ Slash\ \text{PrintOp} (\text{"Ask\_Landing",} r>)
\]

3.1.4 A shared resource becomes a dynamic system

The TLA formula that gives semantics to the resource specification as a dynamic system is given by the definition of the \text{next-step} relation as the disjunction of all actions resulting from the translation of operations. Informally, every transition (step) the resource may experience is triggered by the execution of some of its operations. We are deliberately ignoring the restrictions that the processes impose on the possible operation interleavings.

\[
\text{Next} == \\ Slash\ \text{Ask\_For\_Landing} \\ Slash\ \exists r \in \text{Len (runways)}-1 : \text{After\_Landing} (r) \\ Slash\ \text{Ask\_For\_Take\_Off} \\ Slash\ \exists r \in \text{Len (runways)}-1 : \text{After\_Take\_Off} (r)
\]

The formula \text{Spec} specifies that the system starts in a valid state and every transition it takes is one of these defined by the \text{Next} formula, maybe leaving variables unchanged (not in this example), is then

\[
\text{Spec} == \text{Init} \Slash:\ [] [\text{Next}] _<<\text{runways}}>>
\]

3.2 Validation of the model

In this section it is shown how some execution properties of an example can be studied thanks to the translation of shared resources into TLA and, eventually, the use of the TLC model checker. Being able to use such a tool does not guarantee, in general, the correctness of the system, but it helps to find possible inconsistencies or gaps in the specifications.

Some of the properties to check are generic (i.e., the invariant always holds) and some of them depend on the system at hand. We will use in fact two variants of the specification. The first one is what we described in Section 3.1, which leaves complete freedom to the interleaving of the operations, and is adequate to verify safety properties which are connected with resource reuse. The second one includes the necessary machinery to enact interleaving constraints which model the behaviour of the process.

3.2.1 Internal verification

Without considering the context in which the specified resource will be used, only the integrity of the invariant and type information of variables can be checked.

\(^2\)A TLA action is a predicate which relates input and output states
CONSTANTS
MAX_RUNWAYS_ID = 2

SPECIFICATION Spec
INVARIANT Types
INVARIANT Invariant
INVARIANT NewInvariant

Figure 3.1: TLC configuration definition of the system to be checked.

Checking the invariant

The invariant has been translated into the following TLA specification in the following two predicates:

\[
\text{Types} \equiv \text{runways in } \{\text{Runways_Id} \to \text{BOOLEAN}\}
\]

\[
\text{Invariant} \equiv \text{TRUE}
\]

In this case, the TLC checker goes ahead without showing any error or violation. However, considering changing the invariant in order to always leave a free runway (for emergencies), getting the following invariant:

\[
\text{NewInvariant} \equiv \exists \text{id } \in \text{Len (runways)-1 : runways[id]}
\]

After updating the file and passing it to the model checker TLC, it is mandatory to define the values for constants and the properties (invariants in this case) to be checked. The model checker then, found a couple of violations of the invariant. Both are related with the restriction imposed by the new invariant.

Error: Invariant NewInvariant is violated. The behavior up to this point is:
STATE 1: <Initial predicate>
\ /
state = [FALSE, FALSE]
STATE 2: <Action Ask_For_Landing>
\ /
state = [TRUE, FALSE]
STATE 3: <Action Ask_For_Landing>
\ /
state = [TRUE, TRUE]

These violations is because an airplane is always allowed to use a runway. For that invariant, the specification of the PRE of Ask_For_Landing and Ask_For_Take_Off operations must be changed to

\[
\exists \text{id } \in \text{Len (runways)-1 : } \exists \text{id_2 } \in \text{Len (runways)-1 : } ! \text{runways[id]} \land ! \text{runways[ id_2 ]}
\]

After this, it seems that the resource specification has reached a better degree of integrity. The model checker did not find more errors during the checking of the specified resource invariant. Note that, since we did not impose any restriction to the interleavings of the operations due to the way processes are defined, the properties checked here will be valid in any context of the resource, guaranteeing the reusability of the shared resource.

3.2.2 Adding processes to complete the system

TBD.
Once the model has been validated and optimized (if possible), the next step in the presented methodology is the generation of code based upon a set of templates. As it has been said before, the “heavy code” is distilled in a semi-automatic manner following schemata for code generation. Thus, the programmer still retains the power of modifying part of the code for the resource but the traceability of the code is kept. Such templates specify the code refactoring needed to implement a shared resource, as well as, the crucial points in which the proof obligations have to be placed. The following sections introduce a brief explanation of JML annotation language and, then, present an approach for translating a shared resource specification into a piece of code using a set of code templates presented in [Mar+19].

A suitable code template must be chosen in a two-step process selection. First, a particular concurrency mechanism is selected to determine the critical regions, e.g., using Java synchronized methods (mutual exclusion), priority monitors (mutual exclusion), or using the JCSP message-passing library (selective request processing).

The choice of which concurrency mechanism to use is largely a matter of taste. Although it should be noted that the single code template we present for Java synchronized methods cannot be recommended in general, as it does not guarantee important (process) liveness properties. Second, having chosen a concurrency mechanism, a template user should analyze the concurrency preconditions (CPRE) and post conditions – e.g., to find out if the concurrency precondition depends on both the operation parameters and the resource attributes – i.e. its internal state. Next, a particular code template is selected which is compatible with the analysis result.

First, an extension of JML will be presented to express that a Java interface is, in fact, a shared resource (Section 4.1).

Then, each code template is defined and exemplified in the following sections: Section 4.2 presents a single code template for implementing a shared resource using Java synchronized methods, while Section 4.3 presents a number of code templates for implementation using priority monitors. Finally, Section 4.4 contains code templates for implementing resources using the JCSP Java CSP messaging library. These code templates are originated from the classroom use of the model-driven approach in several undergraduate courses on concurrency [Car+04; CHM13].
4.1 Extending JML for shared resources

Although the mathematical notation has the virtue of being fully independent from the programming language, it is usually practical to have a way of dealing with language-dependent issues that will appear during the code generation phases such as constants definitions, modules, initialization, libraries, parameters, to name a few. Considering the Java programming language, this process begins with transforming the specification into a Java interface, i.e. defining their interaction with the outside world through the methods that they expose but without specifying any concurrent mechanism at all. Moreover, the restrictions specified in the mathematical notation must be translated as well in order to express the behaviour depicted in the specification. For that reason, the well-known and widely used Java Modeling Language (JML) was chosen.

JML is a behavioural interface specification language for Java modules using Hoare style pre- and postconditions and invariants, that follows the design by contract paradigm. JML provides semantics to formally describe the behavior of a Java module, preventing ambiguity with respect to the module designers’ intentions. It combines the design by contract approach of Eiffel [NS87] and the model-based specification approach of the Larch family [GHHW85] of interface specification languages, with some elements of the refinement calculus. Due to the fact that JML specifications can be written as annotations in Java program files, or stored in separate specification files, Java modules with JML specifications can be compiled unchanged with any Java compiler. Other options such as OCL (Object Constraint Language) were discarded due to the widely acceptance and support offered by several tools (including KeY Software) that cover different levels of verification such as static checking, interactive verification, runtime checking, among others.

As annotation language JML provides several constructs to specify the aspects of thread safety explained before [Rod+05]: e.g., the thread_safe predicate specifies that a given object is thread-safe, i.e. it is either local to a thread or access to it is protected by a lock. JML also provides a number of constructs to explicitly deal with locks. The lockset expression refers to the set of locks held by the current thread. Although described in [Rod+05], these constructs were not implemented in the JML compiler, thus not producing any runtime assertion checking code. Note that stating thread safety using thread_safe clause is equivalent to stating it explicitly through locking requirements. The choice is based on ease of specification only.

However, those are either strictly related to the lock and conditions mechanisms or do not offer any way to described when the invoker must be blocked. Therefore, in [MA15] an extension (a set of clause) of JML was presented. Conceptually, this agrees with the view that Java interfaces are meant to express a contract for potential implementations. In the case of a shared resources, this means that different classes can implement the same resources, e.g. using disparate synchronization models, such as shared memory or message passing. Additional advantages of this approach are that the logic annotations sit reasonably well alongside JML and make the specified contract more concise and readable without adding any extra time and effort to learn how the specification should be written. Indeed, only two basic clauses had to be added to the original JML syntax:

- **shared_resource** declares that all instances of the class represent a shared resource, guaranteeing that a transactional semantics of interleavings of methods call holds;
- **cond_sync** declares that any invoker will be blocked when the associate predicate does not hold and should be in this state until the condition become true by the changing of the shred resources inner state. In other words, this correspond to the CPRE clauses in the mathematical notation.

The rest of clauses in the shared resource specification – PRE, POST, INVARIANT, INITIAL, etc. – can be cast into standard JML declarations (requires, ensures, invariant and initially, respectively) which means that our extension should not be very disruptive for JML practitioners. Other works, such as [Rod+05; Ara+11; ABL08], present extensions of

---

1There are several patterns or tools for translating from other specification languages to JML specification sentences. An example of them is presented in [Ham15], so OCL specifications can easily be translated into JML.
4.1. Extending JML for shared resources

```java
public interface /*@ shared_resource @*/
{
    // Constants definitions
    @/ public model instance
    T CONSTANT1;
    @/ public model instance
    T CONSTANT2;
    // Constants definitions
    @/ public model instance
    StateVariableType stateVariable1;
    @/ public model instance
    StateVariableType stateVariable2;
    // Public instance invariant
    Pred2JML(INVARIANT);
    // Public initially
    Pred2JML(INITIAL);
    /*@ public normal behaviour
    @ requires
    Pred2JML(PRE(op1(a1,...,an)));
    @ cond_sync
    Pred2JML(CPRE(op1(a1,...,an)));
    @ assignable
    Pred2JML(free(POST(op1(a1,...,an))));
    @ ensures
    Pred2JML(POST(op1(a1,...,an)));
    */
    public void T N op1(T1 arg1,..., TM argM);
}
```

Figure 4.1: Java generic Interface template with JML annotations for a given shared resources.

JML or other notations but, unlike our proposal, they rely on a specific concurrent mechanism.

In order to make the translation more systematic, we present the resulting interface in general way in Figure 4.1. This template, as well as the ones presented in the following sections, is a combination of immutable parts, and placeholders (marked as gray boxes) for fragments of JML expressions or Java code derived from the specification of the shared resource being implemented. Currently these fragments must be filled in manually by the user – an item for future work is to enable such code fragments to be derived automatically from a shared resource specification.

Given an operation `op` and its associated mathematical predicates `PRE`, `CPRE` and `POST` representing the pre-, concurrent pre- and post-condition as shown in the shared resource specification (for instance, Figures 2.8 and A.21 for tower controller and the readers & writers problems, respectively), we define

\[ \text{Pred2JML : predicate} \rightarrow \text{JML expression} \]

which maps every mathematical predicate of the specification into a JML expression; and

\[ \text{free : predicate} \rightarrow \text{Set\{variables\}} \]

which maps every predicate into a set of identifiers (variables that define the shared resource state) that occur free in the predicate.

For instance, let us consider `afterLanding` operation of the air traffic controller problem specified in Figure 2.8. In this case, the JML boolean expression obtained from the precondition is shown below:

\[ \text{Pred2JML}(\text{PRE}_{\text{afterLanding}}(r)) = \text{true} \]

because it is considered that the caller follows the protocol established by the shared resource. If not, it is mandatory to check whether `r` is between bounds.
However, obtaining the JML expression associated to the postcondition is a little bit trickier:

\[
\text{Pred2JML(\text{POST}_{\text{afterLanding}(r)}) = \{\forall i : i \geq 0 \& \& i < \text{runways.length} \& \& i \neq \text{result}; \}
\
\text{runways}[i] = \text{old}(\text{runways}[i])};
\]

Notice that the \textit{old} operator is used to express that the given runway is free again whilst the others remain with the same value. This quantifier can be avoided by using the JML clause \texttt{@ assignable runways[r]}, i.e, the only assignable position of the array is the one passed as argument. From that, free variables in the postcondition can be deduced as follows:

\[
\text{free(\text{POST}_{\text{enterWarehouse}(n,w)}) = runways[r]}
\]

After defining those functions, the JML-annotated Java interface obtained can be found in Figure 4.2. Once the behaviour of the shared resource is formally defined in our target programming language, the user can proceed to the code generation phase.

### 4.2 Model-based Code Generation Using Synchronized Methods

Java programming language provides two basic synchronization idioms than can be natively used. \textit{Synchronized methods} and \textit{synchronized statements} are built around an internal entity known as the intrinsic lock or monitor lock. Intrinsic locks play a role in both aspects of synchronization: enforcing exclusive access to an object’s state and establishing happens-before relationships that are essential to visibility.

Every object defined as a Java class contains an intrinsic lock associated with it. This synchronization mechanism uses the object’s native lock to prevent two or more threads accessing the code simultaneously in these methods marked with the \texttt{synchronized} modifier. A thread is said to own the intrinsic lock between the time in which the lock has been acquired and released. As long as a thread owns an intrinsic lock, no other thread can acquire the same lock. The other thread will block when it attempts to acquire the lock.

This guarantees the following: first, it is not possible that two different invocations of synchronized methods interleave, i.e, when one thread is executing a method, all other threads that invoke any synchronized methods of the same object block until the first thread finishes. Second, when a synchronized method exits, it automatically establishes a happens-before relationship with any subsequent invocation of a synchronized method for the same object. This guarantees that changes in the state of the object are visible to all threads\textsuperscript{2}.

Thus, the first template introduced takes advantage of Java’s \texttt{synchronized} methods to guarantee mutual exclusion. Synchronized methods enable a simple strategy for preventing thread interference and memory consistency errors: if an object is visible to more than one thread, all reads or writes to that object’s variables must be done through synchronized methods.

In addition, a very basic mechanism for condition synchronization is provided. A thread inside a synchronized method can be suspended by invoking \texttt{wait()}, which also releases the mutual exclusion lock, allowing another thread to enter the code in some synchronized method of that object. On the other hand, threads that modify the state of the shared object may invoke \texttt{notifyAll()}, which gives the suspended threads a chance to resume execution (when the mutual exclusion lock allows).

The generic template for translating a given operation \texttt{op} of the given shared resource, with parameters \texttt{arg1}, ..., \texttt{argM}, to Java code using synchronized methods is presented in Figure 4.3.

\textsuperscript{2}Notice that constructors cannot used the \texttt{synchronized} modifier.
4.2. Model-based Code Generation Using Synchronized Methods

```java
public interface /*@ shared_resource */ AirTrafficController {

    //@ public instance model non_null boolean[] runways;
    //@ public instance invariant runways.length <= runways.length;
    //@ public initially (\forall int i ; i >=0 && i<runways.length; !runways[i]);

    //@ public normal Behaviour
    @ requires true;
    @ cond_sync (\exists int i ; i >=0 && i < runways.length ; !runways[i]);
    @ assignable runways[*];
    @ ensures \result < runways.length && \result >=0 && runways[\result] &&
    (\forall int i; i>=0 && i< runways.length && i != \result;
    runways[i] == \old(runways[i]));

    public int beforeLanding();

    //@ public normal Behaviour
    @ requires runways[r] && r >= 0 && r < runways.length;
    @ cond_sync true;
    @ assignable runways;
    @ ensures !runways[r];

    public void afterLanding( int r);

    //@ public normal Behaviour
    @ requires true;
    @ cond_sync (\exists int i ; i >=0 && i < runways.length ; !runways[i]);
    @ assignable runways[*];
    @ ensures \result < runways.length && \result >=0 && runways[\result] &&
    (\forall int i; i>=0 && i< runways.length && i != \result;
    runways[i] == \old(runways[i]));

    public int beforeTakingOff();

    //@ public normal Behaviour
    @ requires runways[r] && r >= 0 && r < runways.length;
    @ cond_sync true;
    @ assignable runways[r];
    @ ensures !runways[r];

    public void afterTakingOff( int r);
}
```

**Figure 4.2:** The Air tower controller example specified in JML using the defined extension.

This template, as well as the ones presented in the following sections, is a combination of immutable parts, and placeholders (marked as gray boxes) for fragments of Java code derived from the specification of the shared resource being implemented. Currently these Java fragments must be filled in manually by the user – an item for future work is to enable such code fragments to be derived automatically from a shared resource specification.

Considering a given operation \( op \) and its associated mathematical predicates \( \text{PRE}, \text{CPRE} \) and \( \text{POST} \) as it is shown in a shared resource specification, the following total functions are defined:

\[
\text{Pred2Java} : \text{predicate} \rightarrow \text{Java boolean block}
\]

which takes a mathematical predicate from the shared resource specification and converts it into a Java boolean expression\(^3\).

\(^3\)For simplicity, we are assuming that the Java code for checking a boolean condition can be represented by a call to an auxiliary method, which would be part of the outcome of \text{Pred2Java}. Of course, actual code may take a different, but equivalent, form.
public synchronized $T_N \text{ op}_1(T_{arg_1}, \ldots, T_{arg_M})$ throws $←$

    PreViolationResourceException;

    if (!Pred$_{\text{Java}}$(PRE$_{\text{op}_1}(a_1, \ldots, a_n))){
        throw new PreViolationResourceException();
    }

    //@ assume $\text{Pred$_{\text{JML}}$(PRE$_{\text{op}_1}(a_1, \ldots, a_n))}$;
    $T_N$ result = null;

    while (!Pred$_{\text{Java}}$(CPRE$_{\text{op}_1}(a_1, \ldots, a_n))){
        wait();
    }

    //@ assert $\text{Pred$_{\text{JML}}$(PRE$_{\text{op}_1}(a_1, \ldots, a_n))}$
    $\text{\&\& invariant\_for\_this}$ $\text{\&\&}$$\text{Pred$_{\text{JML}}$(CPRE$_{\text{op}_1}(a_1, \ldots, a_n))}$;

    Post$_{\text{Java}}$(POST$_{\text{op}_1}(a_1, \ldots, a_n))$;

    //@ assert $\text{Pred$_{\text{JML}}$(POST$_{\text{op}_1}(a_1, \ldots, a_n))}$
    $\text{\&\& invariant\_for\_this}$;

    notifyAll();

    return result;

Figure 4.3: Generic template for implementing an operation of a shared resource using synchronized methods.

Post$_{\text{Java}}$: predicate $\rightarrow$ Java block

converts a post-condition predicate into a block of code that establishes the postcondition.

For instance, let us consider again the afterLanding operation of the Airport tower controller problem specified in Figure 2.8. In this case, Pred$_{\text{Java}}$ and Pred$_{\text{JML}}$ return the same expression for both PRE and CPRE. The Java boolean expression for the precondition is shown below:

$$\text{Pred$_{\text{Java}}$(PRE$_{\text{afterLanding}}(r))} = \text{true}$$

Similarly, a Java code fragment that encodes the POST predicate is shown below:

$$\text{Post$_{\text{Java}}$(POST$_{\text{afterLanding}}(r))} = \{ \text{runways}[r] \text{ = false; } \}$$

The template first checks the precondition (PRE$_{\text{op}}$) of the given operation, and raises an instance of the exception PreViolationResourceException if not satisfied. The arguments $a_1, \ldots, a_n$ to PRE$_{\text{op}}$ correspond to the (some) operation arguments ($arg_1, \ldots, arg_{M}$) and the resource attributes. After the precondition has been checked, a JML assume annotation is inserted on the premise that the precondition holds.

Notice that JML assume statements are those expressions on which the code can ‘trust’ to hold. In the case of the PRE, it is assumed that the expression is valid before continuing the execution. However, assert expressions must be checked and if some of those expressions do not hold, the code must halt or warn the user that the system is in an inconsistent state. In the case of the CPRE, the expression must hold before continuing executing the critical section of the operation indicating that the invariant holds and the access to the shared resource inner state variables is allowed.

Next the concurrency precondition (CPRE$_{\text{op}}$) is repeatedly checked, until it holds. A call to wait is inserted to permit the calling thread to sleep until the resource state changes. The next line in the code templates provides a JML annotation which states that the both the precondition, the concurrent precondition and the resource invariant are asserted.
Next, the code obtained from Post2Java is executed, i.e., the resource attributes are updated, and the operation result is also computed. In the next line a JML annotation asserts the post condition (POST\textsubscript{op}) and the invariant of the shared resource. Finally, the call to \texttt{notifyAll()} allows another waiting operation to proceed, and the operation result is returned.

The result of applying this code template to all the operations of a shared resource is a set of Java methods, which should be embedded in a single Java class containing the attributes of the resource represented as Java attributes\(^4\).

Note that the process of applying a code template to an object method to obtain a Java method is still carried out manually. As future work, we intend to implement code templates as code refactoring activities in a Java integrated development environment.

However, although the solution presented is valid in the sense that the safety conditions are met, repeatedly checking the synchronization condition in a loop makes it quite inefficient in the presence of a large number of threads. Moreover, the nondeterminism inherent to \texttt{notifyAll()} makes it impossible to put an upper bound on the number of times threads execute the condition synchronization loops at the entry of \texttt{put} and \texttt{get} – i.e. liveness properties such as \textit{bounded wait} cannot be ensured with this scheme.

### 4.2.1 Airport Tower Controller implementation

The result of applying this template to the Airport problem (Figure 2.8), the code showed in Figure 4.4 is obtained\(^5\). The code is simplified and only shows two operations due to the fact that the other two are analogous to these ones.

In this example, several helper methods are defined in order to make the code cleaner. These are the \textit{precondition} of the \textit{after} operations and the \textit{CPRE} for \textit{asking} operations. Notice that both of them are annotated with the \texttt{pure} modifier expressing that all class attributes remain unchanged after the execution.

Let us consider the \texttt{askForLanding} operation of this resource. In this case, all invocations are valid due to the fact that its \textit{precondition} always holds (line 26). However, its \texttt{CPRE} tells that an invocation can proceed if and only if there is an available runway (lines 9-15). If all runways are in used, the caller is blocked and it must wait until another process wakes it up. Immediately after being awakened, the \texttt{CPRE} is checked again in order to find out if there are any available runway. When there exists a free runway, its \texttt{CPRE} becomes true and the invocation can proceed (either the caller was blocked and awakened - line 28 - or there was a free runway when the \texttt{CPRE} was first checked - line 27). After that, the code asserts that the conjunction of the \texttt{CPRE}, \texttt{PRE} and the class invariant must hold denoting that the execution was valid and it can proceed to execute the critical region of the method in a inner valid state.

Thus, the air traffic controller must find which was the available runway and mark it as in use (lines 31-38) fulfilling the \texttt{POST} of the method.

Then, \texttt{notifyAll()} is called waking up all the threads waiting on the object’s lock without discrimination. As can be seen, this invocation has not significant effect regarding the liveness of the system. The reason why is because all blocked threads are waiting for a runway to be release and in this method do the opposite. So, each awaken process is put back to sleep immediately after it was awaken.

Considering now the operation \texttt{afterLanding} of Figure 4.4, the \textit{precondition} (\texttt{PRE}) must hold at call time (line 50). If not, an exception is thrown telling the invoker that the call cannot proceed and it is aborted. Then, immediately before performing the modifications in the resource, it is mandatory to check if its (\texttt{CPRE} - line 54) holds. In this case, it cannot be blocked because its \texttt{CPRE} is always true, that is why we immediately assert that the conjunction of the \texttt{PRE}, \texttt{CPRE} and the class invariant holds.

Thus, the code proceed to execute the core statement (line 58) by updating the resource in order to reflect that the given runway is now available again. This code must ensures that the \textit{postcondition} holds within the invariant, leaving the shared resource in a consistent state (as is expressed in line 60).

\(^4\)In order to achieve the encapsulation principle, any other helper method should be declared private.

\(^5\)The Java interface of the airport example is presented in the Figure B.2.
public class AirTrafficControllerSync extends AirTrafficControllerSync {
    //@ public represents runs <- runways;
    protected volatile /*@ spec_public @*/ boolean runways[];
    //@ ensures ra == (\exists int i; i >= 0 && i < runways.length; runways[i]);
    private /*@ pure */ boolean cpresAsking(){
        for (int i = 0; i < runways.length; i++) {
            if (!runways[i])
                return true;
        }
        return false;
    }
    private /*@ private normal_behavior @*/
    @ requires r >=0 && r < runways.length;
    @ ensures runways[r];
    @*/
    private /*@ pure */ boolean preAfter(int r){
        return runways[r] && r >=0 && r < runways.length;
    }
    public synchronized int askForLanding() {
        //@ assume true;
        while (!cpresAsking()) this.wait();
        //@ assert true && true && cpresAsking();
        int ra = 0;
        for (int i = 0; i < runways.length; i++) {
            if (!runways[i]){
                ra = i;
                runways[ra] = true;
                break;
            }
        }
        //@ assert ra < runways.length && ra >=0 && runways[ra] &&
        @ (\forall int i; i>=0 && i< runways.length && i != ra;
        @ runways[i] == \old(runways[i]));
        notifyAll();
        return ra;
    }
    public synchronized void afterLanding(int r) throws PreViolationSharedResourceException {
        if (!preAfter(r))
            throw new PreViolationSharedResourceException("afterLanding");
        //@ assume preAfter(r);
        while (!true) this.wait();
        //@ assert !preAfter(r) && true && true;
        runways[r] = false;
        //@ assert !runways[r];
        notifyAll();
    }
    public synchronized int askForTakingOff() {
        ...analogous to askForLanding operation...
    }
    public synchronized void afterTakingOff(int r)
        throws PreViolationSharedResourceException {
        ...analogous to afterLanding operation...
    }
}

FIGURE 4.4: Airport Traffic Controller implemented using synchronized methods from the interface in Figure B.2.2.
Finally, the method calls `notifyAll` in order to wake up all threads that are waiting on this object’s lock. In this case, the awakening is effective because if a process is blocked, that means that it is waiting for a free runway.

### 4.3 Model-based Code Generation Using Priority Monitors

The problems with the `wait()`/`notifyAll()` mechanism are caused by its inability to force the resumption of threads whose synchronization condition is known to hold. Monitors [Bri72; Hoa74] were designed to allow for a finer control of thread blocking and resumption. Although Java includes its own realization of the monitor concept in the `locks & conditions` library, we will present a quite efficient implementation scheme based on our own implementation of `priority` monitors [HM12].

Our monitor library provides a class named `Monitor` of monitor objects which, in first place, can be used to enforce mutual exclusion on critical code sections. A monitor object (say `mutex`) acts as a lock which is grabbed by executing `mutex.enter()` and then released by executing `mutex.leave()`. Also, as a way of specifying condition synchronization, a monitor object can have a set of associated `condition queues` of class `Monitor.Cond`. These can be used as dedicated wait queues for threads with specific synchronization needs. A thread willing to block will execute `q.await()` – that will release the lock on the monitor to which `q` is associated – and threads can mark blocked threads as ready to resume by invoking `q.signal()`. The “priority” tag derives from the fact that when some thread executes `q.signal()` on a nonempty queue right before executing `mutex.leave()` the signaled thread is guaranteed to resume execution right after the signaling one, i.e. there is no possibility of a third thread entering the monitor in between. As we will see shortly, this property is crucial for the scheme we are proposing. In order to ensure that signals are effective, invoking `q.waiting()` returns the number of threads blocked on `q`. Of course, the `await()`, `signal()` and `waiting()` methods can only be invoked by a thread holding the monitor associated with the condition.

The key point for the scheme to work properly is to decide how many condition queues are required to block threads and how to manage them for signaling. The basic requirement is that every condition `q` must have an associated assertion `W_q` that is proven to hold every time `q.signal()` is called and such that `W_q` logically implies the CPRE for every thread blocked on `q`. In our model-based approach this decision is made after a logical analysis of the CPRES. We will consider four cases:

1) The CPRE is `true`. In this case, no condition queue is required and no blocking will take place prior to entering the operation code.

2) The CPRE is a formula different from `true` but independent from the operation’s inputs (it might depend, however, on the resource’s internal state). In this case we consider a single condition queue for this operation as blocking or awakening a thread will only depend on the resource state.

These two cases are rather trivial but cover a small class of interesting resources, including the readers & writers protocol whose code is shown in Figure 4.6. In this example, operations `afterRead` and `afterWrite` fall in case 1) and operations `beforeRead` and `beforeWrite` in case 2). See [AMH14] for details on other examples of this kind.

In general, such a simple scheme is not applicable. Typically, resource operations have parameters and return values and some of their CPRES may depend on those parameters. A more sophisticated protocol is then needed.

---

6 A survey on the different monitor flavours, with an analysis of the pros and cons of each proposal can be found in [BFC95].

7 Also known as “condition variables”, “event queues”, etc., in the literature.

8 This idea can be traced back to [Hoa74].

9 Actually, we are assuming that the only distinction between threads blocked on such an operation is calling time and thus an FCFS policy is fine. Of course, this might not always be the case and then input parameters should be taken into account. For simplicity, we are not considering this possibility now.
When CPREs depend on the value of one or more of their parameters a one-to-one correspondence between operations and condition queues is no longer possible. However, variations on the previous scheme are still possible preserving the 0-1 principle. Because of that and taking into account the following definition, two more schemata are presented.

**Definition 4.3.1.** Consider one operation $op$ that takes arguments $x$ and $y$. Let $D_x$, $D_y$ be the domains of $x$, $y$ and $C$ the CPRE for $op$. $C$ is independent from $y$ iff

$$\forall a \in D_x. \forall b, b' \in D_y, C[a/x, b/y] \Leftrightarrow C[a/x, b'/y]$$

$C$ depends on $x$ iff $\exists a, a' \in D_x. C[a/x] \neq C[a'/x]$. $C$ partitions $D_x$ into a set of equivalence classes trivially: $a, a' \in D_x$ are equivalent iff $C[a/x]$ and $C[a'/x]$ are logically equivalent.

### III) [PARAMETER INDEXATION] The CPRE is a formula that depends on some input for the operation. The set $E$ of equivalence classes induced by that input is finite – and “small”, we should say. Then a condition for each equivalence class is created. If the CPRE does not hold at call time, then the call is blocked on the condition queue which corresponds to its equivalence class.

However, when the set of equivalence classes is infinite – or too large – the solution above cannot be applied. One possibility is to assume that the number of active threads is finite, and then each thread can be blocked in a dedicated condition queue. We call this scheme client indexation:

### IV) [CLIENT INDEXATION] When a thread calls an operation and the CPRE does not hold, a condition queue is created anew and stored in some data structure along with the inputs on which it depends. After that, the thread blocks on the condition. Right after waking up the condition queue and the associated input data can be deleted.

### 4.3.1 Trivial Cases

Figure 4.5 shows a generic template for code generation using priority monitors considering cases I and II. The code for all the operations follows a similar pattern. First, mutual exclusion is ensured by enclosing the whole operation in a monitor enter ()/leave () pair. Then the rest of the code is structured in three blocks. The first one evaluates the CPRE and blocks the calling thread – by waiting on some condition queue – if it does not hold. The second code block (ln. 23) corresponds to the operation itself and must establish the POST. Finally, the third block chooses some blocked thread that can be resumed, by executing a signal () right before leaving the monitor and returning. In the example shown the four operations share the unblocking code but this need not always be the case.

It is worth noting that the only point where sequential execution can be interrupted – if any – is the await () in the first segment. After that, the CPRE is assumed to hold, either because the CPRE was true at call time and the await () was not invoked or, because the await () was invoked once and some other thread invoked signal () awakening a proper thread, i.e after checking that the CPRE for the blocked thread was satisfied after completing operation. That is, blocked threads do not need to reevaluate their CPREs on resumption, but this requires them to check some other thread’s CPRE on exit. We call this the 0/I scheme, i.e. at most one await () is executed by a thread during the monitor call and at most one signal ().

### Readers & Writers Implementation

The first two cases are rather trivial but cover a small class of interesting resources, including the readers & writers protocol whose code is shown in Figure 4.6. In this example, operations afterRead and afterWrite fall in case (I) due to the fact that their CPRE always holds (lines 23 and 47). In those cases, the conditional could be omitted. Operations beforeRead and beforeWrite falls within case (II).

The most important part the implementation is the code that unblocks threads, shown in Figure 4.7. Notice that the assertions presented inside the method are just to explain better
private Monitor mutex;

private Monitor.Cond condOp_i;

public T_N op_i(T_1 arg_1, ..., T_M arg_M) throws PreViolationResourceException{
    mutex.enter();
    if (!PredJava(PRE_op_i(a_1,...,a_n))) {
        mutex.leave();
        throw new PreViolationResourceException();
    }
    //@ assume PredJava(PRE_op_i(a_1,...,a_n));
    T_N result = null;
    if (!PredJava(CPRE_op_i(a_1,...,a_n)))
        condOp_i.await();
    //@ assert PredJava(PRE_op_i(a_1,...,a_n)) && invariant_for(this) && ←
                  PredJava(CPRE_op_i());
    PostJava(POST_op_i(a_1,...,a_n));
    //@ assert PredJava(POST_op_i(a_1,...,a_n)) && invariant_for(this);
    // signaling code
    int signaled = 0;
    for each condOp_i && signaled == 0{
        if (condOp_i.waiting() > 0) {
            condOp_i.signal();
            //@ assert PredJava(CPRE_op_i());
            signaled ++;
            break;
        }
    }
    //@ assert signaled >= 0 && signaled <= 1;
    //@ assert PredJava(POST_op_i(a_1,...,a_n)) && invariant_for(this);
    mutex.leave();
    return result;
}

FIGURE 4.5: Generic template for implementing a shared resource using priority monitors when CPRE does not depend on any parameter.

the constraints of our template. As can be seen, this code is a ‘dummy’ in the sense that it can be easily optimized as it will show later. However, the specification for both codes (‘dummy’ and optimized) must remain the same.

This portion of code must guarantee the “liveness” of the system, i.e. if there is a blocked thread that can proceed, it must be awakened. But, the programmer has to be really careful because only threads that can proceed must be awakened: no condition is checked on the invoker code (lines 13, 25, 37, 49 of Figure 4.6). Besides, at most one thread can be unblocked. For instance, if a writer W is blocked but there are still readers performing on the system, W cannot be unblocked and it must wait until all readers finish their operations.

These constraints are shown in the JML specification for dummyUnblockingCode operation (lines 59-77). It says that only the condition queues are modifiable and a counter that accumulates the amount of signals done\(^{11}\) (line 58). By doing this, the modification of the inner attributes representing the state of the shared resource is forbidden. If signaled is 1 after executing the code, there must be an invalid blocked thread, i.e. a thread that can proceed with the execution, called effective signal. This is achieving by expressing that one

\(^{11}\)This counter can be declared as a JML ghost member of the class instead of using a variable as it is explained in Section 2.2 of the JML reference manual [Lea+09].
Chapter 4. Code Generation based on patterns

```
public class ReadersWritersMonitorNaive {
    protected /*@ spec_public @*/ int readers = 0, writers = 0;
    protected /*@ spec_public @*/ Monitor mutex = new Monitor();
    protected /*@ spec_public @*/ Cond writersCond = mutex.newCond();
    protected /*@ spec_public @*/ Cond readersCond = mutex.newCond();

    public void beforeRead() {
        //@ assume true;
        mutex.enter();
        if (! writers == 0) readersCond.await();
        //@ assert true && \invariant_for(this) && writers == 0;
        ++readers;
        //@ assert readers == \old(readers) + 1 && \invariant_for(this);
        dummyUnblockingCode();
        mutex.leave();
    }

    public void afterRead() {
        //@ assume true;
        mutex.enter();
        if (! true) readersCond.await();
        //@ assert true && \invariant_for(this) && true;
        --readers;
        dummyUnblockingCode();
        mutex.leave();
    }

    public void beforeWrite() {
        //@ assume true;
        mutex.enter();
        if (!(readers == 0 && writers == 0)) writersCond.await();
        //@ assert true && \invariant_for(this) && readers == 0 && writers == 0;
        writers++;
        //@ assert writers==1 && \invariant_for(this);
        dummyUnblockingCode();
        mutex.leave();
    }

    public void afterWrite() {
        //@ assume writers = 1;
        mutex.enter();
        if (!true) writersCond.await();
        //@ assert writers==1 && \invariant_for(this) && true;
        writers--;
        //@ assert writers==0 && \invariant_for(this);
        dummyUnblockingCode();
        mutex.leave();
    }

    /*@ spec_public @*/ int signaled = 0;
    //@ assignable readersCond, writersCond, signaled;
   /*@ ensures @*/
    protected void dummyUnblockingCode() {
        //@ threads are awaken if their associated condition holds
        /*@*/
    }
}
```

**Figure 4.6:** Readers & Writers implementation using priority monitors with unoptimized unblocking code. JML annotations were only added to write operations (beforeWrite () and afterWrite ()). PRE verification code is also omitted.

Condition queue decreases its size if and only if its associated condition holds. For instance, if a reader is awaken (readersCond decreases), the CPRE of beforeRead operation must
4.3. Model-based Code Generation Using Priority Monitors

4.3.1. Model-based Code Generation Using Priority Monitors

```java
/*@ spec_public */
int signaled;

//@ assignable readersCond, writersCond, signaled;
//@ ensures
@ { signaled == 1 implies
@ ({ readersCond.waiting() == \old(readersCond).waiting() - 1 &&
@ \old(readersCond).waiting() > 0 || writers == 0) ||
@ { writersCond.waiting() == \old(writersCond).waiting() - 1 &&
@ \old(writersCond).waiting() > 0 || writers == 0 && readers == 0})
@ }
@ } &&
@ { signaled == 0 implies
@ (readersCond.waiting() == \old(readersCond).waiting() &&
@ { readersCond.waiting() == 0 || writers == 0})
@ ||
@ (writersCond.waiting() == \old(writersCond).waiting() - 1 &&
@ { writersCond.waiting() == 0 || (writers == 0 && readers == 0)})
@ }
@ } &&
@ { signaled == 0 || signaled == 1 }
@ 
protected void
dummyUnblockingCode()
{
    signaled = 0;
    if ({ readersCond.waiting() > 0 && writers == 0}){
        signaled++;
        //@ assert writers == 0;
        readersCond.signal();
    } else if ({ writersCond.waiting() > 0 &&
        writers == 0 && readers == 0}){
        signaled++;
        //@ assert writers + readers == 0;
        writersCond.signal();
    }
    //@ assert signaled >= 0 && signaled <= 1;
}
```

**Figure 4.7:** Readers & Writers unoptimized unblocking code.

... hold. In any case, signaled is incremented when a signal was made.

However, if there no thread was awakened (signaled == 0), it is mandatory to guarantee that the previous condition does not hold. Thus, it is declared that for all conditions queue, its size is 0 (no thread are waiting) or its size did not change during the execution and its associated condition was not hold. In both cases, signaled attribute must be 0 or 1 expressing that there was an unlocked thread at most.

**Optimizing the signaling code** Sometimes, optimized signaling schemes can be obtained by a careful analysis of the interplay between the methods executing signal () and the formulas associated to each condition queue. Figure 4.8 shows an optimized version of the monitor implementation for readers and writers showed before (Figure 4.6). The code has been obtained from the table in Figure 4.10. The cell at the intersection between row i and column j contains a (minimal) formula ϕij such that POST(opi) ∧ ϕij ⇒ Wj. For instance, the first row shows that the only possible signal after executing beforeRead is on the readersCond queue, inconditionally12. The third row shows that there is no possibility of signaling after executing beforeWrite because when there is already a writer acting in the system, so no other thread can proceed. The case for afterWrite is a little bit more problematic (fourth row), as after execution the CPRES for both beforeRead and beforeWrite hold.

12 The programmer always has to check if there are threads waiting. If priority is given to writers, signaling readers are only allowed is there is no writer waiting.
In this case, the baton is passed to readers\textsuperscript{13} after checking that there is actually some thread blocked (line 67) and otherwise a signal over writersCond is issued.

The most interesting case is the second row. The right column requires an extra condition for the CPRE of beforeWrite to hold. Apparently, a conflict similar to the previous case should arise if we look at the left column. However, it can be proved that if the threads invoke the methods respecting the protocol readersCond.waiting() = 0 is an invariant at that point\textsuperscript{14}. Consequently, the only useful signal issued on writersCond (line 35).

Even when the unlocking code obtained for each method is different, all pieces must be decorated with the same specification from dummyUnblockingCode method due to the fact liveness and the effective signal properties must hold.

### 4.3.2 Parameter Indexation

When the CPRE depends on at least one of the method parameters and the set of equivalence classes induced from it is "small", then a given call is blocked on the condition corresponding to the associated equivalence class.

**Generic Template**

A generic template for this approach is presented in Figure 4.11. Instead of having one condition queue for each operation, a set of conditions queues is created: one for each of the equivalence classes representatives defined over the domain of the dependent parameters, as is expressed in lines 2-4. Then, if the CPRE does not hold, the call is blocked on the condition associated with the representative of the equivalence class of the current value of the parameter, as it is shown in line 16-17.

Then, the programmer is in charge of writing the code that must accomplish the POST of the method without manipulating the conditions defined.

Following with the unblocking part, the code has to ensure that the associated predicated of the signaled condition queue implies the CPRE of the method invocation, assuring that the awaken thread can safely continue with the execution.

**Shipping Warehouse Implementation**

Let us illustrate the parameter indexation method with the shipping warehouse code shown in Figure 4.12. The CPRE of enterWarehouse depends on the \(n\) and \(w\) parameters. As \(D_n = 1..N\_WAREHOUSES\) and \(D_w = 1..MAX\_WEIGHT\), the set \(E_{ew}\) of equivalence classes is isomorphic to \(D_n \times D_w\) and thus a bidimensional array freedWarehouse of condition queues is used. For exitWarehouse, however, the set \(E_{ex}\) of equivalence classes is isomorphic to \(D_n\) and a two-dimension array freedCorridor is used instead. By construction, the condition associated to each queue \(\text{freedCorridor}[i][j]\) is exactly \(\text{CPRE(enterWarehouse}(n, w))[(i/n, j/w)]\) and thus the basic requirement is trivially fulfilled.

In our example, having one condition per warehouse and weight can be considered an overkill. Client indexation can be used to optimize the previous schema in several ways. First, if we consider the robots trying to enter warehouse zero, rather than having an array of conditions indexed by weight, we can just store pairs \((q_i, w_i)\). Moreover, if a priority queue is used, several strategies can be implemented, such as lightest robot first, etc. Second, for the rest of warehouses a more radical optimization can be applied. As there is only room for one robot in the corridor, we can assume that at most one thread is blocked trying to enter warehouse \(i > 0\), so just one queue and one weight will suffice. The code for these optimized versions can be found in [AMH14].

In the version shown in Figure 4.12 the unblocking code is shared among the two operations. It represents the worst case for signaling by just iterating over the condition arrays of threads and checks: (i) that the assertion for a given queue holds, and (ii) that there is a

\textsuperscript{13} As it was explained before, the priority of this example if given to readers. The priority can be easily changed by processing writers condition queue first.

\textsuperscript{14} All readers are awaken in beforeRead operation or go through the system because beforeRead CPRE is true. Nevertheless, this scenario is not possible when the priority is given to writers threads as is shown in Figure 4.9.
4.3. Model-based Code Generation Using Priority Monitors

```java
public class ReadersWritersMonitorOptimized {
    protected /*@ spec_public @*/ int readers = 0, writers = 0;
    protected /*@ spec_public @*/ Monitor mutex = new Monitor();
    protected /*@ spec_public @*/ Cond writersCond = mutex.newCond();
    protected /*@ spec_public @*/ Cond readersCond = mutex.newCond();

    public void beforeRead() {
        mutex.enter();
        if (!writers == 0) readersCond.await();
        //@ assert true && invariant_for(this) && writers == 0;
        ++readers;
        //@ assert readers == old(readers) + 1 && invariant_for(this);
        unblocking0N();
        mutex.leave();
    }

    public void afterRead() {
        mutex.enter();
        if (!true) readersCond.await();
        //@ assert true && invariant_for(this) && true;
        --readers;
        unblocking00();
        mutex.leave();
    }

    public void beforeWrite() {
        mutex.enter();
        if (!((readers == 0 && writers == 0)) writersCond.await();
        //@ assert true && invariant_for(this) && readers == 0 && writers == 0;
        writers++;
        //@ assert writers==1 && invariant_for(this);
        mutex.leave();
    }

    public void afterWrite() {
        //@ assume writers = 1 ;
        mutex.enter();
        if (!true) writersCond.await();
        //@ assert writers==1 && invariant_for(this) && true;
        writers--;  
        //@ assert writers==0 && invariant_for(this);
        unblocking00();
        mutex.leave();
    }

    protected void unblocking00() {
        signaled = 0;
        if (readersCond.waiting() > 0){
            signaled++; readersCond.signal();
        } else if (writersCond.waiting() > 0){
            signaled++; writersCond.signal();
        }
    }

    protected void unblocking0N() {
        signaled = 0;
        if (readersCond.waiting() > 0){
            signaled++; readersCond.signal();
        }
    }
}
```

Figure 4.8: Readers & Writers implementation using priority monitors with optimized signaling code and readers priority. JML ensures is simplified but equivalent to the one presented in Figure 4.7.
Chapter 4. Code Generation based on patterns

<table>
<thead>
<tr>
<th>cond. queue</th>
<th>method</th>
<th>readersCond</th>
<th>writersCond</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beforeRead</td>
<td>readersCond.waiting() == 0</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>afterRead</td>
<td>readersCond.waiting() == 0</td>
<td>readers = 0</td>
</tr>
<tr>
<td></td>
<td>beforeWrite</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>afterWrite</td>
<td>writersCond.waiting() == 0</td>
<td>true</td>
</tr>
</tbody>
</table>

**Figure 4.9:** Signaling table for the Readers & Writers monitor implementation with writers priority.

<table>
<thead>
<tr>
<th>cond. queue</th>
<th>signaling method</th>
<th>readersCond</th>
<th>writersCond</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>beforeRead</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>afterRead</td>
<td>true*</td>
<td>readers = 0</td>
</tr>
<tr>
<td></td>
<td>beforeWrite</td>
<td>false</td>
<td>false</td>
</tr>
<tr>
<td></td>
<td>afterWrite</td>
<td>true</td>
<td>true</td>
</tr>
</tbody>
</table>

**Figure 4.10:** Signaling table for the Readers & Writers monitor implementation with readers priority.

thread actually blocked on that queue (using the waiting () method comparison). If so, the thread is awakened and the iteration stops to preserve the 0/1 scheme.

Ensuring (i) requires that the signaling thread is able to recover the safety assertion Wq for a given condition q on the unblocking piece of code. In the sample code this is achieved by reconstructing the values for the n and w inputs from the array indexes and passed them to the associated CPRE. If the solution is based on client indexation, this may be accomplished by recovering a copy of the saved parameters as it will be illustrated in the following section.

4.3.3 Client Indexation

When the set of equivalence classes is not so “small” or is impractical to be handled using one condition queue for each representative, a new approach should be followed.

The generic template for this technique is illustrated in Figure 4.13. In this case, when a thread makes an operation call but the CPRE does not hold, the code block that calls on a brand new condition c for that call (line 15). Then, if the CPRE depends on some parameters, say x, the pair <x, c> is added to a map data structure (line 16) associated to the call operation15. Notice that the code still use an conditional statement instead of a loop to check the CPRE.

After the code of the method obtained from the POST, the unblocking code must signal an invalid blocked thread. For that, an iteration over all the data structures associated to any method is performed seeking for a “signable” thread, i.e. a blocked call whose associated CPRE holds. Thus, for each pair presented in the map, the code checks either the corresponding method CPRE using the store parameters (line 33) holds or not. If it holds, the pair is removed from the map (line 34) and the thread blocked in the condition is signaled (line 35). Notice that the “0-1” schema must be accomplished in this template. Besides, if there any invalid blocked thread in the map, it must be awaken in order to guarantee the liveness property of the system.

Shipping Warehouse Implementation

Figure 4.14 outlines the implementation of the exitWarehouse operation using client indexation. In this case, we have inverted the map by using Monitor.Cond objects as key

---

15 If the CPRE is independent, the code only stores the new condition queue in a method associated list.
4.3. Model-based Code Generation Using Priority Monitors

```java
private Monitor mutex;

private Monitor.Cond q1;
...
private Monitor.Cond qM;

public TN op1(T1 arg1, ..., TM argM)
    throws PreViolationResourceException{
    mutex.enter();
    if (!PredJava(PREop1(a1, ..., an))) {
        mutex.leave();
        throw new PreViolationResourceException();
    }
    //@ assume PredJML(PREop1(a1, ..., an));
    TN result = null;
    if (!PredJava(CPREop1(a1, ..., an)))
        qj.await();
    //@ assert PredJML(PREop1(a1, ..., an)) && invariant_for(this) && ←
    PostJava(POSTop1(a1, ..., an));
    //@ assert PredJML(POSTop1(a1, ..., an)) && invariant_for(this);
    mutex.leave();
    return result;
}
```

FIGURE 4.11: Generic template for implementing a shared resource using the parameter indexation technique.
public class WarehouseAccessControlMonitorNaive {

// Resource state
private int weight[];
private boolean occupied[];

// Handling concurrency
// ensures mutual exclusion
private Monitor mutex;
private Monitor.Cond freedWarehouse[][] = new Monitor.Cond[N_WAREHOUSES][MAX_WEIGHT];
private Monitor.Cond freedCorridor[] = new Monitor.Cond[N_WAREHOUSES];

public WarehouseResource() {
    // initialize state monitors and conditions...
}

public void enterWarehouse(int n, int w) {
    mutex.enter();
    if (w + weight[n] > MAX_WEIGHT)
        freedWarehouse[n][w].await();
    //@ assert weight[n] + w <= MAX_WEIGHT;
    if (n != 0) occupied[w-1] = false;
    weight[n] += w;
    dummyUnblockingCode();
    mutex.leave();
}

public void exitWarehouse(int n, int w) {
    mutex.enter();
    if (n < N_WAREHOUSES - 1 && occupied[n+1])
        freedCorridor[n+1].await();
    //@ assert n == N_WAREHOUSES-1 || !occupied[n];
    weight[n]-= w;
    if (n != N_WAREHOUSES - 1) occupied[n] = true;
    dummyUnblockingCode();
    mutex.leave();
}

private void dummyUnblockingCode(){
    int signaled = 0;
    for(int i=0; i<N_WAREHOUSES && signaled==0; i++){
        for (int j=0; j <= MAX_WEIGHT && !signaled; j++) {
            if (weight[i] + j <= MAX_WEIGHT //cpreEnter
                && freedWarehouse[i][j].waiting() > 0){
                freedWarehouse[i][j].signal();
                signaled ++;
            }
        }
    }
    for(int i=0; i<N_WAREHOUSES && signaled==0; i++){
        if ((i==N_WAREHOUSES-1 || occupied[i]) &&
            freedCorridor[i].waiting()>0) {
            freedCorridor[i].signal();
            signaled ++;
        }
    }
    //@ assert signaled >= 0 && signaled <= 1;
}

}
4.3. Model-based Code Generation Using Priority Monitors

```java
private Monitor mutex;
private Map<Set, Monitor.Cond> conditionsOp;

public T op(T arg1, ..., T argM)
    throws PreViolationResourceException{
    mutex.enter();
    if (!PredJava(PRE_op1(a1, ..., an))) {
        mutex.leave();
        throw new PreViolationResourceException();
    }
    //@ assume PredJava(PRE_op1(a1, ..., an));
    T result = null;
    if (!PredJava(CPRE_op1(a1, ..., an))) {
        Monitor.Cond c = mutex.newCond();
        conditionsOp.put(new Set(a1, ..., an), c);
        c.await();
    }
    //@ assert PredJava(PRE_op1(a1, ..., an)) && 
    //@ && invariant_for(this);
    PostJava(POST_op1(a1, ..., an));
    //@ assert PredJava(POST_op1(a1, ..., an)) && 
    //@ && invariant_for(this);
    //@
    for (each condition conditionsOp, && signaled == 0 ) {
        for (Map.Entry<Set, Monitor.Cond> entry:
            conditionsOp.entrySet())
            if (!PredJava(CPRE_op1(entry.getKey()))) {
                signaled++; 
                break;
            
        }
    }
    return result;
}
```

FIGURE 4.13: Generic template for implementing a shared resource using the client indexation technique. With a slight notational abuse, we have used the outcome of a Java method call (entry.getKey()) to instantiate the value of one CPRE with a set of actual parameters. A similar “trick” will be used in Figure 4.22.
public class WarehouseAccessControlMonitorNaive {
    private /*@ spec_public @*/ Monitor mutex;
    private /*@ spec_public @*/ int weight[];
    private /*@ spec_public @*/ boolean occupied[];
    private /*@ spec_public @*/ Monitor mutex;
    private /*@ spec_public @*/ Cond enteringWarehouse[][];
    private /*@ spec_public @*/ Map<Monitor.Cond, Integer> conditionsExiting;

    public void exitWarehouse(int n, int w) {
        mutex.enter();
        //@ assume true;
        if (n != N_WAREHOUSES -1 && occupied[n]) {
            Monitor.Cond cond = mutex.newCond();
            conditionsExiting.put(cond, n);
            cond.await();
        }

        //@ assert true && \invariant_for(this) &&
        @ (warehouse == N_WAREHOUSES -1 || !corridor[warehouse]);
        weight[n] -= w;
        if (n != N_WAREHOUSES -1)
            occupied[n] = true;
        dummyUnblockingCode();
        mutex.leave();
    }

    /*@ spec_public @*/ int signaled;
    private void dummyUnblockingCode(){
        signaled = 0;

        // trying to unblock a robot that wants to enter
        ...same piece of code from Figure 4.12(lines 51-65)

        // trying to unblock a robot that wants to leave the warehouse...
        // for each warehouse...
        for(Entry<Monitor.Cond, Integer> entry: conditionsExiting.entrySet()){
            if (entry.getValue() == N_WAREHOUSES -1 || !corridor[entry.getValue()]) {
                conditionsExiting.remove(entry.getKey());
                entry.getKey().signal();
                //@ assert entry.getValue()==N_WAREHOUSES -1 || !corridor[entry.getValue()];
                signaled ++;
                break;
            }
        }
    }

    ...}

Figure 4.14: Shipping Warehouse implementation using client indexation technique for monitors.

instead of the number of warehouse for practical reasons16 (see line 10).

The rest of the implementation is quite straightforward. When the CPRE of a call does not hold, a brand new condition is created (line 17) and then stored in the map with the warehouse id (needed to check exitWarehouse CPRE).

16 A map associates a specified value with a specified key in the structure. If the map previously contained a mapping for the key, the old value is replaced by the specified value.
4.4 Model-based Code Generation Using Java CSP implementation

The unblocking code differs slightly from the Figure 4.12. The iteration over the condition queues instantiated for the `enterWarehouse` remains untouched but the second loop changes a little bit. For each entry, the id of the warehouse is recovered and the CPRE is checked (lines 54-55). If it holds, there is a thread blocked in the condition that must be awakened (line 57). Notice that it is mandatory to remove the pair from the map and that there is no need to check the existence of the thread. Notice that in both cases the “0-1” schema must hold.

4.4 Model-based Code Generation Using Java CSP implementation

The following paragraphs introduce another two templates for translating a shared resource specification into a Java class implementation using the JCSP library [WAB12]. For each proposal, we follow the same approach as with the already presented monitor-based templates.

This third-party library provides an implementation of CSP\(^{17}\), the well-known theory for specifying and verifying complex patterns of behavior arising from interactions between concurrent objects. It encapsulates fundamental principles of communication by yielding sub-hundred-nano-second overheads for context switching, process startup/shutdown, synchronized channel communication and high-level shared-memory locks. The main components of CSP are messages and channels. While a message is just any object (even a channel), a channel is a model for synchronization. A message may be sent over a channel, and another process is able to read the messages sent over a channel it has a reference to. Those channels are inherently synchronous (rendezvous behavior), i.e. a thread waiting to receive a message from a channel will block until the message is sent.

The core idea of the translation is to identify the shared resource with a `server process` that manages the resource’s internal state in its local memory and deals with the requests of the other (design level) threads in a client/server fashion. In order to avoid contamination of the clients’ code we encourage the creation of a wrapper that hides the actual implementation of the resource. That is, a JCSP-based implementation of a given resource will usually implement both the `CSProcess` interface – so that the public `run()` method implements the server – and the interface where the specification of the shared resource lays – which will force the implementor to provide definitions for the resource operations. However, the implementations of the resource operations will only consist in message-passing operations to and from the server, which does the “real” job, resulting in a kind of remote procedure call style.

This architecture is shown graphically in Figure 4.15. That is, clients of the shared resource cannot distinguish between an implementation using CSP or another concurrency mechanism. Summarizing, the implementation consists of the two aforementioned components whose responsibilities are:

**Wrapper:** Receiving method invocations and propagating them as messages to the server through CSP channels. Also, if the operation returns a value, waiting for the

\(^{17}\) Another popular CSP implementation is *Communicating Threads in Java (CTJ)* [Hil+97]. There is also available a detailed comparison between JCSP and CTJ in terms of their philosophies, similarities and differences, performance, and usage scenarios in [SHW08].
server to produce such a value and read it from a channel (which may have been created at method invocation time).

Server: Processing the requests received from the wrapper methods and modifying the shared resource inner state according to methods POST specification. Of course, requests should only be served if the CPRE is satisfied.

Applying these ideas to a simple resource like the Readers & Writers protocol (Figure A.21) is a fairly easy task. None of the four operations take arguments nor return anything, so their implementation consists in sending out a meaningless token (a null). A channel for each operation works fine. For the after-operations, the CPRE is true, so the channel will always be available and the operation will take virtually no time from a client’s perspective. For the before-operations, the CPRE is a formula which only depends on the resource’s internal state, which makes it possible to enable or disable the corresponding channel using the conditional reception mechanism in CSP. This is implemented in JCSP by creating an object of the Alternative class from a four-channel array, and then invoking the fairSelect (conds) method on that object, where conds encodes the corresponding CPRES. The resulting code is shown in Figure 4.16.

The structure of the service loop inside the run () method is quite simple. The syncCond array with the reception conditions is computed first and then, depending on the index value returned by fairSelect (syncCond), the code for the selected operation is executed. Observe that the code that evaluates syncCond could be slightly optimized to avoid reassigning positions with constant values (i.e. deleting lines 45 and 46). In the present example this is trivial but, in general, optimizing this part of the code trying to avoid recomputing positions that have not changed since the previous iteration is one of the most common sources of mistakes.

Given the structure of the code, a set of proof obligations can be imposed at certain places. These will set the basis for the verification techniques described in Chapter 5:

- The preconditions (PREs) are often independent from state changes in the resource (e.g. those in Figure A.28, but not those in Figure A.21). In such cases, they should be checked at call time – that is, inside the wrapper code – and throw the corresponding exception although, for brevity, we have not included that in these examples. In any case, they must be assumed before the relevant operations (lines 17, 24 of Figure 4.16).

- The invariant (INVARIANT) maps to the loop invariant within the server code. This is because the structure of this code serializes the operations (refusing them until their CPRES hold), performing one (and only one) in each loop – and no state changes happen between the start of the loop and the start of an operation. Therefore, the proof obligation regarding the invariant is to establish it before first entry to the loop (line 41) and that it is restored by the end of the loop (line 65). Again because of the structure of the server code, the end of the loop is the end of each operation (lines 55, 60, 64, 68).

- The concurrent or synchronization pre-condition (CPRE) must hold right before entering the code for each operation (lines 53, 58);

- The post-condition (POST) must hold on exit of the code of each operation (lines 55, 60, 64, 68).

In our model-based approach this decision is, again, made after a logical analysis of the CPRES. From that, four cases can be distinguished:

1) The CPRE is true. Thus, a unique ‘always-open’ channel per operation is required and all messages are processed when the server receives them.

2) The CPRE is a formula independent from the operation’s inputs. In this case, a single channel is also needed but its status depends on the inner state of the resource.

However, as explained for the monitor-based template, such a simple scheme is not always possible. A more sophisticated protocol is then needed for the server to decide when a request may be processed. That is why we present some variation of the previous two cases.
4.4. Model-based Code Generation Using Java CSP implementation

public class ReadersWritersCSP implements CSProcess, ReadersWriters {
    protected final /*@ spec_public @*/ Any2OneChannel chBeforeWrite;
    protected final /*@ spec_public @*/ Any2OneChannel chBeforeRead;
    protected final /*@ spec_public @*/ Any2OneChannel chAfterWrite;
    protected final /*@ spec_public @*/ Any2OneChannel chAfterRead;
    protected /*@ spec_public @*/ int readers = 0, writers = 0;

    // interface ReadersWriters
    public void beforeWrite() {
        chBeforeWrite.out().write(null);
    }

    public void afterWrite() {
        //@ assume writers == 1;
        chAfterWrite.out().write(null);
    }

    public void beforeRead() {
        chBeforeRead.out().write(null);
    }

    public void afterRead() {
        //@ assume readers > 0;
        chAfterRead.out().write(null);
    }

    // server side
    public void run() {
        // One entry for each method
        Guard[] guards = {chBeforeWrite.in(), chBeforeRead.in(),
                          chAfterWrite.in(), chAfterRead.in()};
        final int BEFORE_WRITE = 0, BEFORE_READ = 1, AFTER_WRITE = 2, AFTER_READ = 3;

        boolean syncCond[] = new boolean[4]; // conditional reception
        Alternative services = new Alternative(guards);
        while (true) {
            syncCond[BEFORE_WRITE] = (readers == 0 && writers == 0);
            syncCond[BEFORE_READ] = writers == 0;
            syncCond[AFTER_WRITE] = true; syncCond[AFTER_READ] = true;

            int service = services.fairSelect(syncCond);
            switch(service) {
                case BEFORE_WRITE:
                    chBeforeWrite.in().startRead();
                    //@ assert writers == 0 && readers == 0;
                    writers++;
                    chBeforeWrite.in().endRead();
                    break;

                case AFTER_WRITE:
                    chAfterWrite.in().startRead();
                    writers--;
                    chAfterWrite.in().endRead();
                    break;

                ... analogously for AFTER_READ and BEFORE_READ cases ...
                // end switch
            } // end while
        } // end run
    }
}

Figure 4.16: Readers & Writers implementation using JCSP with channel replication technique.

III) [CHANNEL REPLICATION] When CPRE formula depends on some input for the operation but the set of equivalence classes induced is finite (and relatively small), a channel for each equivalence class is created. If the CPRE is evaluated as true at call time, then the channel will be opened for the server point of view.

IV) [DEFERRED REQUESTS] When CPRE depends on some input but the set of equivalence
classes induced is either infinite or too big, a unique channel for the operation is used. Every call is translated into a message including a fresh channel (allowing the communication between the server and the invoker) and the input data needed to check the associated CPRE.

The latter two styles are fully described in the following sections\(^\text{18}\) by showing how the CPRE can be systematically treated.

### 4.4.1 Channel Replication

When an operation’s CPRE depends on some of its parameters, the server cannot receive the request for that operation on a single channel and decide whether it addresses the request based on data it has not received yet. Indeed the same channel should be open for some requests and closed for others at the same time. For example, reconsider the Shipping Warehouse specification (Figure A.28). As it is stated before, CPREs depend on values obtained from their parameters (the number of the warehouse and the robot’s weight).

**Generic Template**

The first solution considered drops the assumption that every operation has a single dedicated channel to the server. In fact, if we manage to use so many channels as requests for implementing some operation never coincide on the same channel when their CPREs evaluate to different boolean values, then the problem is solved. Systematically doing this corresponds to the technique we have named *channel replication* and the generic template for this approach is presented in Figure 4.17.

Considering the set \( E_i \) of equivalence classes induced by the parameter \( x \) of the operation \( \text{op}_i \) as is expressed in the Definition 4.3.1. If \( E_i \) is finite, then one channel can be defined for each equivalence class (as is represented by the conditions list in line 1). This way, we ensure that two requests serving \( \text{op}_i(a, b) \) and \( \text{op}_i(a', b) \) will only be routed to the same channel if the precondition holds (or fails) for both, as required above (line 8).

Due to the signature of fairselect CSP operation, these channels will have to be placed somewhere inside the overall array of channels of the server process as is shown in line 23. This leads to the following definitions:

**Definition 4.4.1 (Parameter Recovery functions).** Consider one operation \( \text{op}_i \) that takes arguments \( x \) an \( y \) so that its CPRE \( C_i \) depends on \( x \). Let \( D_x, D_y \) be the domains of \( x, y \) and \( E_x \) the set of equivalence classes induced by the parameter \( x \). As \( E_x \) is finite, there must be a partial injective function

\[
pe_i : E_x \to \mathbb{N}
\]

Therefore, by construction of \( E_x \), there must also exist a (total) injection

\[
px_i : D_x \to E_x
\]

These functions are used in the following way: \( pe_i \) maps an integer (array position) to each element of the set equivalence class while \( px_i \) helps us select which channel to send the message through. That means that in order to route a call to \( \text{op}_i(a, b) \), the wrapper method will have to send a request over the channel that occupies position \( pe_i(px_i(a)) \) from the server’s perspective (line 8). The server is responsible for enabling that channel based on the values of the inner resource. This is accomplished by the update made over the syncCs array, making this piece of code critical due to the fact that enabling an erroneous channel could lead to a system failure.

If the same technique is applied on another operation \( \text{op}_j \), it must be ensured that the ranges of \( pe_i \) and \( pe_j \) are disjoint. This will allow the server to identify the operation from the index returned by the fairSelect method. As a final remark, if some operation depends on several parameters, the method can be adapted using products as long as the finiteness of the equivalence classes condition holds for each parameter. Also, observe that the case of

---

\(^{18}\)The resemblance with the monitor templates should be evident. We are using synchronous channels to emulate the functionality provided by condition queues.
private final List<Any2OneChannel> channels;

// WRAPPER IMPLEMENTATION */
public T N op1(T1 arg1, ..., T M argM) throws PreViolationResourceException{
//@ assert Pred[JML](PRE op1(a1,...,an));
One2OneChannel chResp = Channel.one2one();
channels[pxi(px(argl,...,argr))].out().write(new Request(chResp.out(), argl,...,argr));
//@ assert Pred[JML](CPRE opi(a1,...,an));
T N res = (T N)chResp.in().read();
//@ assert Pred[JML](POST op1(a1,...,an)) && invariant_for(this);
return res;
}

// SERVER IMPLEMENTATION */
private static final int METHODiXl = fun(i, l);
//@ Object ghost params;
public void run() {
//Union of all method's associated channels
Guard[] inputs={...methodiXlChannel.in(),...};
// Conditional reception for fairSelect().
//@ assert inputs.length == K+channels.size();
boolean syncCs[] = new boolean[K+channels.size()];
... initialized syncCs array
final Alternative services = new Alternative(inputs);
int service;
while (true) { // Server loop
// refreshing synchronization conditions
... updating syncCs array
//@ assert (\forall int i; i< syncCs.length; syncCs[i] => C[i]);
service = services.fairSelect(syncCs);
//@ assume service < guards.length && service >= 0 && syncCs[service] &&
@ guards[service].pending() > 0;
@/ switch(choice){
...
// receiving and processing the request
case METHODiXl:
//@ set params = pxi−1(px−1(METHODiXl));
//@ assert Pred[JML](PRE opi(params)) && Pred[JML](CPRE opi(params));
Request request = ((Request) ((AltingChannelInput) inputs[METHODiXl]).←
read());
T N result;
Post2Java(POSTopi1(params));
//@ assert Pred[JML](POSTopi1(params)) && invariant_for(this);
request.getChannel().out().write(result);
break;
} // end while
} // end run

Figure 4.17: Generic template for implementing a shared resource using the channel replication technique over JCSP. Channel definition based on the domain of the input parameters.
private int weight[]; // # = N_WAREHOUSES
private boolean occupied[]; // # = N_WAREHOUSES

/** WRAPPER IMPLEMENTATION */
Any2OneChannel chEnteringWarehouse[][];
Any2OneChannel chExitingWarehouse[];

public WarehouseAccessControlCSP(int n, int w) {
... initialize state and create channel
}

public void enterWarehouse(int n, int w) {
//@ assume n => 0 && n < N_WAREHOUSES && w > 0;
chEnteringWarehouse[n][w-1].out().write(
    new WeightedRequest(n,w));
}

public void exitWarehouse(int n, int w) {
//@ assume n => 0 && n < N_WAREHOUSES && w > 0;
chExitingWarehouse[n].out().write(w);
}

Figure 4.18: Warehouse implementation using the channel replication technique. Channel definition based on the domain of the input parameters. Wrapper (client side) operations.

CPREs independent of parameters can be seen as a degenerate case where there is only one class of equivalence.

Shipping Warehouse Implementation

Figure 4.18 outlines an implementation of the shipping warehouse resource using the channel replication technique.

Following the same analysis of the warehouse operations CPRE presented in Section 4.3, enteringWarehouse operation depends on both parameters: n (id of the warehouse which finite domain is Dn) and w (weight of the robot with finite domain is Dw). Therefore, enteringWarehouse can be dealt with by sending its message down an appropriately chosen element from the matrix of channels (chEnteringWarehouse, position (n,w)). Hence, requests that it cannot at the moment handle can be refused (i.e. blocked).

In the case of exitWarehouse, it only depends on the parameter n. Therefore, these operations can be handled similarly to the previous one, by sending a message down one of a different array of channels (chExitingWarehouse).

Figure 4.19 outlines an implementation of the process serving all the requests. The channels are put into a Guard array (ls. 55-62) according to the pe functions:

\[ pe_{nw}(n, w) = n \]
\[ pe_{ew}(n, w) = N\_WAREHOUSES + n \times MAX\_WEIGHT + w \]

Notice that pe_{nw} is the flattening function that maps each position of the matrix of channels to the array passed as argument to fairSelect. Figure 4.20 illustrates pe and px graphically.

The JCSP Alternative is constructed with these guards (line 64), which will be fairSelected using the syncCond array of pre-conditions (line 67). Since these pre-conditions depend on the inner state of the shared resource (current weight in each warehouse and the robots present in the corridors), they are computed at the start of each service loop (lines 71-81).
```java
public void run() {
    /* inputs[0..N_WAREHOUSES] -> exitWarehouse
     * inputs[N_WAREHOUSES..] -> enterWarehouse
     */
    Guard[] inputs = new Guard [N_WAREHOUSES + (MAX_WEIGHT * N_WAREHOUSES)];
    for (int n = 0; n < N_WAREHOUSES; n++) {
        inputs[n] = chExitingWarehouse[n].in();
        // flattening enteringWarehouse channels
        for (int w = 0; w < MAX_WEIGHT; w++) {
            inputs[N_WAREHOUSES + n*MAX_WEIGHT+w] = chEnteringWarehouse[n][w].in();
        }
    }

    Alternative services = new Alternative(inputs);

    boolean[] syncCond = new boolean [N_WAREHOUSES + (MAX_WEIGHT * N_WAREHOUSES)];
    while (true) {
        // updating syncCond for enterWarehouse(0,w)
        for (int n = 0; n < N_WAREHOUSES; n++) {
            //exitWarehouse(n,w) CPRE
            syncCond[n] = ! (n != N_WAREHOUSES || !occupied[n]);
            // enterWarehouse(n,w) CPRE
            for (int w = 0; w < MAX_WEIGHT; w++) {
                syncCond[N_WAREHOUSES + n*MAX_WEIGHT+w] = weight[n] + (w+1) <= MAX_WEIGHT;
            }
        }
        ...
        int service = services.fairSelect(syncCond);
        ...
        if (service < N_WAREHOUSES) { // exit OP
            //@ assert service != N_WAREHOUSES ==> !occupied[service];
            ... make POST for exitWarehouse hold
        } else { //service>=N_WAREHOUSES ==> enter
            //@ assert weight[service] + w <= MAX_WEIGHT;
            ... make POST for enterWarehouse hold
        }
    } // end of server loop
} // end of run method
```

Figure 4.19: Warehouse implementation using the channel replication technique. Server component code.

![Diagram](image)

Figure 4.20: Graphical representation of the syncCond array for the warehouse problem with two warehouses (N_WAREHOUSES == 2) and a maximum weight of two (MAX_WEIGHT == 2). Starting from the left corner, the first two positions are used for enterWarehouse CPRE whilst the other for exitWarehouse CPRE.

The fact that the $pe$ functions have disjoint ranges ensures the existence of an inverse mapping from $N$ to \{enterWarehouse, exitWarehouse\} that is used to translate the value returned by fairSelect into the original operation (line 88) and, then, to recover the original
parameters by taking the inverses $px_{eq}^{-1}$ and $px_{eq}^{-1}$ (lines 93 and 99).

### 4.4.2 Deferred Requests

One restriction that is imposed by the previous template, is the necessity of having a “small” finite set of equivalence classes of all calls to some resource operation, modulo logical equivalence of their CPRES. Even when those set are relatively small in size, the definition of the mappings $(pe_i, px_j, \ldots)$ involved could be error-prone. As an alternative to the previously presented schema, we introduce a new template called deferred requests that does not suffer from these problems. As we will see, the only requirement is that the number of clients is finite.

**Generic Template**

In Figures 4.21 and 4.22 the generic template of the Wrapper and Server Components, respectively, for this approach are presented. The core idea is to have a unique channel per operation as can be seen in lines 1-3. As explained before, this implies that if an operation CPRE depends on a parameter, the condition cannot be encoded in the corresponding reception list. Hence, the method channel is always open but the wrapper send a request instead of a simple message (lines 15-20, 43-55). It must contain the footprint of the parameters (values needed to compute the CPRE) and a fresh channel used either to keep the client blocked until the operation has been performed (in the case that no results are to be returned) or for information exchange.

**Shipping Warehouse Implementation**

Figure 4.23 outlines the implementation of the wrapper component operations for the warehouse example using the aforementioned technique. The message sent consists of a fresh channel (lines 21 and 31) bundled together with a footprint of the method parameters that contains sufficient information for the server to compute the relevant CPRE. Depending on the content of the footprint sent, two cases can be distinguished:

- **single send**: when it contains all the actual parameters (e.g. the enterWarehouse operation, line 19), the method call can proceed without sending any further information;

- **double send**: when the footprint does not contain all the parameter information (e.g. the exitWarehouse operation, line 33), having to send the rest of the formal parameter through the already created channel (line 35).

Notice that in both cases, the client will block its execution until a message (the return value or a meaningless token) is written into the invocation associated channel notifying that the request was processed successfully (lines 25 and 37 of Figure 4.23).

Nevertheless, the tricky part comes when designing the server code (Figure 4.24). This component is a sequential process that defers all requests into some auxiliary data structure, to be processed later. In order to avoid errors and to obtain a correct implementation of this component, a set of considerations must be taken into account:

- every request is stored in some data structure as soon as it is received by the server. Typically, there will be one collection per method;

- it must be ensured that no pending request whose synchronization condition holds is left unattended before entering into a new iteration of the service loop;

- finally, mutual exclusion of the servicing of the requests must be guaranteed by the server implementation.

Considering the first point, requests are not served in the same order as they are processed by the server, so the implementation of component has the responsibility of ensuring fairness. Of course, the data structures chosen to store pending requests must facilitate searching for the requests whose synchronization precondition holds.
private Any2OneChannel chOperation1;
private Any2OneChannel chOperationM;
private Any2OneChannel chOperationM;

// single send: PRE depends on the actual parameters and on the resource state
public T no(I arg1, ..., M argM)
    throws PreViolationResourceException{
    post(I arg1, ..., argM)
    // before deferring the call
    One2OneChannel chResp = Channel.one2one();
    OpIPetition pet = new OpIPetition(arg1, ..., argM, chResp.out());
    // sending petition to the server
    chOperation.out().write(pet);
    // waiting for notification from the server
    Object obj = chResp.in().read();
    if (obj instanceof PreViolationResourceException){
        throw (PreViolationResourceException) obj;
    }
    // assert
    Pred JML (post(I arg1, ..., argM)) && invariant_for(this);
    return (T) obj;
}

// double send: when the footprint does not contain all parameter information.
public T no(I arg1, ..., M argM) throws PreViolationResourceException{
    One2OneChannel chResp = Channel.one2one();
    OpIPetition pet = new OpIPetition(argk, ..., argN, chResp);
    // sending footprint to the server
    chOperation.out().write(pet);
    // assert
    Pred JML (pre(I arg1, ..., argM)) && Pred JML (pre(Cpre(I arg1, ..., argM)))
    && invariant_for(this);
    // sending the rest of the parameters
    chOperation.out().write({arg1, ..., argk-1});
    // waiting for notification from the server
    Object obj = chResp.in().read();
    // assert
    Pred JML (post(I arg1, ..., argM))
    && invariant_for(this);
    return (T) obj;
}

Figure 4.21: Generic Wrapper Component of the deferred requests template.

Regarding the other points, if some enabled request is left unattended and the server invokes fairSelect, this may force a client block unnecessarily, with the resulting loss of concurrency. Finally, mutual exclusion is guaranteed as long as the server is purely sequential code (processing each request completely without being interleaved with another one).

Getting back to the warehouse example, the reception of messages in the server code is straightforward. In the implementation, the requests are simply put in a list – one for each operation. Figure 4.24 shows part of the server code component for treating the requests.

The basic structure of the code is a set of repetitive sentences. The first loop (line 102) unblocks exitWarehouse requests whilst the second (line 134) processes all enterWarehouse
/* SERVER IMPLEMENTATION */
/** for each method, a list for deferring */
final Queue<Operation1Petition> operation1Requests;
//
final Queue<OperationNPetition> operationNRequests;
/** Constants representing API method's */
private static final int OPERATION_1 = 0;
//
private static final int OPERATION_M = M-1;

public void run() {
    /* One entry for each method. */
    Guard[] inputs = { chOperation1, ..., chOperationM};
    Alternative services = new Alternative(inputs);
    
    while (true) { // receiving a petition from the wrapper
        int service = services.fairSelect();
        switch (service) {
            case OPERATION_1:
                Operation1Petition request = (Operation1Petition) chOperation1.in().read();
                operation1Requests.offer(request);
                break;
            ...
        }
        // processing all possible requests
        boolean anyResumed;
        do {
            anyResumed = false;
            for (each list operationIRequests) {
                int lastItem = exitRequests.size();
                for (int i = 0; i < lastItem; i++) {
                    Operation1Petition curr = operationIRequests.pull();
                    if (Pred2Java(CPRE_op1(curr.a1, ..., curr.ann))) {
                        Post2Java(POST_op1(curr.a1, ..., curr.ann));
                        // assert Pred2JML(POST_op1(curr.a1, ..., curr.ann)) && invariant_for(this);
                        // send a result back down the channel
                        curr.getChannel().write(result);
                        anyResumed = true;
                    } else {
                        operationIRequests.offer(curr);
                    }
                }
            }
        } // end for each list of requests
        // JML asserts that the CPREs for all the items left in all deferred requests lists are false
        while (anyResumed);
    } // end server loop
} // end run

Figure 4.22: Generic Server Component of the deferred requests template.

petitions. Usually, looping over the two lists of deferred requests will be enough to guarantee that all remaining requests cannot proceed. However, in some cases, there can be a 'ping pong' effect between operations, i.e. serving a request from one operation can enable a petition that was already checked (its CPRE did not hold with the state at that time).
4.4. Model-based Code Generation Using Java CSP implementation

```java
public static final int N_WAREHOUSES;
public static final int MAX_WEIGHT;
private int weight[];
private boolean occupied[];
...

public void enterWarehouse (int n, int w) {
    EnterWarehousePetition pet =
        new EnterWarehousePetition (n, w,
            Channel.one2one ());
    // sending petition to the server
    chEnterWarehouse.out ().write (pet);
    // waiting for notification
    pet.getChannel ().in ().read ();
}

public void exitWarehouse (int n, int w) {
    ExitWarehousePetition pet =
        new ExitWarehousePetition (n,
            Channel.one2one ());
    // sending petition to the server
    chExitWarehouse.out ().write (pet);
    // sending the weight
    pet.getChannel ().out ().write (w);
    // waiting for notification
    pet.getChannel ().in ().read ();
}
...
```

Figure 4.23: Warehouse implementation using the deferred requests technique using only two channels. Wrapper (client side) operations.

A possible scenario for a 'ping-pong' effect is when a robot Y is waiting for leaving warehouse 0 (exitWarehouse (0, w_Y)) and another robot X is waiting to enter to the warehouse 1 (petition enterWarehouse (1, w_X)) that means that corridor 0 is occupied, corridor[0] holds. When the server has already processed all exitWarehouse petitions (first loop), the robot Y is still blocked due to the fact that the next corridor is occupied by the robot X (corridor[0] holds). Then, the server continues its execution by processing all enterWarehouse invocations and, in particular, it processes the petition enterWarehouse (1, w_X) of the robot X, setting corridor[0] to false. After that second loop, the state of the server does not meet the second requirement of the three presented above for a correct implementation – because the CPRE of the petition of the robot Y holds and has not been processed. That is why, we have to enclose the two request processing loops in another loop (lines 99-174) that exits only when none of their CPREs are true (handled through the anyResumed flag). For verification (Section 3.2.1), asserts are added to that effect and are made following that enclosing loop.
private final List<EnterWarehousePetition> enterRequests = new ArrayList<>();
private final List<ExitWarehousePetition> exitRequests = new ArrayList<>();
...
private static final int ENTER = 0, EXIT = 1;

public void run() {
    /* One entry for each method. */
    Guard[] inputs = { chEnterWarehouse.in(), chExitWarehouse.in() };
    Alternative services = new Alternative(inputs);
    while (true) {
        int service = services.fairSelect();
        // synchronization not checked here!
        switch (service) {
            case ENTER:
                EnterWarehousePetition enterRequest = (EnterWarehousePetition) chEnterWarehouse.in().read();
                enterRequests.add(enterRequest);
                break;
            case EXIT:
                ExitWarehousePetition exitRequest = (ExitWarehousePetition) chExitWarehouse.in().read();
                exitRequests.add(exitRequest);
                break;
        }
        ...
        boolean anyResumed;
        do {
            anyResumed = false;
            int lastItem = exitRequests.size();
            for (int i = 0; i < lastItem; i++) {
                ...dequeue an ExitWarehousePetition from exitRequests
                if (curr.getWarehouse()==N_WAREHOUSES || !occupied[curr.getWarehouse() + 1]) {
                    //@ assert curr.getWarehouse()==N_WAREHOUSES || !occupied[curr.getWarehouse() + 1];
                    ...make POST of exitWarehouse hold
                    // send a null back down the channel
                    curr.getChannel().out().write(null);
                    anyResumed = true;
                } else {
                    ...enqueue curr back in exitRequests
                }
            }
        }
        lastItem = enterRequests.size();
        for (int i = 0; i < lastItem; i++) {
            ...dequeue an EnterWarehousePetition from enterRequests
            if (curr.getWeight()+weight[curr.getWarehouse()]<=MAX_WEIGHT) {
                //@ assert curr.getWeight()+weight[curr.getWarehouse()] <= MAX_WEIGHT;
                ...make POST of enterWarehouse hold
            } else {
                ...enqueue curr back in enterRequests
            }
        }
        ... JML assertion depicting all unprocessed requests CPRE does not hold
    }
}

Figure 4.24: Shipping Warehouse implementation using deferred requests.
Server side message processing.
5

Code Verification

It’s OK to figure out murder mysteries, but you shouldn’t need to figure out code. You should be able to read it.
—Steve McConnell

In the previous chapter, we focused on the generation of code by presenting a set of patterns for code generation with their respective proof obligations to make the obtained source verifiable. This is done due to the fact that applying those presented templates is not a fully automatic process. While the templates provide a syntactic structure, ensuring that the actual code fragments that “fill in” the holes in the template comply with the corresponding proof obligations is still the programmer’s responsibility. The goal of the verification process presented is to prove, mechanically, that the code obtained by application of some template – which we assume to adhere to its structure, syntactically – also complies with the proof obligations required. As we will show in the following paragraphs, doing this, even with the help of a mature verification environment for Java is a challenging task. However, with this “template-driven” approach to verification, we manage to replace a much harder verification problem – i.e. to fully verify the behaviour of the Java class against a transition semantics of the shared resource – by a set of smaller verification challenges, once per proof obligation.

For this task, we have chosen KeY [BHS07] because it supports Java code with JML annotations and is continuously maintained offering a solid verification framework that suites perfectly the purposes of this thesis. KeY is a theorem prover for Java Dynamic Logic based on a sequent calculus. It allows for full functional verification of sequential Java (without considering floats variables, garbage collection processes or multithreading code) and programs written in Java Card 2.2.x. It enables proof correctness of a given behavior specification annotated in JML (or could be used Java Dynamic Logic directly), using contract based techniques for verification. It contains pure Java expressions as a sublanguage and is therefore easy to learn for Java programmers. Its main extension beyond Java are quantified expressions (\(\forall T \ x, \exists T \ x\)) and, of course, suitable keywords for contracts. JML specifications are attached to the Java declarations they belong to in .java files. These specifications include precondition \(\text{PRE} \ (\text{requires JML clause})\), postcondition \(\text{POST} \ (\text{ensures JML clause})\) and method modifiers \(\text{assignable} \ (\text{clause})\). KeY tries to prove that the execution of an operation, in a state in which class invariant \(I\) and the precondition \(\text{PRE}\) holds, will terminate normally and in a state in which the postcondition holds.

To accomplish the verification, we assume that the code under analysis adheres, at least, syntactically to the structure described in the previous chapter (Chapter 4). This code is expected to contain logical (JML) annotations at certain points determined by the template logic (for instance, ln. 37 and 43 of Figure 4.18). Then, the code (Java and JML) needs to

\footnote{Other systems examined – and discarded for various reasons – were Slam [Bal+04], Bogor [RDH03] and TACO [Gal+10].}
be instrumented for KeY to be able to reason about the concurrent interplay between the processes and the shared resource. This instrumentation helps KeY figure out what happens behind the scenes with the specific concurrent mechanisms used in the implementations (such as channels, priority monitors, etc.).

Our key goal of this verification is proving that requests processing code (no matter what template is chosen) is sound and effective, and that the code can be verified by using sequential program provers.

The goal of this verification process is just to establish the conformance of the implementation with the template applied and, consequently, that the implementation behaves as expected with respect to the state transition semantics of the shared resource specification. The correctness of the whole system needs to be established in some other way, for instance by reasoning about the transition system generated by clients and shared resources – e.g. using model checking techniques, as shown in Chapter 3 and in [Her+09]. This system level validation is, however, language-independent, what means that dealing with code issues is circumscribed to the first part of the problem. This is specially convenient given that tools for reasoning about state machines are very powerful, while support for program verification is rather unequal and often restricted to small (and sequential) subsets of certain programming languages.

The following sections present the verification process using a common structure for all the presented templates. First, we enumerate the proof obligations that the templates impose at relevant points of the program for both monitor (Section 4.3) and CSP-based implementations (Section 4.4). Then, we present how to instrument the generated code in order to make it verifiable by KeY. Finally, we give details on actually proving the properties in KeY.

5.1 Verification of Monitor-Based Implementations

The starting point of our framework is a set of proof obligations that are imposed by the resource interface (the specification) and the code template (the implementation scheme) at certain places. These are:

1. The preconditions (PRE) are often independent for state changes in the resource (e.g. Notify_Weight method in Figure A.28). In such cases, they should be checked at call time and throw the corresponding exception although for brevity, we have not included that in the example. However, they must be assumed before the relevant operations.

2. The invariant (INVAR) must have been established at object initialization time, must hold right before entering the core block (S), right before finishing method body execution, but also right before invoking wait().

3. The concurrent or synchronization pre-condition (CPRE) must hold right before entering the core for each operation.

4. The postcondition (POST) must hold on return of each operation.

5. Whenever a signal() is invoked on a condition q, the safety assertion for that condition W_q can be established (soundness of signals).

6. If some thread has performed its operation and there is one (or more) thread blocked on some condition q such that its CPRE holds, then one of these threads must be signaled (completeness of signals).

7. At most one signal() can be issued during the execution of a monitor operation.

Properties 1–4 have to do with the observable behaviour of the shared resource, i.e. the service offered to the client threads and are, then, independent of the implementation method chosen. Properties 5–7 in this list, however, are specific to the implementation method (priority monitors in this case) and will be the ones we will pay more attention in the following paragraphs.

\(^2\)In terms of a shared resource model, being in a deadlock means reaching a state in which no CPRE holds in the current shared resource inner state.
5.1. Verification of Monitor-Based Implementations

![Code Example]

```java
public class MonitorKey {

  protected /*@ spec_public @*/ boolean mutex;

  /*@ public normal_behaviour
  @ assignable mutex;
  @ ensures !this.mutex;
  @*/

  public MonitorKey() {
    this.mutex = false;
  }

  /*@ public normal_behaviour
  @ requires !this.mutex;
  @ assignable mutex;
  @ ensures this.mutex;
  @*/

  public void enter() {
    this.mutex = true;
  }

  /*@ public normal_behaviour
  @ requires this.mutex;
  @ assignable mutex;
  @ ensures !this.mutex;
  @*/

  public void leave() {
    mutex = false;
  }

  /* Returns a new condition associated to this monitor. */
  public Cond newCond() {
    return new Cond();
  }

  /* Conditions. */
  public class Cond {
...
  }
}
```

**Figure 5.1:** Priority Monitor instrumented code to be recognizable and verifiable in KeY.

### 5.1.1 Instrumentation

As KeY does not offer support for concurrency or monitor primitives, we had to *instrument* our monitor library so as to provide KeY with a basic understanding of what is happening behind the scenes. For example, each condition is substituted by an `int` variable which will count the number of waiting threads on it, and therefore the signal operation just decreases that variable, etc. The instrumented monitor code obtained is presented in Figures 5.1 and 5.2. Notice that the code is simplified and the instrumented methods are those that are involved in the verification process. After that, each property described below can be easily expressed as `ensure` clauses as is presented below.

**Soundness**  (prop cs preservation) Conditional synchronization (CPRE formula) of a method holds immediately before executing the method code. To complete the verification, we need to prove that when a method is blocked, it will be enqueue in the corresponding condition queue based on the equivalence class which belongs to as it was shown in Section 4.3. The
public class Cond {
    private /*@ spec_public @*/ int waiting;
    //@ public instance invariant waiting >= 0;
    //@ accessible \inv: waiting;
    private Cond() {
        waiting = 0;
    }
    /*@ public normal_behaviour
    @ assignable waiting;
    @ ensures waiting == \old(waiting) + 1;
    @*/
    public void await() {
        waiting++;
    }
    /*@ public normal_behaviour
    @ requires waiting > 0;
    @ assignable waiting;
    @ ensures waiting == \old(waiting) - 1;
    @*/
    public void signal() {
        waiting--;
    }
    /*@ public normal_behaviour
    //@ ensures \result == waiting;
    public /*@ pure @*/ int waiting() {
        return waiting;
    }
}

Figure 5.2: Condition instrumented code to be recognizable and verifiable in KeY.

main difficulty in verifying this property comes from the issue that KeY cannot find any relation between the assertion of line 20 (Figure 4.12) and the program point immediately after the thread is unblocked (line 52).

Due to that, we rely on the following facts: (i) all signals are safe, (ii) only one signal is executed and (iii) the semantic of exclusion of priority monitors, i.e. there is no possibility of a spare thread reaching the monitor between executions. Hence, the proof is trivial without needing to perform any extra instrumentation.

Post-condition (prop_signal_and_exit) Even though a signal is executed or not, all variable representing the inner state of the shared resources must not change and preserves the Post-condition of the operation. That can be easily described by adding an assignable clause (ls.56-57) containing only the instrumentation variables.

The goal of this property is to avoid side effects on the unblocking code. Another approach to ensure this property would be to directly implement a signal and return [How76] policy. Such policy would suit perfectly since it also helps in simplifying the verification of prop_01_signal.

Soundness of Signals (prop_safe_signal) The unblocking code must guarantee that when a signal is executed, the conditional synchronization of the awakened thread holds. This property ensures safety of the unblocking code present in each method. As can be seen in the unblocking code, we define an assert for each operation’s CPRE as is shown in Figure 4.12.

Since KeY cannot easily interpret JML assertions, we define a JML ghost field. A ghost field is similar to a JML model field, in that it is also only present for purposes of specification and thus cannot be used outside of JML annotations. However, unlike a model field, a ghost field does not have a value determined by a represents clause; instead its value is
directly determined by its initialization or by a set-statement. called `awakenThread` that will contain the position of the array of the signaled condition queue. Initially, it must be -1 and then updated if a signal was performed. Besides, we add an `ensure` clause depicting that if a signal is executed, the synchronization condition of the thread must hold as is shown in lines 55-68 of the appendix C.2.2.

This clause expresses that if a condition queue decreases its size, i.e a `signal()` is executed, then the associated CPRE of the method of the signaled thread must hold.

The aim of this property is to ensure that every time a thread is awakened, it is able to proceed with the execution, i.e. its associated synchronization condition holds. The errors detected by this property are associated with a bad definition of the conditions array or an improper unblocking schema.

Completeness of Signals This is implemented using two properties in KeY:

- **(prop_signal_effective)**
  The executed `signal` (if any) in the unblocking code is effective, i.e. it actually wakes a blocked thread up. This is visible by a decrement in size of the condition queue. It is expressed as the `ensure` clause in lines 94-106 of the appendix C.2.2.

- **(prop_liveness)**
  The current thread must not leave the monitor without signaling one `improperly blocked` thread, if it exists. Improperly blocked threads are those blocked in a condition queue when its CPRE already holds. This property checks the existence of an illegally blocked thread. For every condition queue, if it is greater than 0 and its associated method’s synchronization condition holds, a `signal` must be executed and `signaled` variable must have the value 1. The resulting `ensure` clause is shown in lines 108-126 of the appendix C.2.2.

The aim of the property is to avoid having improperly blocked threads without being signaled after the execution of the unblocking code.

**At most 1 signal** (prop_0_1_signal) Only one thread can be awaken in each execution. The unblocking code must guarantee this in order to forbid that several threads access to the critical section of the given shared resource at the same time.

Since we have defined `signaled` variable and we must accomplish “0-1 coding scheme”, the property `signaled >= 0 && signaled <= 1` is added as an instance invariant. Besides, it is mandatory to ensure that one and only one condition queue is decremented, unless any thread was awakened. In that case, we must verify that all condition queues are keeping their size by adding a JML ensure clause depicted in lines 70-92 expressing that only one condition queue decreases in size.

The goal of having this property is the detection of badly nested conditional or iterative sentences, allowing several executions of `signal` operation. A relevant piece of instrumented code of `WarehouseAccessControlMonitorNaive` class is shown in appendix C.2.2.

## 5.2 Verification of CSP-Based Implementations

### 5.2.1 Channel Replication

The implementation of the Warehouse resource enable the server to determine whether the relevant CPRES hold for all the operation request channels being serviced, before taking any request. Hence, requests for service are blocked (if the CPRE is false) or accepted and immediately processed (if the CPRE is true). This is not the case for the deferred request technique and verification of such implementations is presented in Section 5.2.2.

This subsection considers verification of servers that can block currently ‘unserviceable’ requests or process serviceable ones immediately. In the code fragments illustrating this, we show the discovery of the particular request selected for servicing via Java’s `switch/case`
structure (used by the Readers/Writers example), rather than if/else (used by the Warehouse server via channel replication).  

**Instrumentation** As we did for monitors, we proceed to instrument the behaviour of the concurrent mechanism, such as the channels. They are translated into either an int variable representing the length of the message queue (when the contents of the request are immaterial for the subsequent verification) or into a List (when some operation makes use of the request contents and that makes it necessary to keep a sequence of request blueprints so as to verify the correctness of the code implementing that operation.). Moreover, we need to simulate the behavior of fairSelect via the homonym method as is shown in Appendix C.1. This tells us that we are able to select a channel if and only if there is a message waiting on it and its associated synchronization condition holds.

An advantage of KeY is the possibility of using of contracts (with the correct Method treatment option) that make simpler to simulate JCSP method fairSelect. To fully express its behavior, when there is no “selectable” channel (shared resource is in a state in which no CPRE holds), -1 is returned to halt the server loop. Because of this definition, the server loop condition is changed to iterate until there is no request that can be processed, as shown in Appendix C.1.

For the purposes of KeY verification, it does not matter that this adaptation of the server shuts down (unlike the actual servers in Figures 4.18 and 4.19).

As we did for the monitor-based implementations, we start by proposing a set of proof obligations for the server component of our code generation template using channel replication technique. Then, we show how to instrument them in order to be verifiable in KeY:

3. the CPRE must hold right before reading the request from the operation associated channel (soundness of the server);

5. the selection of the channel to read from must be open (its associated syncCond position must hold) and it must have a message waiting to be read (correctness of the server);

6. only one request can be processed in each server iteration (completeness of the server).

**Soundness of Server Component** (prop_cs_preservation) Immediately after the conditional statement that decides upon the index that tells the server which call must serve, the CPRE of that call must hold. As can be seen in Figure 4.19, it is asserted that CPRE holds just before updating the shared resource (line 93 for enterWarehouse operation and line 99 for exitWarehouse).

```java
switch (chosenService) {
    ....
    case METHOD_l:
        //@assert CPRE_method_l;
        ....
        break;
    ...
    }
```

As was done before, a new class instance attribute, called cprePreservation, is created and initialized to true that will express this property. Every time a service is chosen, this accumulator variable is updated (conjunctively) with the associated CPRE. To be verifiable in KeY we added as an invariant of the server loop as well as an ensure clauses.

This expresses that when a method invocation is selected, processing of the request will not violate the CPRE of the method, i.e. no request is served in a state that does not satisfy its synchronization condition. Once the server knows which branch of the conditional sentence will be executed, cprePreservation is updated before processing the request as it is said before.

3Modifying this generic template for the if/else structure is straightforward.
5.2. Verification of CSP-Based Implementations

Correctness of Server (prop_safe_selection) The server code must guarantee that a valid service is selected in each iteration, i.e. the selected service must belong to pe range, and has a message waiting to be read. This property is strictly related to syncCond array structure because fairSelect will chose a service depending on the available channels expressed by the aforementioned array.

The aim of this property is to ensure that the services and syncCond arrays are well-formed and consistent to each other: (i) services must include all input channels and its length must be equal to #rg(pe); (ii) a channel associated with a position i in services must have its synchronization predicate in the same position of syncCond and (iii) their length must be the equal.

To make the assertions generated by the template (after updating syncCond array) verifiable in KeY, a new class instance boolean attribute wellFormedSyncCond is defined in order to check that each position of the array is consistent.

It is important to check that every time syncCond is updated the resulting array is consistent. The aim of this verification is to detect when this re-evaluation is incorrect/incomplete and could allow the processing of a method invocation whose CPRE does not hold.

Completeness of the Server (prop_only_one_request) This property represents that exactly one request is processed per server iteration. This is already guaranteed if using nested if statements, but when using a switch, the execution of more than one branch is possible. The instrumentation for this property is rather simple: a counter variable processedMessages and a boolean variable oneMessageProcessed are created. The attribute complements the previous variable by contrasting it against 1 to express that only one request is processed at the end of each iteration. Finally, an ensure clause is defined to express the truth of the aforementioned attribute.

The goal of having this property is the detection of missing break statements in each switch pattern.

5.2.2 Deferred Requests

Instrumentation In this case, instead of verifying the whole server code, we focus only on the fragment in charge of processing all deferred requests. A method encapsulating that portion of code is created and annotated with JML ensures clauses as before. There is no need to present any further instrumentation as there is no JCSP mechanism that impacts any variable or the program flow of control.

In order to make the verification more efficient, the structure of deferred requests for each method is translated into an int variable representing its size, when CPRE and method do not depend on the input parameters. Thus, a dequeue of a requests is represented by a decrement of the given variable. Otherwise, a List is used as we did for the previous template.

The set of proof obligations for verifying the generated code based on deferred requests are:

3b. CPRE must hold immediately after the server code retrieves a requests to be processed (correctness of the server)

6b. If the server exits the code for processing deferred requests - and is about to loop back to the fairSelect - there should be no valid pending requests (completeness).

Correctness of the server (prop_cs_preservation) Immediately after the server code retrieves a request to be processed, the CPRE of the method associated with the request must hold. This is established with a JML assert before updating or accessing the resource – see Figure 4.24 (line 142 for the enterWarehouse operation and 109 for exitWarehouse). This restriction ensures safety of the processing code because changes to the state of the resources are performed only for those requests that represent valid invocations.

The verification of this property is extremely related with the homonym one from the previous template. To verify that this is always honored, a new class attribute is created and initialized to true and anded with the relevant CPRE each time an operation is processed –
Chapter 5. Code Verification

hopefully remaining true to confirm compliance. The ensures clause verifies whether this is the case. The server then loops back to await further operation requests (as in Figure 4.24).

Completeness of the Server Component (prop_completeness) If the server exits the code for processing deferred requests – and is about to loop back to the fairSelect – there should be no valid pending requests. In Figure 4.24 (immediately after line 174), we assert that for every request in the data structure (i.e. pending), their associated CPRE does not hold.

A request is either processed (if its CPRE holds) or enqueued again. If the former is true, the correctness of the server guarantees that is going to be processed. Otherwise, when the predicate does not hold, two cases can be distinguished. The first case, when the CPRE does not depend on the input parameters, can be described using the following ensure clause:

```c
//@ ensures \forall i (method\_iRequests > 0 \implies \neg CPRE\_i);
```

expressing that requests exist that cannot be processed and their associated condition does not hold. Notice that if the CPRE does not hold for a request, it will not hold for the rest of them due to the fact that the predicate only depends on the inner state of the resource.

However, when the CPRE depends on input parameters, we need to follow a similar approach as we did for the property prop_cs_preservation. A new variable is defined, completeness, and accumulates the value of the associated CPRE of requests. Finally, we conclude by ensuring that if there are still some requests to be processed, this variable must hold true:

```c
/*@ ensures \sum_{i=1}^{n} method\_iRequest.size () > 0 ==> completeness;
@*/
```

Notice that completeness encapsulates all unprocessed requests and, hence, there is no need to create one variable per method.

In this case, the errors detected are related with the associated condition for treating each request. In several examples, we can present a stronger condition than the required one, leaving messages without being processed. Moreover, errors related to the ping-pong effect can be detected due to the fact that it also leaves processable requests enqueued.

5.3 Preliminary Experimental results

In order to show that errors produced by an improper application of the schemata presented in the previous section can be detected by existing program verification tools, we have tested our approach on a number of examples.

A (small) set of (non trivial) problems have been considered. These include the readers and writers that we have used as running example as well as the warehouse example, some relatively small examples such as a shared resource specification of counting semaphores, a car park, and an airport traffic control tower that controls landing and taking off in several runways. The growing set of examples, including specifications, documentation and (verified) code is accessible at [AMH14]. Configuration files for reproducing the examples in KeY are included.

For each example, we have tested both code generation techniques on correct and buggy implementations:

- Correct implementations of all approaches
  - implementations following the templates in the most straightforward way, i.e. with no clever optimization.
  - optimized versions of the previous implementations. In the case of solutions based on the monitors, a better usage of conditions queues or an intelligent un-blocking code. Considering the channel replication template, most of these optimizations are related to avoiding unnecessary recomputations of the syncCond
5.3 Preliminary Experimental results

array. Analogously, for deferred requests, optimizations often have to do with the data structure used to store and access the pending requests.

- Erroneous/buggy implementations
  - erroneous optimizations,
  - violations of protocol definitions.
  - erroneous or incomplete update of the syncCond array for those implementations based on channel replication.
  - missing break statements in switch statement for JCSP-based code;
  - incorrect optimizations on the code processing the pending requests, e.g. leaving requests unattended, or too strong conditions on these requests when using JCSP deferred requests technique;

We have successfully verified in KeY all the correct implementations of the first kind and some of their optimized versions. However, on those implementations in which KeY stops the execution (some goals remain open), the tool gives back the control to the user in order to interact manually. That is, when KeY stops, it may be because the property under study is false or – a false negative – because the property holds but the theorem prover needs some assistance to establish it. In this scenario, the user can ask for a counterexample that leads to that unclosed goal – if the required prover is installed and linked, KeY will reconstruct it from the reached formula – or, alternatively, it can try to figure out whether the formula presented is valid or not, or some rule can be applied so that the proof continues.

5.3.1 Monitor-based implementations

We have compared three implementations of the warehouse problem:

- **WarehouseAccessControlMonitorNaive** class is a naive application of the template, thus, no optimization is made in the code. In this implementation, we applied the parameter indexation technique for both operations. For enterWarehouse operation, a \( N_{\text{WAREHOUSES}} \times \text{MAX}_\text{WEIGHT} + 1 \) matrix of condition queues is created and for exitWarehouse suffices with an array of \( N_{\text{WAREHOUSES}} \) condition queues. The unblocking code simply loops over the whole matrix and array trying to find an illegally blocked thread, i.e a blocked thread which CPRE does hold.

- **WarehouseAccessControlMonitorOpt** class keeps using parameter indexation for the condition queues but has an intelligent unblocking code. Every time a thread finishes executing enterWarehouse \((n,w)\), it will try to wake up either an enterWarehouse \((0,w)\) blocked invocation or an exitWarehouse \((n-1,w)\) call (when \(n > 0\)). Considering exitWarehouse, the code will wake up a thread that is either waiting to enter the given warehouse or those that are waiting to enter the first warehouse. Figure 5.3 shows an automaton expressing the heuristic of the unblocking code.

- **WarehouseAccessControlMonitorBestOpt** class is an improvement of the previous one. For exitWarehouse operation, a simple array is still used. But, instead of using a matrix of condition queues for enterWarehouse invocations, we define simpler structures: (i) when \(n > 0\), it suffices to declare an array of \( N_{\text{WAREHOUSES}} \) of WeightedConditions, i.e condition queues that are associated with a determined weight; (ii) when \(n = 0\), thread indexation technique is applied by using a PriorityQueue of WeightedCondition to gathers all blocked threads. Notice that in both cases, we need to store the weight of the call in order to check the CPRE before signaling the thread. The unblocking code differs slightly from the second implementation but follows the same principles expressed in Figure 5.3.

After following the proof obligations and the rules for instrumenting the monitor library stated in Section 5, we obtained the instrumented code for the naive implementation as shown in Appendix C.2.2.
Chapter 5. Code Verification

As it can be seen in the first implementation, the unblocking code iterates over all the defined condition queues. In each iteration, every blocked thread is checked against its associated synchronization condition even if it is already known that the associated condition does not hold. Then and only then, the code signals the correct condition queue. Having codified those clauses, the verification is possible in KeY and ends successfully, meaning that our naive unblocking code is correct.

Even though the solution is correct and satisfies the specification of the problem, this piece of code can be easily optimized. An invocation of enterWarehouse \((n,w)\) can signal an enterWarehouse \((0,w)\) invocation (when \(w=0\)) or exitWarehouse \((n-1,w)\) (when \(w>0\)). However, an exitWarehouse \((n,w)\) can wake up either an enterWarehouse \((0,w)\) call or enterWarehouse \((n-1,w)\) when \(n>0\). Considering the previous description, we obtained the instrumented unlocking code for enterWarehouse operation shown in Appendix C.4.2. Notice that the optimized code only iterates over those thread in which the associated weight will not exceed the constraint of the warehouse without checking directly its associated predicate (it is implicit by the starting point and invariant of the loop).

Both the elimination of one of the loops and the improvement of the other are nontrivial modifications and require the analysis of all possible scenarios. The KeY verification can detect errors easily due to the fact that we know beforehand that all ensures clauses of the previous implementation are correct and remain unmodified. If the prover stops, there must be some error in the optimization and the code needs to be revised. Otherwise, we are able to set the correctness of the performed optimization.

Once the second implementation is verified correctly and no errors were found, an optimization of the condition queues is made as described before. As a consequence of the redefinition of them, it is mandatory to rewrite all the ensures clauses. Once we have done all the modifications, we are able to verify the unblocking code in KeY by checking if the new structures satisfy the proof obligations depicted in Chapter 5. Finally, after carrying out all the verification process, we can conclude that the fully optimized version is correct.

5.3.2 JCSP-based implementations

Regarding the templates using JCSP, we only analyzed the code generated using the replication channel technique\(^5\). For that, we have conducted the same experiment over the following implementations using JCSP of the warehouse problem:

- WarehouseAccessControlMonitorCSP class is a naive application of the template without performing any optimization. For enterWarehouse operation, a \(N\_WAREHOUSES \times MAX\_WEIGHT + 1\) matrix of channels is created and for exitWarehouse suffices with an array of \(N\_WAREHOUSES\) channels. syncCond array is fully updated in each server iteration.

- WarehouseAccessControlMonitorCSPOpt class keeps the same principles as the former but the syncCond array is updated on those positions regarding the warehouse involved in the previous iteration. For instance, only the positions in correspondence

\(^4\)Notice that the unblocking code is now separate in two different pieces, one per each iteration. The code for exitWarehouse operation is analogous and can be seen in [AMH14].

\(^5\)The experimentation over those implementations that used ‘deferred request’ techniques are not shown in this thesis but downloadable from [AMH14].
5.3. Preliminary Experimental results

with row \( n \) of \( \text{chEnterWarehouse} \) and \( \text{chExitWarehouse}[n] \) are updated if the last processed operation was \( \text{enterWarehouse} \ (n,w) \).

As it can be seen at the beginning of the server loop in the first implementation, \( \text{syncCond} \leftarrow \) is fully updated, i.e. each position of the array is recalculated even if it is already known that a specific position does not change. Then, having codified the clauses explained in Section 5.2.2, the verification is possible in KeY and ends successfully, meaning that our simplest code is correct.

Even though the solution is correct and satisfies the specification of the problem, the server component can be easily optimized to gain efficiency (e.g., the portion of code in charge of updating \( \text{syncCond} \) array).

After an invocation of \( \text{enterWarehouse} \ (n,w) \), the channels associated with any other warehouse but \( n \) remains in the same state (open or close) than before. That is, either the weight of the warehouse \( n \) has changed or the corridor \( n-1 \) (if \( n > 0 \)) is set free. Therefore, only those positions associated with the given warehouse (of both operations) must be updated 6. Notice that all instrumentation regarding the proof obligations described in Section 5.2.1 remains unmodified, i.e. the variable \( \text{wellFormedSyncCond} \) should check that all positions of the array are consistent.

In some cases, improving the update code for \( \text{syncCond} \) is not trivial and require the analysis of all possible scenarios based on the previous processed request. The KeY verification can detect errors easily due to the fact that we know beforehand that all clauses of the previous implementation are correct and remain unmodified, as we did for the monitor-based template. If the prover stops, there must be some error in the optimization and the code needs to be revised. Otherwise, we are able to set the correctness of the performed optimization.

Table 5.1 summarizes part of the results of our experiment showing whether the verification was successful or not, which template was applied and how detailed is the output of KeY. This table is focused only on three examples to illustrate the most common errors that can be introduced by the user in a trivial and in a more complex example. This can be extended to all the shared resource implementations listed in \[AMH14\].

The first column tell us which problem is implemented followed by the third column which gives a short description of the code. The legend in the second column indicates which template was applied to the implementation. We add \( \text{MON} \) for those using priority monitors and \( \text{DF} \) and \( \text{CR} \) for the implementations based on JCSP library following ‘deferred requests’ or channel ‘replication technique’, respectively. The last column illustrate whether KeY was able to closed all the goals or not. Besides, we added if the unclosed goals or the proof tree give us enough information to retrieve the user which is the issue with the given implementation.

When trying to verify \( \text{ReadersWritersCSPDeferredRequestBuggyKeY} \) on KeY, the tool stops with the unclosed goal shown in Figure 5.4. As can be seen, there are equations constraining almost all the relevant variables in the code that are rather illegible. To clarify the output, we have highlighted in boxes two pieces of the predicates, \( \text{self.readers} \leftarrow 0 \) and \( \text{self.afterReadRequests} = 1 \). These lines describe a scenario in which there is a request of \( \text{afterRead} \) (upper box) to be processed but the set of processes reading in the system is empty (lower box) leading to a clear violation of the class invariant (the \( \text{readers} \) variable would be decremented taking the value \( -1 \)). This is an example of an easily read unclosed goal.

However, it is not always straightforward to reconstruct the counterexample or even to detect the problem for the unreadable formulas. As can be seen in Figure 5.5, there is no concrete information about the state of the shared resource at all (just a couple of formulas expressing that the core variables might have been \( \text{null} \)). But when reading the proof tree, the error easily shows up.

---

6The analysis of the other scenario, when a \( \text{exitWarehouse} \) request is processed, is analogous to the presented one.
<table>
<thead>
<tr>
<th>Example</th>
<th>Short description of the implementation</th>
<th>KeY Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Readers &amp; Writers MON</td>
<td>correct implementation with readers priority</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Readers &amp; Writers MON</td>
<td>WRONG implementation: a reader and a writer could proceed at the same time</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(readers == 1 &amp; writers == 1)</td>
</tr>
<tr>
<td>Readers &amp; Writers MON</td>
<td>WRONG implementation: two readers can be awakened</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(signaled == 2)</td>
</tr>
<tr>
<td>Multibuffer MON</td>
<td>correct implementation</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Multibuffer MON</td>
<td>WRONG implementation: allowing writing beyond the capacity of the buffer</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td>(bad CPRE codification)</td>
<td>incomprehensible goal</td>
</tr>
<tr>
<td>Shipping Warehouse MON</td>
<td>correct implementation</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Shipping Warehouse MON</td>
<td>WRONG implementation: unblocking code does not wake up robot with maximum</td>
<td>× error detected</td>
</tr>
<tr>
<td></td>
<td>weight</td>
<td></td>
</tr>
<tr>
<td>Readers &amp; Writers DF</td>
<td>correct implementation with readers priority</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Readers &amp; Writers CR</td>
<td>correct implementation with writers priority</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Readers &amp; Writers DF</td>
<td>WRONG implementation: formula for proceeding afterWrite and afterWrite is</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td>weaker than expected allowing to have a false positive</td>
<td>(readers == -1)</td>
</tr>
<tr>
<td>Readers &amp; Writers CR</td>
<td>WRONG implementation: missing break statement</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(oneMessageProcessed == false)</td>
</tr>
<tr>
<td>Shipping Warehouse CR</td>
<td>correct implementation with readers priority</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Shipping Warehouse DF</td>
<td>correct implementation with readers priority</td>
<td>✓ verified</td>
</tr>
<tr>
<td>Shipping Warehouse CR</td>
<td>WRONG implementation: missing break statement</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(oneMessageProcessed == false)</td>
</tr>
<tr>
<td>Shipping Warehouse CR</td>
<td>WRONG implementation: syncCond is not consistent</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td></td>
<td>(wellFormedSyncCond == false)</td>
</tr>
<tr>
<td>Shipping Warehouse CR</td>
<td>WRONG implementation: updates to syncCond are not complete enabling illegal</td>
<td>× error detected:</td>
</tr>
<tr>
<td></td>
<td>invocations to proceed</td>
<td>(cprePreservation == false)</td>
</tr>
</tbody>
</table>

Table 5.1: Summary of the results obtained when verifying several examples using KeY. Major attention is giving to detect which implementations have failures or bugs without considering the (finite and practical) time spent by the prover.
5.3. Preliminary Experimental results

**Figure 5.4:** Unclosed goal verifying ReadersWritersCSPDeferredRequestBuggyKey.java.

```java
self.afterReadRequests = 1;
self.beforeWriteRequests = 1,
wellFormed(heap),
self.<create> = TRUE,
es.upm.babel.ccml.samples.readerswriters.java.csp.ReadersWritersCSPDeferredR
self.readers = 0,
self.beforeWriteRequests = 0,
wellFormed(anon_heap.coreBeforeWrite1=anonHeapFunction>>),
heap1.self.corePreservation = TRUE
[anom1, anon_heap.coreBeforeWrite1=anonHeapFunction>>]
= heapAfter.coreBeforeWrite1,
exc_23 = null,
result_5 = TRUE,
wellFormed(anon_heap_innerBeforeWrite1=anonHeapFunction>>),
heapAfter.coreBeforeWrite1.self.corePreservation = TRUE
{self.beforeWriteRequests = 0}
[anom1 self,.
es.upm.babel.ccml.samples.readerswriters.java
anon_heap_innerBeforeWrite1=anonHeapFunction>>
= heapAfter_innerBeforeWrite1,
exc_24 = null,
self.writers[anon_heap_innerBeforeWrite1=anonHeapFunction>> = 1
}=>
self = null
```

**Figure 5.5:** Unclosed and unreadable goal verifying ReadersWritersCSPDeferredRequestBuggyKey.java.
6

Model Driven Testing of Shared Resources

Program testing can be a very effective way to show the presence of bugs, but is hopelessly inadequate for showing their absence.
—Edsger Dijkstra

Now, the last phase of the methodology is presented. There are different aspects of a system implemented using shared resources that we can test. We can for instance focus on testing the specification itself, to validate that the specification is internally consistent, and that it faithfully expresses the informal requirements an implemented system should satisfy. An example of a consistency property is that all postconditions should preserve the resource invariant.

Thus, this phase is be divided into two solutions that are iterative and inclusive. First, we focus on the task of verifying that an implemented system faithfully conforms to the resource specification on which it is based. Apart from testing the safety (CPRE, POST) aspects of the system under test, there are often implementation requirements on the order in which the implemented system services the calls whose concurrency preconditions hold. Secondly, we will consider the problem of testing a (well-formed) protocol guarded shared resource based system, composed of a set of processes, a set of shared resources, a set of environment items, and a set of associated protocols, written $\langle \prod_{i=1}^{n} p_i, \prod_{j=1}^{m} shr_j, \prod_{k=1}^{o} e_k \rangle \| \prod_{i=1}^{n} prot_i$.

6.1 Shared resources Semantics for testing

Let us consider an operational semantics for resource based systems. At first, just a single resource was considered but, then, the definition was extended by adding information about the execution environment. Let us start by defining some central concepts and sets. Those will help us to depict the form of a shared resource configuration. This and the resource step will define the semantics of a resource itself [Fre+16].

Definition 6.1.1 (Names). Let be the names of all actors involved in the system the following set

$\text{Names} = \text{Rids} \cup \text{Pids} \cup \text{Eids}$

where Rids are the resource identifiers, Pids identifies the name of the processes and Eids are the environment identifiers.

Definition 6.1.2 (Calls). Consider Calls as the set of all terms of the form $\text{call}(\text{to, from, op, args})$ that encodes an invocation from a process to the resource or an environment entity, where
to ∈ Rids ∪ Eids is the “target” of the call, a resource or an environment entity.

Note that the semantics does not permit direct process-to-process calls;

from ∈ Pids identifies the process that made the call/

op ∈ Operations name of the operation that was invoked;

args ∈ \( P(K) \) is a tuple of the actual arguments of the call.

Once the calls are depicted, it is mandatory to have a way of define the results of those invocations.

**Definition 6.1.3 (Returns).** A return is a value of the form \( \text{return}(\text{to}, \text{from}, \text{value}) \) which encodes the return values from a process call:

- \( \text{to} ∈ Rids ∪ Eids \) is the “target” of the call, a resource or an environment entity, which is returning a value to the source of the call;
- \( \text{from} ∈ Pids \) identifies the process that initiated the call, and which will receive the return value;
- \( \text{value} ∈ P(K) \) is the value being returned.

Notice that \( K \) is set of values from some specific domain strictly related to a shared resource. Then, an action is a combination of the previous definitions:

**Definition 6.1.4 (Action).** An action is either a call of the form \( \text{call}(\text{to}, \text{from}, \text{op}, \text{args}) \), a return value \( \text{return}(\text{to}, \text{from}, \text{value}) \), or an internal action on the form \( \tau(\text{name}, \text{args}) \) where \( \text{name} ∈ Names \) and \( \text{args} ∈ P(K) \).

Any invocation from a process can be seen in fact as a decomposition: each call is split into the activation of the call (a process is asking to execute \( \text{op} \)-calls) and the finalization of it (when the invoker received the confirmation as a return value).

Consider \( \alpha \) to range over infinite subset of actions, and let \( \alpha_{\text{comm}} \) range only over all call and return actions. Also, the auxiliary function \( \text{target} : \text{Actions} → (Rids ∪ Eids) \) is defined and it extracts the first component of an action tuple (projection)

Now that all the basic sets and definitions were presented, it is time to define the form of a shared resource component. Considering the inner state of a shared result as a set of correspondence between variables and values (\( \text{ResourcesState} ≡ (\text{Var}id × K) \), the definition below is obtained:

**Definition 6.1.5 (Shared Resource Configuration).** A shared resource configuration \( (shr) \) is an ordered six-tuple of the form

\[
⟨r-id, \text{ActiveCalls}, \text{SA}, s_0, ss, \text{Ops}⟩
\]

where:

- \( r-id ∈ \text{Rids} \) is the name unequivocally identifying the resource;
- \( \text{ActiveCalls} ∈ \mathcal{P}(\text{CInfo} × \text{Calls}) \) is a set of calls with true preconditions that have not been yet completed, in addition with scheduling control information generated by the WAITING clause;
- Notice that the set \( \text{CInfo} \) could be of any type strictly attached to the scheduling component. For instance, to represent a FIFO policy, suffice to have \( \text{CInfo} ≡ N \).

- \( \text{SA} ∈ \text{ResourceState} \) is the current internal state of the shared resource
- \( s_0 ∈ \text{SA} \) is the initial configuration of shared resource as defined by the INITIAL of the specification.
- \( ss ∈ \text{SchedulingState} \) is the schedule state. The exact implementation mechanism for delivering such a continuation permission is left to the user.
- \( \text{Ops} ⊆ \text{Operations} \) is set of operations of the shared resource.
6.1. Shared resources Semantics for testing

\[
\begin{align*}
\text{pre} & : \text{Calls} \rightarrow \mathbb{B} \\
\text{WAITING} & : (\text{Calls} \times \text{SchedulingState} \times \text{ResourceState}) \rightarrow \text{SchedulingInfo} \times \text{SchedulingState} \\
\text{ENABLED} & : (\text{Calls} \times \text{SchedulingInfo} \times \text{SchedulingState} \times \text{ResourceState}) \rightarrow \mathbb{B} \\
\text{post} & : (\text{Calls} \times \text{K} \times \text{ResourceState}) \rightarrow \text{ResourceState} \\
\text{RETURN} & : (\text{Calls} \times \text{ResourceState}) \rightarrow \mathcal{P}(\text{K}) \\
\text{POSTWAITING} & : (\text{Calls} \times \text{SchedulingInfo} \times \text{SchedulingState} \times \text{ResourceState}) \rightarrow \text{SchedulingState}
\end{align*}
\]

**Figure 6.1:** Operational semantics auxiliary functions.

Given a shared resource identifier, an auxiliary function \( \text{ops} : \text{Rid} \rightarrow \text{Operations} \) selects the set of operations names of the resource.

Once defined the configuration of a shared resource, a rule that replace a configuration by a new one is depicted in the following rule. If more than one action is eligible, there will be more than one possible transition from one state to the other. Even when more than one transition can be processed, the model process just one single action at a time by non-deterministically ordering the transitions or steps.

**Definition 6.1.6 (Resource Step).** A resource (execution) step is an item of \( \text{shr} \times \alpha \times \text{shr} \), written \( \text{shr} \xrightarrow{\alpha} \text{shr}' \), where \( \text{shr}, \text{shr}' \) are shared resource configurations, and \( \alpha \) is an action.

Each step will be illustrated as a operational semantics rule using the following format:

\[
\begin{align*}
c_1 \ldots c_n
\end{align*}
\]

\[
\begin{align*}
\langle r, \text{calls}, \text{sa}, \text{ss} \rangle \xrightarrow{\alpha} \langle r, \text{calls}', \text{sa}', \text{ss}' \rangle
\end{align*}
\]

indicating that a shared resource specification in the state \( \langle r, \text{calls}, \text{sa}, \text{ss} \rangle \) can transit into state \( \langle r, \text{calls}', \text{sa}', \text{ss}' \rangle \), provided that conditions \( c_1, \ldots, c_n \) hold. Notice that the initial state and the operations set are removed in order to simplify the rules.

The semantics is parametric over the concrete definition of six functions used to manipulate the scheduling state at different stages of a call being processed, and which are derived from a concrete shared resource specification such as for instance any of the specification depicted in Appendix A. The signatures of these functions are presented in Figure 6.1. \( \text{PRE} \) function will define if the precondition of a given call holds based on the operation and the arguments of the call. Based on the call, the current scheduling state and the inner state of the resource, \( \text{WAITING} \) will associate the scheduling information of the given call while preparing the scheduling component for the next call. \( \text{ENABLED} \) points out which call can be processed based of the associated scheduling information and scheduling component. The function \( \text{POST} \) is used to compute the next shared resource state, to be a \( \text{function} \) which depends on the call, the returned value from the operation, and the current resource state. However, \( \text{RETURN} \) is permitted to return a set of values. That is, a shared resource specification may specify that the value returned for a call is non-deterministic, but given that such a value is chosen, the computation of the next resource step must be deterministic. Finally, \( \text{POSTWAITING} \) will retrieve the next scheduling component state after a call being processed.

The \( \text{REJECT} \) rule triggers when a call does not satisfy its precondition. The current call is then dismissed. If the call satisfy the precondition, \( \text{CHECK-IN} \) rule is triggered. The scheduling component is updated and the call together with a “call information” data is stored in the set of calls. The third and final rule (\( \text{EXEC} \)) models the execution of a method call. Any received call which is enabled (i.e., \( \text{CPRE} \) holds), and which is enabled to execute according to the scheduler (i.e., \( \text{ENABLED} \) holds) can be executed. The result of the execution is that both the safety state is updated (through the \( \text{POST} \) rule) and the scheduling state is updated (through the \( \text{POSTWAITING} \) rule).

Note that far simpler operational semantics rules can be obtained if the scheduling of calls is not considered. The modified \( \text{EXEC} \) rule, for instance, then becomes as follow:
Chapter 6. Model Driven Testing of Shared Resources

To define a semantics for systems (of shared resources, of processes, or of environment entities) is presented as is shown below:

\[
\forall x \in \prod_i x_i \quad x \xrightarrow{\alpha} x' \quad \prod_i x_i \xrightarrow{\alpha} (\prod_i x_i)[x'/x]
\]  

Then, the composite operational semantics rules are defined in Figure 6.3.

In this rule, the execution of a call is simplified by just modifying the shared resource state according to the postcondition of the operation and the current call being dismissed from the set.

Now, let us start a definition of environment entities and processes. Considering \( p \) range over the set of processes, and let \( e \) range over the environment entities, the following definitions can be made. It is assume that there exists an operational semantics for processes with a computation step \( e \xrightarrow{\tau} e' \). and that there exists an operational semantics for environment entities with a computation step \( e \xrightarrow{\tau} e' \).

\textbf{Definition 6.1.7} (An operational semantics for systems). To define a semantics for systems first a semantics rules for sets (of shared resources, of processes, or of environment entities) is presented as is shown below:

\[
\forall x \in \prod_i x_i \quad x \xrightarrow{\alpha} x' \quad \prod_i x_i \xrightarrow{\alpha} (\prod_i x_i)[x'/x]
\]  

The semantics for systems is quite standard; internal computation steps (labelled by \( \tau \)) affect only the “component” executing it, while \( \text{calls} \) and \( \text{returns} \) require a joint action by a process and either a shared resource or an environment component.
Definition 6.1.8 (Trace of a process; action sets). A trace from a process \( p \) is a sequence of transitions \( p \xrightarrow{\alpha_0} p_1 \xrightarrow{\alpha_1} p_2 \xrightarrow{\alpha_2} \ldots \). Let the action set for a process \( p \) be the set of call and return actions occurring in any trace of \( p \), denoted \( \text{actset}(p) \).

Definition 6.1.9 (Well-formed systems). A well-formed system is composed of a set of resources (\( \prod_i \text{shr}_i \)), a set of processes (\( \prod_i \text{prot}_i \)), and a set of environment entities (\( \prod_i \text{E}_i \)), written \( \langle \prod_i \text{prot}_i, \prod_i \text{shr}_i, \prod_i \text{E}_i \rangle \) (also written \( \langle R, P, E \rangle \)), such that all shared resources, processes and environment entities have unique names.

Next, it is mandatory to formalize the notion of a protocol for a given process in order to determine and validate all traces.

Definition 6.1.10 (A process protocol). Given a process \( p \), a protocol for that process is an automaton \( \text{prot} \) with states \( Q \cup \{\text{fail}\} \), and action set \( \text{actset}(p) \), such that

\[
\forall q \in Q \cdot \forall \alpha_{\text{comm}} \in \text{actset}(p) \cdot \exists q' \in Q \cup \{\text{fail}\} \cdot q \xrightarrow{\alpha_{\text{comm}}} q'
\]

and \( \text{fail} \) has the single transition \( q \xrightarrow{\tau(\text{fail})} \text{fail} \).

The definition ensures that the protocol can run in lockstep with the associated process up to some failure point indicated by the \( \text{fail} \) state, after which the only action possible by the protocol is the \( \tau(\text{fail}) \) action indicating a protocol failure.

Definition 6.1.11 (A protocol guarded system). Given a system \( \langle \prod_i \text{prot}_i, R, E \rangle \) with processes \( p_1, \ldots, p_n \), a protocol guarded system is a well-formed system of the form \( \langle \prod_i \text{prot}_i \rangle || \prod_i \text{prot}_i \rangle \), where \( \text{prot}_i \) is a protocol for process \( p_i \).

Once the formal definition of a protocol is given, an operational semantics is developed for protocol guarded systems, ensuring that processes and protocols execute in lockstep. Note that the definition permits internal actions by processes, shared resources, and environment entities, as long as the protocol has not reached its fail state.

Definition 6.1.12 (An operational semantics for protocol guarded systems). Let the following be the operational semantics rules for protocol guarded systems:

- \( (P, R, E) \xrightarrow{\alpha_{\text{comm}}} (P', R', E') \):
  - \( (P, R, E) \parallel \text{Prot} \xrightarrow{\alpha_{\text{comm}}} (P', R', E') \parallel \text{Prot'} \) (LOCKSTEP)
  - \( (P, R, E) \parallel \text{Prot} \xrightarrow{\tau(\text{fail})} (P, R, E) \parallel \text{Prot} \) (FAIL)

Considering our running example, the shipping warehouse problem, there is just a single requirement on servicing calls, to enforce progress:

*if the set of calls with true concurrency preconditions is non-empty, the system must eventually select a call to execute.*

We can illustrate the semantics of this requirement by an example, assuming that the maximum weight permitted in warehouse 0 is 1000kg:

- \( \text{enterWarehouse} (0, 900) \)
- \( \text{enterWarehouse} (0, 200) \)
- \( \text{enterWarehouse} (0, 100) \)

We assume that calls are made sequentially (meeting a linearizability criteria). The first call does not block, as \( 900 \leq 1000 \). The second call blocks, as \( 900 + 200 > 1000 \). The third call is permitted by the concurrency precondition as \( 900 + 100 \leq 1000 \), and thus cannot be blocked for infinitely long. Such requirements are expressed formally by associating a scheduling specification with the system under test.

We will test a shared resource by developing a model for the behaviour of the resource as a Quviq QuickCheck [Art+06] state machine. In the following we assume that the system is
implemented using Java, although this is not crucial to the approach\(^1\). A first question to ask is what errors can we expect programmers to make when implementing a shared resource. The errors can be broadly separated into three classes:

**non-atomic:** calls are not evaluated atomically. That is, the evaluation of the concurrency preconditions and postconditions (or of scheduling checks) of different calls are interleaved, although the concurrency precondition and postcondition of a given call should be evaluated in sequence. These errors are likely due to basic misunderstandings with regards to using a particular concurrency feature in the implementation language. To find such errors concurrent calls to the shared resource must be issued.

**bad-spec-impl:** badly implemented CPRES or POSTs. To detect such errors issuing a sequence of sequential calls is sufficient.

**bad-sched:** bad implementation of scheduling requirements. That is, the programmer has made mistakes in the selection of a call candidate eligible to enter the resource; this can be a difficult task due to ordering constraints and the manner in which blocked tasks must be woken up. Correctly programming this functionality in e.g. Java is not an easy task [Mar+19], and we can expect to see many errors here.

To detect such errors we must be able to observe which pending calls were unblocked by the execution of a non-blocking call. That is, if the concurrency preconditions for all pending calls in a shared resource are false, and a new call \(\text{call}_1\) arrives whose precondition is true, we should observe which pending calls \(\text{call}_2, \ldots, \text{call}_n\) are unblocked due to the execution of \(\text{call}_1\).

### 6.1.1 Observational Power of Tests

An implementation of a shared resource will be tested by issuing, concurrently, a number of calls to the resource and observing which calls complete (and which remain blocked).

With regards to the semantics presented in Section 6.1, this corresponds to providing a sequence of calls as input (in the \(\text{in}\) component), and observing what calls complete, and in what order (i.e., noting the contents of the \(\text{out}\) component).

In the following it is assumed that implementations of shared resources cannot be instrumented in any way due to the fact that this technique is based on black-box testing. The only interactions possible with such an implementation is to create a new resource, and issue calls to it and observe whether a call completes.

Concretely, if calls are made concurrently, there is no possibility of observing either in which order calls arrive at the shared resource (i.e., the ordering of calls in \(\text{in}\)), nor of the order in which calls complete (i.e., the ordering of calls in \(\text{out}\)). Moreover, there is no failsafe method to observe when a call is blocked, i.e., because its concurrency precondition is false, or because it is not permitted to complete due to scheduling constraints, as we cannot distinguish a very slow but non-blocked call (e.g., because the computation of the postcondition is time consuming) from a truly blocked call.

To observe whether a call blocks then, a pragmatic approach is taken. A test is separated into a number of **test phases**, where the exact number of phases is chosen randomly. Each phase starts with issuing a number of concurrent calls to the shared resource. Then the testing code waits a sufficiently long time for all calls to complete, and calls which has not completed within the time limit are considered blocked. Then a new test phase begins, with issuing new concurrent calls to the resource, potentially unblocking old blocked calls too. Clearly this approach is unsound in general, if the completion of a call can take a unknown amount of time. However, for most systems there are requirements on the maximum amount of time spent on calls, as the I/O and time-consuming operations will be part of the client processes code, while the code implementing resources is assumed to be essentially pure and instantaneous. In practice the time spent waiting for calls to complete in a phase is a test parameter, which can be specified at the start of testing. Currently, all test phases use the same waiting time.

\(^1\) Any programming language is suitable for this task (even Erlang), the system under test only needs to be executable on Erlang. For instance, there are available many connectors for different languages like Java and C for instance.
As a conclusion, if we abstract away from the return values of calls (as is done for presentation purposes), the test code can judge the success of a test solely based on which calls complete, and which calls block during each test phase.

### 6.1.2 Implementing Tests

When testing a shared resource there are two essential tasks: (i) deciding which sequences of calls to issue to the shared resource, and (ii) deciding whether the execution of a set of calls was successful or not.

![Diagram of state transitions]

### 6.1.3 Deciding on the Contents of a Test

Although it would be possible to generate purely random calls to a shared resource under test this is often not desirable, as most such randomly generated calls would likely be rejected immediately by a call pre-condition, or would simply be nonsensical (some history might not be reachable). Instead of generating purely random calls we use a QuickCheck state machine to generate sensible calls, which are tailored to the particular shared resource under test. That is, the responsibility of dealing with the inherent incompleteness of test coverage is transferred (in part) to the test designer.

As explained earlier, a test case is composed of a number of test phases. The QuickCheck state machine for call generation issues in each test phase either a single call, or a small number of concurrent calls. As an example, Section 6.2.1 discusses how calls are generated for the robot case study.

### 6.1.4 Deciding on a Test Result

The decision of whether the execution of a test was successful is broken down into one decision for each test phase.

In the following example we will consider progressing implementations only, i.e., implementations that eventually execute a continuously enabled call, and moreover, we assume that the waiting time before executing an enabled call is sufficiently small, and its runtime sufficiently quick, so that the implementation will have always completed any call it plans to complete before the end of the current phase.

Using the operational semantics developed in Section 6.1 we can compute the possible completed states of each phase, which are consistent with the actions of the implementation, denoted with $\text{Viable}_i$ for phase $i$. Initially this is the set

$$\text{Viable}_0 \equiv \{ (\epsilon, \emptyset, sa_0, ss_0, \epsilon) \}$$

where $sa_0$ is the initial state of the shared resource (defined by the INITIAL rule), $ss_0$ is the initial state of its scheduling policy, and $\epsilon$ is the empty sequence.

We compute the set $\text{Viable}_{i+1}$ from $\text{Viable}_i$ by computing the completed states $\text{completed}(.)$ for any state in $\text{Viable}_i$ with as input any ordering of the calls in $\text{cmd}_{i+1}$. Moreover, the set $\text{Viable}_{i+1}$ is restricted to states such that the completed calls by the implementation $\text{finished}_{i+1}$ should coincide with the completed states predicted by the transition relation.

Formally then, we define $\text{Viable}_{n+1}$ as:
Chapter 6. Model Driven Testing of Shared Resources

Viable_{i+1} = \left\{ s \in \text{completed}(\langle in_{i+1}, calls_i, sa_i, ss_i, \epsilon \rangle) \right. \\
\text{such that} \\
\langle \epsilon, calls_i, sa_i, ss_i, out_i \rangle \in \text{Viable}_i \\
\left. \quad in_{i+1} \in \text{fac}(cmds_{i+1}) \right. \\
\left. \quad \text{to\_set(out(s))} = \text{to\_set(finished}_{i+1} \right\}

where \text{fac}(cmds_{i+1}) is the set of sequences constructable from the set of calls in \text{cmds}_{i+1}, and \text{to\_set(out}_{i+1} is the set containing all the calls in the sequence \text{out}_{i+1}.

Note that even with the restriction of a single call per phase, the above set can have a size greater than one, as the scheduling specification may well permit multiple calls to execute at any point in time. Consider for instance a resource with two methods \textit{b} and \textit{a}, and in the two first phases one call to \textit{b} is made in each, which block. Then in the third phase a call to \textit{a} is made, which unblocks one of the calls to \textit{b}, but the scheduling specification may well permit both to proceed.

However, note that in such a case, the \textit{out} components of the resulting states are guaranteed to be distinct, as the order in which the call complete will differ.

In our testing framework, however, the order in which completed calls occur in the \textit{out} component cannot be observed as explained in Section 6.1.1), and moreover, concurrent calls to the resource are permitted (leading to another race between applying operational semantics rule 2 and 3). This restriction on observational power has the effect that our testing framework must cope with an uncertainty regarding which is the real state of the tested resource (and its scheduler).

We can thus summarize the testing procedure, for phase \(i + 1\), given the viable states at phase \(i\), as follows:

1. Generate the calls \text{cmd}_{i+1} for phase \(i + 1\) and execute them concurrently.
2. Record which calls have been completed by the implementation at the end of the phase as \text{finished}_{i+1}.
3. Calculate the next set of viable states, \text{Viable}_{i+1}, according to the above definition (from \text{Viable}_i, \text{cmd}_{i+1}, and \text{finished}_{i+1}).
4. If the size of \text{Viable}_{i+1} is 0, signal testing failure, since there is no execution of the test model which can explain the results of executing the implementation.

The computation of the set of viable state is done in a lazy manner, taking care not to generate all potential states at once, but rather computing \text{completed}(s) in a stepwise manner, discarding failed intermediate states at once, and merging intermediate identical states as soon as possible, to improve analysis efficiency. Nevertheless, in the worst case there may be an exponential number of viable states to explore. To combat such situations we explore only tests of a limited size, and where the number of concurrent calls are severely limited by design.

6.2 Testing the Warehouse Resource

To test an implementation of the Warehouse resource using our prototype shared resource testing framework\(^\text{2}\), there are four essential tasks to complete:

- Coding the warehouse shared resource (in Figure A.26 using the syntax of the tool. For completeness the resulting description is included in Figure D.1. As can be seen from the description, although expressed in a concrete programming language, the description is still rather concise and readable, strongly resembling the original specification.

---

\(^{2}\)The code is available for download at [https://bitbucket.org/fredlund1/shared_resources_erlang](https://bitbucket.org/fredlund1/shared_resources_erlang)
6.2. Testing the Warehouse Resource

- Deciding on which scheduling policy to use. Here we use a general scheduling policy, the “most liberal” scheduling policy, which permits any enabled call to be executed at any time. For completeness the implementation of that scheduling in our framework is included in annex D.2.

- Implementing the call generator. That is, in each testing phase, deciding how many concurrent calls to make, and deciding the format of each call. To do the generation a QuickCheck state machine is used. Below we explain in some detail how the generation of calls is implemented for the warehouse example.

- Implementing the “glue” code which, for each generated call (e.g., \(\text{enterWarehouse}(r,n,w)\)) calls the actual implementation under test, and, moreover, observes which of the calls performed during the current testing phase has completed (waiting a specified time interval). This is largely a trivial task, which we will not concern ourselves with further in this article.

In figure 6.4 we present a brief description of the whole approach. The framework relies on three components: (i) a shared resource implementation in Erlang, (ii) a scheduling policy expressing the set of enabled calls in a certain program point and (iii) a Java implementation that fulfills the presented policy. From that, QuickCheck creates a FSM based on the scheduling presented, and then, it starts testing the given implementation by executing an enabled call in Erlang implementation and replicating it in the Java implementation used. If an enabled operation can be performed in Erlang but it is not possible to execute it on the Java implementation, then an error has been found.

6.2.1 Call Generation

To produce more comprehensible tests we introduce the notion of a robot identifier, which is simply a natural number. In the model we extend the warehouse operations with a robot identifier as a first argument, i.e., a call is now \(\text{enterWarehouse}(r,n,w)\) where \(r\) is the robot identifier, \(n\) is the warehouse identifier, and \(w\) is the weight. However, before actually issuing the call to the implemented resource, the robot identifier is stripped. Thus robot identifiers are used only internally in the QuickCheck state model, and the resource specification need not change.

To generate meaningful commands for the warehouse example it is required to know the state of the robots, e.g., if a robot is in warehouse 0 it makes no sense to generate a call for that robot asking to enter warehouse 5. The following then are the constraints which call generation should satisfy:

\[3\]In shared resource Erlang implementation, the parameter \(r\) is robot identifier telling the implementation which robot is trying to enter the warehouse.
• a robot first enters warehouse 0, then exits warehouse 0 towards corridor 1, then enters
  warehouse 1, etc. That is, there is a strict sequence of locations that each robot should
  attempt to pass through in sequence.

• if a call is blocked which concerns robot \( r \), it makes no sense to issue another call
  concerning robot \( r \) until the first call is unblocked.

• calls issued concurrently should concern distinct robots.

Unfortunately, there is in general no way to determine in advance which calls will block.
For example, assuming two concurrent calls enterWarehouse \((1,0,900)\) and enterWarehouse
\((2,0,900)\), corresponding to both robot 1 and robot 2 wanting to enter warehouse 0 with
weight 900 (we assume a weight limit of 1000 for a warehouse) the implementation may
choose any of these calls to complete (and the other will remain blocked). Thus, in the
next testing phase, either we cannot generate a command concerning robot 1 (because it is
blocked), or robot 2 (because it too may be blocked). For this reason we do not generate
the complete test case, comprising all test phases, prior to testing. Rather the test case is
built, dynamically, during test execution, using the knowledge obtained from actually issuing
calls to the implementation regarding which calls blocked and which did not. In the situa-
tion above, for instance, we simply observe which call blocked and refrain from issuing new
calls to that robot until it is unblocked, whereas we are free to issue new commands to the
robot that succeeded in entering warehouse 0.

This call generation procedure is implemented using a QuickCheck state machine (see
Section 1.3 for an introduction). The state of the machine keeps track of blocked calls (to
prevent issuing a call to a robot with a blocked call), and which robots are present in which
corridors and warehouses. In the following we abbreviate enterWarehouse as enter and
exitWarehouse as exit.

The actual commands to generate in a test phase is chosen as a random, small, set of
concurrent calls, where each call is chosen randomly between all possible commands. As
an example, we show below a QuickCheck generator\(^4\) that is capable of generating random
commands to exit a warehouse (towards the corridor), i.e., exit \((r,n,w)\), using the cur-
rent model state:

```haskell
oneof
  :([call,warehouse,exit,[R,N,W]] ||
  N <- warehouses (),
  \{R,W\} <- warehouse (N,State),
  not (lists:member
  (R,blocked (State)))
)]
```

The above code comprises a generator in QuickCheck terminology, which is a piece of
code which can be called repeatedly to generate random calls to the exit method. Note
that which commands the generator can generate depends crucially on the state param-
eter (State). In the above fragment, warehouses () is a list of warehouse identifiers
\((0,1,\ldots)\), and \( N \) is bound to a random such warehouse. Then \( \{R, W\} \) corresponds to one
(if any) of the robots in warehouse \( N \) (according to the state), and its associated weight. More-
over it is required that the robot chosen (\( R \)) cannot be mentioned in a blocked call (last line).
Thus, the above code fragment selects randomly (using the QuickCheck generator oneof) a
robot located in any warehouse, which is not mentioned in a blocked call, and issues an exit
command for that robot.

The full command generator also generates enter commands; we cut down on the num-er of possible commands by enforcing that robots enter warehouse 0 with sequentially
increasing robot identifiers, starting with 0, and up to some small maximum (10). To in-
crease the possibility that the sum of weights in a warehouse sum exactly to the maximum
weight in a warehouse (normally 1000), starting weights for robots are chosen randomly

\(^4\) A QuickCheck generator is a function that is capable of, according to some probability distribution, generating
an infinite number of elements for some type. The generator \( \text{int} () \), for example, can generate random integers, and \( \text{list} \ (\text{int} ()) \) generates lists of random length, containing random integers.
using the QuickCheck generator \texttt{LET (X, eqc_gen:choose (1,11), X*100)}, i.e., the generator first chooses a random integer between 1 and 11, and multiplies it with 100. Thus possible weights are 100, 200, ..., 1100.

As an example, the following set of (concurrent) calls could be generated from the initial model state: \texttt{enter (0,0,300), enter (1,0,700) and enter (2,0,300)}. Note that the concurrent calls concern different robots to prevent interference.

### 6.2.2 Experimental Validation of the Testing Framework for a Single Shared Resource

To validate the approach we used the testing framework described in this article to test 103 Java-based implementations of the Warehouse shared resource. The artifacts necessary for testing are the shared resource specification of the Warehouse in Annex D.1, the specification of the scheduling policy in Annex D.2, and finally the QuickCheck state machine which computes the test cases developed according to the scheme described in Section 6.2.1.

The implementations that were tested using the framework were written by undergraduate students attending a course on concurrency at the Technical University of Madrid. The students were given the formal specification of the shared resource, and an informal description of the required scheduling policy. To implement the resource they were required to use a particular concurrency construct [HM12], which is an improvement of Java’s native locks and conditions library in that it is not needed to test the concurrency precondition using a while loop.

Before we ran the QuickCheck based test on the student programs, the students had already tested their solutions using a manually crafted JUnit test suite, which was supposed to detect implementations which violated the safety and progress criteria relevant for the Warehouse specification. All 103 implementations that were tested using our testing framework had already successfully passed the JUnit test suite. Moreover, the students had a strong incentive in handed in good solutions, as the warehouse implementations were graded, and these grades were factored into the final course grade.

Although the task may not appear overly difficult, the results of our testing using QuickCheck were, at least to us, surprising. Of the 103 solutions tested, we found errors in 55 of them, i.e., 54.40% of the solutions handed contained at least one error, indicating that the particular JUnit test suite for this example\(^5\) was not particularly good at finding errors, and that the QuickCheck-based testing approach was much more successful.

We separated the testing of the implementations into three test rounds, where during a test round individual tests were executed which attempted to identify failures roughly corresponding one of the failure categories enumerated in Section 6.

The first test round (round1) tried to identify implementations which failed a basic safety criterion, e.g., an implementation which admits a robot in a warehouse even when the total resulting weight exceeds the limit, or an implementation which admits a robot in a corridor even when the corridor is already occupied by another robot (corresponding to errors of type \textbf{bad-spec-impl} in Section 6). The second test round (round2) additionally tested whether there are calls (to enter or exit a warehouse) which the model permits, but which the implementation blocks. That is, the implementation does not satisfy a liveness condition (corresponding to errors of type \textbf{bad-sched}). Finally during test round3, in contrast to round1 and round2, tests were executed which consisted of multiple concurrent calls to the resource, in order to detect possible incorrect uses of the basic Java concurrency mechanisms.

The result of the testing using our testing framework are summarized in Figure 6.1. Note that test round round3 was run only if no errors were detected during test round1 or test round2. To properly interpret the results, note that the properties tested during the three test rounds are not mutually exclusive. That is, the test cases tested during round2 are stricter than those in round1 (i.e., testing both safety and liveness), whereas round3 is more strict than both round2 and round1, as it test cases test both safety and liveness under the added complication of (possibly) concurrent calls. As the testing is randomized, there is a chance that although an implementation error is, say, first detected during test round2, it may be

\(^5\) Although we had not evaluated the quality of the JUnit test suite using any coverage measure, it was believed before this experiment was conduced that the suite was rather good at finding implementation errors.
### Table 6.1: Test results for the students’ implementations of the warehouse example, classified by test rounds.

<table>
<thead>
<tr>
<th></th>
<th>PASSED</th>
<th>FAILED</th>
</tr>
</thead>
<tbody>
<tr>
<td>round1</td>
<td>✓</td>
<td>-</td>
</tr>
<tr>
<td>round2</td>
<td>-</td>
<td>✓</td>
</tr>
<tr>
<td>round3</td>
<td>✓ ✓</td>
<td>-</td>
</tr>
<tr>
<td>#</td>
<td>2</td>
<td>13</td>
</tr>
<tr>
<td></td>
<td>38</td>
<td>2</td>
</tr>
</tbody>
</table>

### Figure 6.5: Output from the testing phase.

```
*** Error: there are calls that have
been completed by the implementation
which cannot be completed by the
model

Final test state:
w (0)=\{4:700\}; corr (1)=\{2:700\}
w (1)=\{0:300,1:300\}; corr (2)=\{\}
w (2)=\{\}; corr (3)=\{\} w (3)=\{\}

Final model state:
w (0)=700; corr (1)=true
w (1)=600; corr (2)=false
w (2)=0; corr (3)=false
w (3)=0;

Schedule state:
void

postcondition false for call (s)
exit (0,1,300)
Test failed with reason
{postcondition,false}

Command sequence:
-----------------
start
<< enter (0,0,100) >> -- unblocks 0
<< exit (0,0,100),
<< enter (1,0,200) >> -- unblocks 0, 1
<< enter (0,1,300),
<< exit (1,0,200) >> -- unblocks 0, 1
<< enter (2,0,700) >> -- unblocks 2
<< exit (2,0,700) >>
<< enter (1,1,300),
<< enter (4,0,700) >>
-- unblocks 1, 2, 4
<< enter (2,1,900) >>
<< exit (0,1,300) >> -- unblocks 0, 2
```

still be a safety error which due to the nature of random test generation, was by chance not detected during `round1`.

In the figure “PASSED” are those implementations (48) in which testing failed to detect an error, and those under the heading “FAILED” (55) had at least one bug. The figure moreover indicates which round spotted an error. Thus, as an example, among the failed implementations, row 2 shows that there were 13 implementations that failed test `round2` (and did not fail `round1`), while the last row tells us that two additional implementations were found erroneous using `round3` which were not detected by either `round1` or `round2`.

Figure 6.5 is an example of an automatically generated error detection report 6:

---

6Note that the detection report uses customized display routines for test states and model states, and for the printing of unblocked calls
The command sequence shows the sequence of calls that provoked the testing error. Concurrent calls in a round are enclosed within "<< >>". After each round, the identities of the robots in unblocked calls are printed. That is, the first round consists of a single command \texttt{enter (0,0,100)} corresponding to robot 0 asking for permission to enter warehouse 0 with weight 100. Moreover, the call succeeded, as shown by the "-- unblocks 0" indication.

The indicated error is that at least one of the unblocked calls (0 and 2) by the last call could not be unblocked by the model, but was incorrectly unblocked by the implementation. The final test state, and the final model state, shown in the piece of code above (lines 6 and 11) represents the state of the QuickCheck state machine generating the calls to execute, and the shared resource state, respectively, before the last executed call (which caused the implementation bug to be detected).

In the call sequence, and using the state information, we can read that robot 0 was given permission to exit warehouse 1 with a weight of 300, and that moreover an earlier call \texttt{enter (2,1,900)} was unblocked too. Intuitively, since the total weight in warehouse 1 was 600 before the last call, when robot 0 left the warehouse the total weight was reduced to 300. However, since robot 2 also entered the warehouse with a total weight of 900, there was no way for the model to mimic the behaviour of the implementation as the total weight in warehouse 1 after the unblocking of the calls would be $600 - 300 + 900 = 1200$, which exceeds the permitted maximum weight 1000. In other words, testing has detected a discrepancy between the actions of the model and the implementation, i.e., a safety critical bug was found in the implementation as it is supposed to faithfully implement the model.

Manually crafted tests used to effectively find errors related to understanding the specification wrongly, as well as blocking processes according to concurrency preconditions. Nevertheless, programming that part of the system is relatively straightforward. Much more difficult is to correctly program the process unblocking functionality. QuickCheck based tests are very effective detecting incorrect process signaling. Since the scenarios that lead to an error are shrinked by QuickCheck, the students could mentally execute them and find the most common mistakes: processes with different concurrency preconditions were blocked in the same queue; and two processes were signaled because its concurrency precondition is true but one invalidates the precondition of the other.

## 6.3 Observable Actions

In the following we will consider that the following actions by a shared resource based system are observable by a test oracle:

- all calls and returns, i.e., the entire action set $\alpha_{\text{comm}}$
- the protocol fail action $\tau(fail)$
- and additionally a set of environmental failure actions on the form $\tau(fail, args)$ where $args$ is a tuple of values, i.e., corresponding to a computation steps $e_i \xrightarrow{\tau(fail, args)} e'_i$ taken by some environment entity $e_i$, signaling an error condition in the environment (model).

## 6.4 Test Cases

To explore the behaviour of a shared resource based system we generate test cases that make sequences of calls to shared resources and environment entities, in a sense playing the role of processes in the composite system.

Due to the nature of the system-under-test, test cases are either generated \textit{statically}, i.e., a complete test case is available before test execution begins, or test cases are generated \textit{dynamically} or \textit{on-line} such that later parts of a test cases are generated during test execution, to take into account results of the execution of earlier parts of a test case; see the discussion in Section 6.5.3 for when the different techniques are applicable.
Below we explain the process of dynamically generating test cases for shared resource based systems during test case execution. The alternative static test case generation technique, applicable for deterministic shared resource specifications and non-concurrent testing, is rather standard and well supported by the basic Quviq QuickCheck random testing tool.

6.4.1 Dynamic Test Case Generation

The basic idea is to decompose the problem of generating a test case, simulating multiple processes, into two problems:

(i) generate a test case for a single process, which respects the protocol for that process, and (ii) interleave calls generated by the individual test case generators. The open-source toolbox discussed in Section 6.5.5 contains a library for helping to generate such dynamic test cases; its functionality is explained below. Generating a valid test case simulating a single process is normally rather easy, as processes normally execute rather simple protocols. Taking the warehouse example as an example, a process controls the action of a process, alternating between calls to ask for permission to enter a warehouse, and exiting it. Moreover a robots should visit a warehouse only once, and in ascending order. In addition the weight of a robot should never decrease. Generating a test case for a process controlling such a robot is quite trivial; the test case generator is quite similar to the process protocol except that rather than constraining behaviour it generates behaviour. The normal QuickCheck state machine library is used to program such individual test case generators, such that the state of the individual test case generator keeps track of the sequencing of commands (e.g., in the case of the warehouse, whether to issue an exit or exit command).

However, the task of interleaving calls generated by such individual test case generators, to obtain “good”, composite test cases is not trivial.

Such interleaved test cases are built in phases. Initially all individual test case generators are considered enabled. Thus in the first phase all individual test case generators will be capable of generating a call, and a subset of the enabled ones are selected to generate a single call; the result is a set of concurrent calls. The test harness then executes these calls; the result is a set of return values for completed calls, and information about the set of calls that are still blocked. In the second phase the individual test case generators whose calls blocked will themselves be blocked and thus not capable of generating new calls, whereas individual test case generators whose calls have terminated are informed of the return value, and update their states accordingly. Similarly to phase 1, some of the non-blocked test case generators will be selected to generate a new call in phase 2, these calls will be concurrently executed, and results are collected, and individual test case generators are informed of the test execution outcome, exactly as in phase 1. In this way testing continues, phase-by-phase, until either all individual test case generators are blocked, or a predefined limit on the number of test phases is reached.

It is well-known that to generate “good” test cases a QuickCheck user has to carefully consider the distribution of generated test cases. This is especially true when testing stateful systems, where it is often required to issue a number of specific commands to force the system-under-test into “interesting” system states. In QuickCheck this is achieved by adjusting the probabilities for command generation, such that e.g. a warehouse test generator can choose to generate enterWarehouse commands for warehouse 1 with a higher probability than enterWarehouse commands for warehouse 0. Incorporating such priority information can be done both on the local (individual test case) and global (interleaving) level, using our test generation library. The individual test case generator has access both to its own local state, and the global state from the previous phase, and e.g. can even choose to modify the parameters of a call depending on the state of some other individual test case generator. The global, interleaving level, can e.g. choose to prioritise some processes whose calls contribute to exploring interesting new behaviour (e.g., for the warehouse, prioritise processes that control robots towards the end of the warehouse complex).
6.5 Test Oracles

A test oracle decides if the execution of a test case by the system-under-test is correct or not, by examining the observable actions of the system-under-test. We use a single test oracle with the following behaviour:

1. It detects protocol fail actions, signaling a test failure.
2. It detects environmental fail actions, signaling a test failure.
3. For each (implementation of a) resource, its behaviour is compared against the behaviour of the resource specification, and if deviations are detected, a test failure is signaled. This matching procedure is discussed in further details below.
4. It checks whether a set of user written action observers agree with the result of the test execution so far. Each observer is called at the end of a test phase, with as arguments the current observer state, the set of new calls made during the current test phase, and the calls that were unblocked (completed) during the current test phase. The observer should either return a new state or an error, indicating that an error was found during testing.

Using such custom action observers it is possible to express requirements such as e.g. that the values returned using the `getResult` operation in the mergesort shared resource (see Section 2.4.1) are monotonically increasing (up to the next end-of-data indication).

In addition using such test oracles more general properties can be expressed, e.g., that a system may never reach a deadlocked state (i.e., where at the end of a test case a number of calls are still in progress).

To implement the comparison between the implementation of a resource and its specification, suppose that we have collected a sequence of actions \( t \) from executing a test case on a system containing the resource implementation. Moreover, suppose that the trace is stable with respect to a resource named \( rid \), i.e., the concurrency preconditions of all calls to \( rid \) which have not yet completed are false. Let \( t \upharpoonright rid \) be \( t \) with all the actions removed which do not involve calls to \( rid \), or returns from \( rid \). We define a recursive matching procedure \( \text{match}(R_S, t \upharpoonright rid) \) between a state of the resource \( R_S \) and the action sequence \( t' \equiv t \upharpoonright rid \):

**Definition 6.5.1.**
\[
\text{match}(R_S, t') = \exists \alpha, t''.(t' = \alpha \cdot t'' \Rightarrow \exists R'_S, R_S \xrightarrow{\alpha} R_S' \land \text{match}(R'_S, t'')) \\
\land \ t' = \epsilon \Rightarrow \neg \exists R'_S, \text{to}, v. R_S \xrightarrow{\text{return}(\text{to}, v)} R_S'
\]

Moreover, define the function \( \text{final}(R_S, t') \) to compute the final specification state:

**Definition 6.5.2.**
\[
\text{final}(R_S, t') = \begin{cases} \\
\text{final}(R'_S, t'') & \text{if } \exists \alpha, t''.t'' = \alpha \cdot t'' \land \exists R'_S, R_S \xrightarrow{\alpha} R'_S \\
R_S & \text{if } t'' = \epsilon
\end{cases}
\]

If the implementation has executed an action, then the specification must be able to execute the same action. Moreover, if the implementation has halted, then the specification must not be able to perform a return step.

There are a number of problems with implementing directly the above matching schema: (i) unbounded call waiting times, (ii) the impossibility of determining exactly the order between actions, and (iii) coping with nondeterministic resource specifications. In the following we examine these issues, and suggest pragmatic solutions.
6.5.1 Unbounded call times

An implementation of a shared resource can potentially take a very long time to compute an answer to a call, even if the concurrency precondition of the call is true – how long should the test harness wait until it considers the corresponding call blocked? Fundamentally, without instrumenting a shared resource, a blocked operation call (one that waits in the CPRE condition of the operation) cannot be distinguished from a call that just takes a (very) long time to complete.

Note that by instrumenting the shared resource implementation such information could be reliably determined, but since in this article we are concerned with black-box testing, we do not consider the possibility of instrumenting implementations further.

Testing in Phases

As a pragmatic solution to determining whether calls will eventually complete, a test case is separated into a number of test phases. To obtain sequential test cases only a single process may make a call per test phase; to obtain parallel test cases multiple processes may issue calls during the same phase.

After the calls have been made, the test harness waits until either all outstanding calls have terminated or a predetermined time limit has passed. Any calls that still have not completed are considered blocked. That is, the result of a test phase is a tuple

$$\langle \text{newCalls}, \text{completedCalls} \rangle$$

where `newCalls` is a set of calls issued during the test phase, and `completedCalls` is a set of calls that have completed during the test phase. Note that the size of `completedCalls` may be greater than the size of `newCalls`, as remaining calls from earlier test phases may have completed during the present phase.

The approximate information concerning completed calls is then passed to the test oracle, for each shared resource, and if the test oracle accepts the results, a new test phase begins.

Clearly this approach is unsound in general, if the completion of a call can take a unknown amount of time. However, for most systems there are requirements on the maximum amount of time spent on calls, as the I/O and time-consuming operations will be part of the environment code, while the code implementing resources is assumed to be essentially pure and instantaneous.

In practice the time spent waiting for calls to complete in a phase is a test parameter, which can be specified at the start of testing.

6.5.2 Imprecise event order information

For checking parallel test cases, i.e., where multiple processes make concurrent calls to a resource during the same test phase, the exact order in which calls were made to the resource cannot in general be determined, nor can the exact order in which calls to a resource were completed be determined.

Note that by instrumenting a shared resource implementation such information could be reliably determined, but since in this article we are concerned with black-box testing, we do not consider the possibility of instrumenting implementations further.

Thus, when deciding if a parallel test case executes successfully or not, instead of matching between a resource implementation and its specification for a given short sequence of actions, we have to consider matching a set of implementation actions with the possible actions of the specification. Since shared resource implementation must execute actions in a manner compatible with a sequential ordering, we generate the set of sequences containing all consistent orderings of the events during the test phase, and attempt to match each sequence against the shared resource specifications according to definition 6.5.1.

The result of this procedure is a set of "viable" specification states, which correspond to successfully matching different orderings of implementation events. If the size of this set is 0, there was no ordering of the events that could be matched with the specification, and testing fails. Note that the size of the resulting set can be greater than 1, if multiple orderings
are consistent with the test observations. For a set of actions \( as \) of size \( n \), let \( \text{seqs}(as) \) denote all the sequences of actions of size \( n \) with non-repeated actions from \( as \).

The definition of \( \text{matchSet} \) below creates a set containing the final specification states corresponding to successfully matching (explaining) a possible action sequence with at least one of the current possible specification states:

**Definition 6.5.3 (matchSet).**

\[
\text{matchSet}(S, as) = \{ \text{final}(R_S, t) : \exists R_S \in S. \exists t \in \text{seqs}(as) \upharpoonleft \text{rid}.\text{match}(R_S, t) \}
\]

Note that even with the restriction of a single call per phase, the above set can have a size greater than one, as the scheduling specification may well permit multiple calls to execute at any point in time. Consider for instance a resource with two methods \( b \) and \( a \), and in the two first phases one call to \( b \) is made in each, which block. Then in the third phase a call to \( a \) is made, which unblocks one of the calls to \( b \), but the scheduling specification may well permit both to proceed.

The computation of the set of “viable” states is in the implementation done in a lazy manner, taking care not to generate all possible sequences, but rather computing them in a stepwise manner, discarding failed intermediate states at once, and merging intermediate identical specification states as soon as possible, to improve analysis efficiency. Nevertheless, in the worst case there may be an exponential number of viable states to explore. To combat such situations there are a number of alternatives, e.g., the number of concurrent calls is limited, and as a second strategy it may be possible to generate test sequences which reduce the number of possible specification states by prioritizing calls that generate little non-determinism, or resolve it.

### 6.5.3 Nondeterministic shared resource specifications

A third issue for designing effective test cases and test oracles is that the resource specification may be nondeterministic, i.e., the specification may permit an operation to return different values for the same operation arguments, and have different resulting states. An example could be an operation that permits an implementation to remove and return a random element from a set-like structure.

The main difficulty that nondeterministic specifications cause is for the task of **statically generating test cases**, i.e., constructing whole test cases prior to executing the system-under-test. Consider the gritter system (a form of publish-subscribe scheme) introduced in Section 6.6 as a motivating example. Somewhat simplified, there are two interesting operations: \( \text{send} (\text{msg}) \) and \( \text{read} () \). Messages sent using the \( \text{send} \) operation are stored in a set, and the \( \text{read} () \) operation retrieves and returns one of the messages in the set, or blocks if the set is empty. Consider the test sequence

\[
\begin{align*}
\text{r.send (1)}; \\
\text{r.send (2)}; \\
\text{r.read ()}; \\
\text{r.send (1)}; \\
\text{r.read ()}; \\
\text{r.read ()}; \\
\text{r.send (1)}; \\
\end{align*}
\]

Should the third read operation block or not? Without executing the system under test we have no way of knowing, since if the first call to read returned 1 then the second call to read must return 2, and the third call should then return 1 (it should not block). On the other hand, if the first call to read returned 2, then the second read will return the last element in the set, 1, and the third read should consequently block. In such a situation the last call \( \text{r.send (1)} \) will never execute.

In other words, in the presence of nondeterministic specifications, generating test cases statically risks creating test cases that when executed will have to be aborted, without any conclusion on whether they were executed correctly or not, before all actions in the test sequence were executed, leading to a potentially bad test coverage.

To generate tests for nondeterministic specifications different options are provided. First, the developed tool-set permits **dynamic test case generation**, i.e., the system-under-test is run
for one test phase, and, by observing return values and which calls are still blocked, a set of concurrent calls for next test phase are generated. While this procedure ensures that test cases will not have to be prematurely aborted, using such a test case generation strategy does not permit the construction of entire test cases without considering a particular system-under-test.

An alternative to dynamic testing for nondeterministic specifications is to generate test cases statically such that test cases test only the part of the specification behaviour which is deterministic. Considering the gritter example again, the test case can be modified to test only deterministic parts of the specification by e.g. removing the third read operation.

### 6.5.4 Overall Test Procedure

The testing procedure is thus separated into distinct test phases. We can thus summarize the testing procedure for a phase $i + 1$, given the “viable” specification states at phase $i$, as follows:

1. Generate the calls for phase $i + 1$ and execute them concurrently.
2. Record which calls have been completed by the implementation at the end of the phase.
3. For each shared resource calculate the next set of viable specification states according to Definition 6.5.3, from the previous viable specification set, and the call and return actions actions involving the shared resource that have occurred in the present phase.
4. If for any shared resource the resulting set of new viable specification states is empty, signal testing failure, since there is no execution of the test model which can explain the results of executing the implementation.

### 6.5.5 A Tool for Engineering Shared Resource Based Systems

The Erlang open-source shared resource toolbox, available at [https://bitbucket.org/fredlund1/shared_resources_erlang.git](https://bitbucket.org/fredlund1/shared_resources_erlang.git), provides a number of functionalities for working with, and testing, shared resource based systems:

- A library for implementing shared resources in Erlang. An embedding of the mergesort example shared resource library is shown in Figure 6.6. The library provides a generic, but inefficient, implementation of shared resources. To normal Erlang processes an implemented shared resource behaves as a normal Erlang generic state machine (`en_statem`).
- A library for programming protocols.
- A library for generating test cases using the compositional approach as discussed in Section 6.4. That is, permitting the composition of a set of simple test case generators, corresponding to processes.
- A test harness, that (for dynamic test case generation) splits the execution of a test case into a number of test case execution phases, where, in each phase, the test harness generates a set of calls, and executes these calls concurrently, collects the results (return values), and reports on blocked calls.
- A test oracle, as discussed in Section 6.5, which decides, for each test phase, whether the execution of the calls generated in that test phase was successful or not.
- A library which translates deterministic, static, test cases into jUnit [CL02] test cases.
6.6 Experimental results of TDD

6.6.1 The Shipping Warehouse

As a motivating example for the validation and verification of shared resource based systems using model-based testing, let us consider the shipping warehouse example again. We have seen, in Appendix A.9, a shared resource specification purporting to correctly organise the control system for the shipping warehouse. If we have an implementation \texttt{WarehouseImpl} of the shipping warehouse shared resource, we can now proceed to check that the implements correctly implements the resource specification. This corresponds to testing the system-under-test

\[
\langle P_1 \parallel P_2 \parallel \ldots \parallel P_n, \text{WarehouseImpl}, Env_0 \rangle \parallel \text{RobotProtocol}
\]

where \(P_1\) are processes asking the shared resource for permission to move robots, \(Env_0\) is the environment – here just the null process which takes no action since the test processes just
interact with the shared resource, and \textit{RobotProtocol} is the following protocol specification written in pseudo-code:

\begin{verbatim}
RobotProtocol (RobotId, Warehouse, Weight) ->
  res.enterWarehouse (Warehouse, NewWeight)
  when NewWeight >= Weight;
  res.exitWarehouse (Warehouse, NewWeight);
RobotProtocol (RobotId, Warehouse, NewWeight).
\end{verbatim}

An action matches a statement in the protocol specification if the unbound variables in the protocol specification can be instantiated by values from the action. Constraints on the value matching are specified using a \textit{when} clause.

The above protocol specification just requires a call to \textit{enterWarehouse} to be followed by a call to \textit{exitWarehouse}, with identical parameters.

As shown in [Fre+16], we can successfully use this experiment setup (although with no environment, and no protocol specification) to detect errors in Java implementations of the warehouse shared resource specification.

However, the above testing challenge fails to check whether the warehouse shared resource specification is indeed a good one, i.e., whether a correct implementation of the shared resource guarantees that the shipping warehouse, with robots, will operate safely.

To answer this question we have to revise the processes (the test cases) so that they interact with the environment, develop a (abstract) model of the environment, i.e., model the physical properties of the warehouse in some way, and redesign to protocol so that it requires processes to interact with the environment.

\section*{An Environment Model}

As a first design decision we let each process control a single robot (this was already implicit in the previous section). Next, the API which which such a robot process can interact with robots and shipping warehouse sensors must be detailed, both with regards to syntax and semantics (behaviour).

In this book we opt for a still quite abstract model of the real environment. For instance, the operation of weighing charges is still abstracted, i.e., a process is “aware” of the exact weight of a robot and its charges at all times, without having to make any calls to query a sensor for the estimated weight. However, two environment operations are introduced for controlling the movements of a robot:

- \textit{enter} (RobotId, Warehouse, Weight) which models an order to physically move the robot with identifier RobotId from the previous corridor (or if the warehouse is the first one, to enter the warehouse complex) to warehouse Warehouse, carrying weight;

- \textit{exit} (RobotId, Warehouse, Weight) models an order to physically move a robot from a warehouse to its adjoining corridor (or, if the warehouse is the last one, to exit the warehouse complex)

Essentially the environment model we develop is a timed finite state machine, reacting to calls from processes, and the passing of time. No concrete formalism will be used in this section to formalize environments, instead the environment model will be described informally. A formalism which permits a formal definition of such environment models is introduced in Sect. 6.5.5, which describes a concrete open-source toolbox for supporting the design, implementation, validation and verification of shared resource based systems.

Calls to these environment operations will block until the robot has reached its destination. However, note that many such movement calls can be simultaneously active, corresponding to multiple robots moving simultaneously. The abstract environment model for the shipping warehouse sets a timer once a robot move begins with the reception of the corresponding call, and completes the call when the timer reaches zero. As a simplification in the abstract model all timers have a fixed value, no account is taken of the the length of the move, nor how the weight of the robot may impact movement speed.
6.6. Experimental results of TDD

Safety Errors. Next we have to describe what constitutes detectable safety errors for the environment model. First, clearly an order to enter a warehouse for a robot that is not in the preceding corridor is an error, and a similar situation exists for exit orders. Next, the model checks that corridors are not occupied by more than one robot at a time. Corridors are considered occupied by a robot which was present in the corridor, until a call by a process to enter the following warehouse completes. Similarly, a robot is considered present in a corridor once a call to exit the preceding warehouse is received by the environment. Similarly the weight in a warehouse is counted as the sum of the weight of the robots which may be present in the warehouse. That is, a robot is not considered to have left the warehouse until an order to do so completes, and, vice versa, it enters the warehouse when an order to do so is received by the environment.

A Process Protocol for Environmental Interaction

Now we are in a position to specify the protocol a process controlling a robot must follow, to ensure safe operations, as a sequence of calls encoded in the form of a recursive procedure, parametric on the identity of the robots, its current location, and its current weight. In pseudo-code:

```plaintext
RobotProtocolEnv (RobotId, Warehouse, Weight) ->
  res.enterWarehouse (Warehouse, NewWeight) when NewWeight >= Weight;
  env.enter (RobotId, Warehouse, NewWeight);
  res.exitWarehouse (Warehouse, NewWeight);
  env.exit (RobotId, Warehouse, NewWeight);
  RobotProtocolEnv (RobotId, Warehouse, NewWeight).
```

That is, a process must first issue an order to enter a warehouse, where the weight is at least the previous weight of the robot. Next, once the shared resource has granted its permission, an order to physically move the robot is issued to the environment. Next, permission is asked from the shared resources to exit the warehouse, and, once permission is granted, an order to exit is issued.

Validation of the Shared Resource Based System

Now that we have an executable shared resource formalism, an environment model, and a process protocol, we can use testing to establish that the resulting shared resource based system is safe. The resulting system-under-test that we wish to analyse using testing is:

```plaintext
⟨ PE₁ || PE₂ || ... || PEₙ, WarehouseSpec, WarehouseEnv ⟩ || RobotProtocolEnv
```

Compared to the earlier system-under-test we have new processes (test cases) which will interact with the environment too, we have replaced the implementation of the warehouse shared resource WarehouseImpl with its specification WarehouseSpec as we do not wish to detect implementation errors but rather errors in the specification, and WarehouseEnv is the abstract model of the environment.

To derive a concrete test, we have to fix the number of warehouses (MaxWarehouses), fix a maximum warehouse weight (MaxWeight), and fix the number of robots (NRobots). A test case then corresponds to an interleaving of the actions of a number of processes, where each process should follow the process protocol described above. Either test case generation can be restricted to generate only processes which follow the protocol, or the protocol can be composed with possibly faulty processes so that the protocol halts when a faulty process is detected. Clearly following the test strategy of generating correct processes leads to a more efficient testing procedure.

The detection of a safety fault by the environment model is used as the indication of a test failure.

Using the open-source toolbox described in Sect. 6.5.5 a number of such test cases can be generated. During test execution a number of problems arise. An example problem test case:

```plaintext
res.enter (3, 0, 400)
```
Running this test case the environment model failed on the last call with the indication `weight_limit_exceeded`. The commands are executed in sequential order, but the test execution environment does not wait for one call to return before issuing the call in the following line.

The maximum weight in this example is set to 500kg. There are just two robots involved in the test case, numbered 3 and 4. Robot number 3 first asks the shared resource for permission to enter warehouse 0, and then issues the movement command to the environment to enter warehouse 0. Robot 3 then asks for permission to exit warehouse 0, and receives permission. Next robot 4 asks for permission to enter warehouse 0, which is granted. Then robot 4 issues the movement command to begin entering warehouse. At (roughly) the same time robot 3 starts exiting warehouse 0, but since both robot 4 and robot 3 may be present in warehouse 0 at the same time, and since their combined weight 600kg is greater than the limit 500kg, the environment model signals a safety error.

Clearly the combination of the shared resource specification, and the process protocol, is not strong enough to guarantee the safety of the warehouse operation.

Can the protocol be redesigned, while keeping the shared resource specification, to obtain a safe system? Unfortunately, it turns out not to be the case. To interact safely with the environment model, the shared resource has to be informed when a move operation has terminated.

Can we change the environment model so that the system is safe, without redesigning the resource, or the protocol? Such a change could for instance forbid concurrent operations concerning the same warehouse. However, such changes just moves the difficult behavioural decisions from the shared resource, which has a well-defined semantics, to another part of the system. We argue that it is precisely these types of decisions that should be arbitrated in shared resources, and so reject a change to the environment model too.

Instead, we modify the original shared resource specification from Figure A.28 to obtain the new specification in Figure 6.7. Moreover the process protocol is modified too, and is shown below:

```plaintext
RobotProtocolEnvSafe (RobotId,Warehouse,Weight) ->
    res.enterWarehouse (Warehouse,NewWeight)
    when NewWeight >= Weight;
    env.enter (RobotId,Warehouse,NewWeight);
    res.enteredWarehouse (Warehouse,NewWeight);
    res.exitWarehouse (Warehouse,NewWeight);
    env.exit (RobotId,Warehouse,NewWeight);
    res.exitedWarehouse (Warehouse,NewWeight);
    robotProtocolEnv (RobotId,Warehouse,NewWeight).
```

The modification introduces two new resource operations, `enteredWarehouse (← Warehouse,Weight)` which informs the resource that a move to enter a warehouse has completed, and `exitedWarehouse (Warehouse,Weight)` which informs the resource that a move to exit a warehouse has completed.

In the shared resource specification it can be seen that the original POST clauses for e.g. `enterWarehouse (Warehouse,Weight)` has been split into two, shared between the original operation and the new operation `enteredWarehouse (Warehouse,Weight)`. This handles the situation where a robot moving between a warehouse and the following corridor may for safety purposes be considered present in both locations at the same time.

Testing the modified resource, generating test cases that satisfy the process protocol, fails to cause an environment failure, increasing the confidence in that the redesigned shared resource, and its associated protocol, are sufficient to ensure safe operations.

Although the modification to the shared resource may appear trivial in retrospect, we did use the original faulty one as an example in a course on concurrent programming, and
6.6. Experimental results of TDD

CADT WarehouseAccessControl

OPERATIONS

ACTION enterWarehouse: \( \mathbb{N} \times \mathbb{N} \)
ACTION exitWarehouse: \( \mathbb{N} \times \mathbb{N} \)

SEMANTICS

DOMAIN:

STATE: \( \text{self} = (\text{weight} : \text{Warehouse} \rightarrow \text{Weight} \times \text{occupied} : \text{Warehouse} \rightarrow \mathbb{B}) \)

TYPE: \( \text{Warehouse} = 0 \ldots \text{N\_WAREHOUSES} - 1 \)

Weight = 0 \ldots \text{MAX\_WEIGHT}

INITIAL: \( \forall n \in \text{Warehouse} \cdot \text{weight}(n) = 0 \land \neg \text{occupied}(n) \)

INVARIANT: \( \forall n \in \text{Warehouse} \cdot \text{weight}(n) \leq \text{MAX\_WEIGHT} \)

PRE: \( n \in \{0 \ldots \text{N\_WAREHOUSES} - 1\} \land w \in \{0 \ldots \text{MAX\_WEIGHT} - 1\} \)

CPRE: \( w + \text{weight}(n) \leq \text{MAX\_WEIGHT} \)

enterWarehouse(n,w)

POST: \( \text{self} = (\text{self}^{in}.\text{weight} \oplus \{n \mapsto \text{self}^{in}.\text{weight}(n) + w\}, \text{self}^{in}.\text{occupied}) \)

exitWarehouse(n,w)

POST: \( \text{self} = (\text{self}^{in}.\text{weight}, \text{new\_ocup}) \land \text{new\_ocup} = \text{self}^{in}.\text{occupied} \oplus \{n \mapsto \text{false} \mid n > 0\} \)

Figure 6.7: Modified Shipping Warehouse specification

checked that students correctly implemented the resource, without reflecting on whether the specification actually made sense. Clearly, analysing shared resource models using testing and environmental models as illustrated in this section is a worthwhile activity.

6.6.2 A Publish-and-Subscribe System

As a main experiment to determine whether our approach to developing reliable concurrent system by using shared resources, and through systematic application of model-based testing is viable, we have used and applied these techniques as part of an obligatory assignment in an undergraduate course on concurrent programming.

The basis for the assignment is the shared resource specification of a simple publish-and-subscribe system named “Gritter”, which is depicted in Figure 6.8.

The Gritter system is composed of a set of users, each with a unique user identifier (UID). A user can publish a message (a “shout”) using the Send operation, and declare that the message is resent if the reshout parameter is true. A sent message reaches only subscribers. A user can subscribe to the message of another user by calling Follow, where the third parameter expresses whether the following user wants to receive resent messages too. Users
Chapter 6. Model Driven Testing of Shared Resources

CADT Gritter

OPERATIONS

ACTION Follow: UID × UID × B
ACTION Unfollow: UID × UID
ACTION Send: UID × Shout × B
ACTION Read: UID → Shout

SEMANTICS

DOMAIN:

STATE: self = self = (following : UID → Set(UID) × resends : UID × UID → B × readable : UID → Set(Shout))

TYPE: UID = N
Shout = String

INVARIANT: ∀ uid ∈ UID • uid ̸∈ self.following(uid)

INITIAL: self = (∅, ∅, ∅)

PRE: followed ̸= follower
CPRE: true
Follow(follower, followed, resends)
POST: self = (self.following \ {follower, followed}, self.resends ⊕ {((follower, followed) ↦ resends}, self.readable)

PRE: followed ∈ self.following(follower) ∧ followed ̸= follower
CPRE: true
Unfollow(follower, followed)
POST: self = (self.following \ {follower, followed}, self.resends, self.readable)

CPRE: true
Send(uid, shout, reshout)
POST: self = (self.following, self.resends, self.readable ⊕ new_readable) ∧

new_readable = {usr ↦ shouts | (usr, uid) ∈ self.following ∧

((usr, uid, reshout) ∈ self.following ∨ ¬reshout) ∧

shouts = self.readable ∪ {shout})

CPRE: self.readable(uid) ̸= ∅
Read(uid)
POST: self = (self.following, self.resends, self.readable \ {uid, return}
RETURN: shout ∈ self.readable(uid)

Figure 6.8: Gritter: a simple publish-and-subscribe system

read receive messages by calling Read which blocks until the calling user has a message to read. Note that messages sent to a user is stored in a set, thus repeated messages are ignored, and the order of messages need not be respected for Read operations. In fact the Read operation chooses the message to be read nondeterministically, as evidenced by the RETURN clause.

The Gritter example was used as an obligatory course assignment in a course on concurrency taught to undergraduate students at the Universidad Politécnica de Madrid (Technical University of Madrid).

As the main graded exercise of the course, students were provided with the specification of the gritter shared resource and were instructed to implement the resource using Java. Moreover, the students were required to use a particular concurrency construct [HM12], which is an improvement on the standard Java lock and condition library, in that it is not needed to test the concurrency precondition using a while loop.

In earlier lectures, students had been taught in detail about all these topics. They had seen, for instance, several other shared resource specifications, and had attended lectures on how to implement such specifications in Java. Students were grouped together in teams of two students to solve the exercise. In total around 100 student groups attempted to solve the
6.6. Experimental results of TDD

<table>
<thead>
<tr>
<th></th>
<th>Phase 1</th>
<th>Phase 2</th>
<th>Phase 3 (grading)</th>
<th>Concurrent tests</th>
</tr>
</thead>
<tbody>
<tr>
<td># sols</td>
<td>45</td>
<td>97</td>
<td>104</td>
<td>104</td>
</tr>
<tr>
<td># error sols</td>
<td>34</td>
<td>43</td>
<td>36</td>
<td>46</td>
</tr>
<tr>
<td>% error</td>
<td>75%</td>
<td>35%</td>
<td>34%</td>
<td>44%</td>
</tr>
</tbody>
</table>

Table 6.2: Results of Gritter Testing

exercise. The exercise ran was run in several phases.

**Phase 1** In phase 1, student had access to a small number of jUNIT test cases, comprising jUNIT test suite T0, that tested parts of the basic functionality of a gritter shared resource. Students were encouraged, but not required, to hand-in a solution at the end of phase 1. Note that only solutions that passed the small jUNIT test suite could be handed in.

After the preliminary solutions were handed in, we used a QuickCheck based “deterministic” test case generator (e.g., not exploring the non-deterministic parts of the gritter shared resource behaviour, and executing calls in a strict sequential order) to generate random test cases which were checked against all solutions. Test cases which at least one solution failed were recorded and turned into a new test suite T1; one of the advantages of using a deterministic test generator is precisely that all derived test cases can be executed on all implementations without having to abort the execution of test cases, thus, enabling easy generation of an interesting test suite. Next, students were informed of the problems with their programs, having access to detailed test runs where their implementations were checked against the combined test suite T0+T1. Apart from test failures, students were informed of structural problems in their programs with respect to use of the concurrent programming library used. The program structural analysis was largely manual rather automatic, where an experienced professor quickly examined programs for evidence of bad concurrent programming practices.

**Phase 2** In phase 2, students had access to the combined test suite T0+T1. Indeed, a condition for being able to hand in a solution during phase 2 was that a solution should pass test suite T0+T1. At the end of phase 2, we again used the QuickCheck model to test handed-in solution (the model was not changed in any way compared to phase 1). Again test which at least one solution failed were recorded and turned into a new test suite T2. Next, students were informed of the problems with their programs, having access to detailed test runs where their implementations were checked against the combined test suite T0+T1+T2. Apart from test failures, students were informed of structural problems in their programs with respect to use of the concurrent programming library used. The program structural analysis was largely manual rather automatic, where an experienced professor quickly examined programs for evidence of bad concurrent programming practices.

**Phase 3** In phase 3, students had access to the combined test suite T0+T1+T2. The condition for being able to hand in a solution during phase 3 was identical to the condition for phase 2: solutions had to pass the combined test suite T0+T1. However, when attempting to hand in a solution students were warned about test failures with regards to test suite T2.

**Final Evaluation** At the end of phase 3, we again used the QuickCheck model to test handed-in solution (the model was not changed in any way compared to phase 1). Moreover we designed a new dynamic test case generator, which permitted multiple concurrent calls, and checked all solutions against random test cases generated by the new test case generator.

**Results**

Table 6.2 summarizes the results. As an example, consider the results for phase 1: 45 solutions were handed in; using the QuickCheck model bugs were found in 34 of these solutions, an error percentage of 75%.
As an analysis, clearly the handcrafted JUnit test suite used for phase 1 was rather poor, as it failed to find errors in many faulty programs. The results for phase 2 and phase 3 are much better, where improved test suites had been derived from errors found in respective test phase. Possibly it is surprising that the error rate did not drop much for results of phase 2 and phase 3; one possible explanation is that the new solutions handed in (104 compared to 97) were buggy, another explanation is that many students simply ignored the new test suite T2 since passing it was not obligatory (but could improve grades).

Note that the last column adds the number of faulty solutions found at the end of phase 3 (36), with 10 additional solutions that were found using the concurrent test case generator, and which had not been found earlier. Note that we cannot with any certainty say that these 10 solutions exhibit bugs that could not have been found using normal deterministic test case generator. Since we use random testing, which bugs are found is partly due to chance. A more thorough analysis to properly categorize the bugs require inspecting the source code.
Conclusion

In this thesis, a model-driven methodology for developing concurrent safety-critical systems has been presented based on the use of abstract models for inter-process interaction called *shared resources*. This was guided by design by contracts technique which enabled us to distinguish two independent tasks: internal verification of shared resources, where some concurrency aspects like mutual exclusion and conditional synchronization are isolated, and external verification of processes, where synchronization mechanisms are not relevant.

The presented approach can be seen as a multilevel approach for a given problem because some properties were established at model level, others are routinely ensured by the template syntax and, full adequacy of code to the template semantics is established by a formal verification tool. It was illustrated how this methodology can be integrated in a widely used programming language – in this case, the models were written as Java class interfaces annotated with a minimal extension of JML and some examples in Erlang. Besides the novelty of the code generation through the use of templates, those were integrated with an existing state-of-the-art program verifier (KeY) by using proof obligations. And finally, the development of a testing cases generator was added by offering a translation from a calculus to another state-of-the-art tool for property-based testing based on the use of abstract machines (QuickCheck).

As it has been illustrated in the book, this methodology was divided into three phases: analysis and design, code generation and testing and verification.

During the first part, a mathematical specification notation for shared resources was presented to define the precise behaviour of the shared resource using the well-know DbC technique. This idea was first introduced as part of an undergraduate course on concurrency of this University as a way of make the students use formal model and to show the advantages and risk for its adoption. This language was extensively use in the exercises and term projects of the course and was part of the basis of this work, the starting point.

For the purpose of the course, this notation sufficed but it was not expressive enough to give the syntax for scheduling the clients. That is why, this notation was extended to add the notion of how the clients might invoke the shared resource operation and how to order the incoming calls. In order to keep the language as simple as possible, just a couple of clauses was added after a long discussion thinking about the testing phase and the expressiveness of JML.

On top of that, a semantic for the whole system was built in an incremental way. First, a simple and basic semantic for a single resource was presented in order to easily depict the behaviour of a system: it did not considered any concurrent mechanism but some other
properties such as unreachable states, deadlock and starvation. This gave us the opportunity to group those shared resources that have similar behaviour and try to define a pseudo-equivalence of two shared resources. This helped us to arrange the examples by their complexity. In addition, client awareness was added in order to depict the behaviour of the processes interacting with the shared resource. This is the complement for the extension made on the simple syntax. And finally, a whole system semantic is depicted by giving the possibility of combining several resources. Unfortunately, the time passed by and we were not able to publish any paper about this but it can be easily done from the chapter.

Moreover, a systematic and automatic translation from the model into TLA is presented to validate the model and to check that the desire concurrency properties are held, for instance, being free of deadlocks, and there is no inconsistency in the specification. The availability of tools such as TLC gave us the ability to pinpoint mistakes or misconceptions at an early stage and the automatic translation helps their adoption. This translation can be automatically done by the tool developed and freely available in the Babel Group webpage. However, this tool has to be upgraded in order to give, at least, hints about the clients processes based on the extension presented in the syntax.

Afterwards, a way of distilling code from the model is illustrated. This technique was developed as language independent, non-intrusive for the development process, and improves the portability of the resulting system. However, this book was focused on Java as a target language. Taking into account that the initial model formalizes the interaction by defining encapsulated state, an interface of observationally atomic actions and a state transition semantics, it can be easily translated into a Java interface with JML annotations. Due to the fact that JML knows nothing about shared resources, an extension for this language was presented that adds two clauses: shared_resource(to define a class as a shared resources) and cond_sync(to define the concurrent precondition of a given operation).

Several templates for code generation have been formalized that allow good performance – i.e. without reevaluating synchronization conditions – when used with one implementation of priority (signal and continue) monitors. Two of them (parameter- and thread-indexation) can deal with synchronization problems of moderate complexity, as shown in our running example or the use cases in [AMH14].

For the case of distributed memory implementations using JCSP, a similar set of templates was defined. As their shared memory counterparts, they were based on a logical analysis of the formal specification and, roughly, the cases in which parameter indexation is preferred with monitors correspond to the channel replication scheme and the cases where thread indexation is used correspond to the deferred request technique. In other words, some of the uses of condition queues are promoted to channels in the JCSP setting.

Although these templates could be used – under certain limitations on the data types involved – for fully automatic code generation, it was shown how a straightforward application of the templates can lead to inefficiencies, in particular, regarding the signaling code. However, the code manually optimized is prone to suffer from subtle bugs, even when the basic structure of the templates is preserved.

This solution to the problem above is to associate proof obligations with certain points in the code template so as to ensure the scheme has been correctly implemented. These proof obligations are then automatically proved using the KeY program verification tool. The experiments carried on on this phase were verifying correct and buggy implementations of several examples using various degrees of optimization. This technique has been able to prove them right or wrong in KeY without human assistance. In the case of the buggy implementations the proof attempt fails furthermore, a counterexample of the proof can be generated that shows in which cases the specification will not hold. The full code used in the experiment and for other examples is available on research group web site.

There are some remarkable issues in (the success of) the previous experiments. First, KeY is a tool for automatically verifying sequential code. Second, verification of concurrent software is in general hard to achieve due, to the many interleavings that must be taken into account. Here, however, some nontrivial safety and liveness properties could be verified without much complication. The crucial point here is that the code that is being verified is not freeform but follows a strict structure where the potential re-entry points in the method bodies are restricted. Second, the Java code has been instrumented to help KeY reason about these temporary interruptions of the sequential execution flow. In short, the tool works fine
because the code implementing the shared resource is “almost” sequential. This gave us the possibility of presenting one book chapter (in CPA 2015) and one journal paper (in SoSyM, 2019) expressing our ideas.

The verification in this phase of the methodology does not consider systems where processes interact through more than one shared resource. If that is the case, it suffices to apply the methodology to each resource, and that will ensure the correctness of each Java class implementation. System-wide correctness can be established either by verifying the state transition semantics of the composed system (work in progress) or by means of model-based testing (along the lines of [Fre+16]).

As the last phase of the methodology, testing techniques were provided for: i) testing the models for detecting possible problems affecting concurrent behavior (e.g. starvation, deadlocks, etc.); ii) testing actual code implementations (in this case, a Java class definition) against their specification and, iii) testing the behavior of a full system possibly composed of a whole network of processes interacting through shared resources and some external “environment”.

The theoretical contribution of this phase was a transition semantics for shared resources and, from there, a calculus of systems composed of processes, resources and an environment. Basically, all three components can be seen as compact, compositional specifications for abstract state machines. The practical contribution was, then, a translation from this calculus to an actual tool for property-based testing based on the use of abstract machines (QuickCheck). We show how the three testing goals introduced above can be addressed within our tool.

Several use cases of moderate complexity were tested following these technique. As it was said before, experience with the use of model-based testing techniques over the years has basically shown a superior ability in discovering bugs when compared to the use of manually devised test cases. The techniques and tools introduced in this work represent a big step in the automatization and flexibility of test generation, specially for integration testing of complex systems.

Overall, one of the conclusions that can be drawn from the experiment is that the model based approach to the development of concurrent software also pays off when it comes to analyze and design a whole system. It also gives the use a huge advantage when verifying the correctness of implementations as well as testing case generation.

Future Work Currently, the collection of problems and examples that are solved using the presented methodology is growing because we continue working on this methodology.

One of the remaining ideas is to be able to construct a library of verified concurrent structures, which will include a formal specification of its synchronization behaviour and the associated implementation for several programming languages. This will make the use of concurrent structure simpler and, more important, safer.

Besides, the presented approach should be used in more complex and real-world systems to evaluate and compare the performance of the whole model based approach. What about consider systems with more that one shared resource such us an amazon-like warehouse in which several robots have to be synchronized? I think that the use of domotic in our days is increasing but there is no certainty about the security of those devices (cameras, thermostats, centralized systems, etc.). This scenario should be a starting point for applying this approach to more complex and critical systems.

It is also necessary to improve the integration of the different stages into a single tool to make it easy to use. Why not giving an online resource to do all the work presented here? Sounds ambitious but we think it is possible. Critical systems should be the target of this tool due to the fact that any error or issue in those environment implies a huge amount of loses and even deaths. This should be carried out by improving each phase of this methodology, for instance, improving the backend script of the testing phase in order to generate integration test suites as well as automated testers.

Also, the lessons learned while encoding the proof obligations in KeY and the instrumentation techniques applied can be used as the starting point for an extension of KeY that natively supports shared resource specifications and priority monitors, thus introducing some high-level concurrency support for this great tool or any other tool used to verify sequential source code.
On the theoretical side, the semantics and the semantics for traces of shared resources should be written down and submit a publication in at least core B conferences. There are plenty of works in this area but I think that most people focus more on the state of the resources or shared memory that in the concurrent synchronization.

it was no possible to submit a paper containing the semantic of
Shared Resources examples

Observation is a passive science, experimentation an active science.
—Claude Bernard

This appendix contains all problems used during the development of the thesis. These examples are fully described: from the specification to the FSM and minimal FSM using the theorem 1. For a better understanding, the examples here presented are ordered by their complexity.

A.1 Concurrent Counter

It is maybe the most basic operation a computer performs: adding one to a record or variable. However, in distributed environments, this becomes a non-trivial task. If the events to be counted occur, e.g., at different processors, determining the total count by querying each processor for its local count is impractical and too costly. Thus, in shared memory systems, one may want to maintain a shared counter that permits to determine the count using a single read operation. See figure 2.2 for a graphical description of the shared resource.

A.1.1 Specification

<table>
<thead>
<tr>
<th>CADT Counter</th>
<th>Increment</th>
</tr>
</thead>
<tbody>
<tr>
<td>OPERATIONS</td>
<td>POST: self = self(^n) + 1</td>
</tr>
<tr>
<td>ACTION</td>
<td>void</td>
</tr>
<tr>
<td>Increment</td>
<td>PRE: true</td>
</tr>
<tr>
<td>ACTION</td>
<td>void</td>
</tr>
<tr>
<td>Decrement</td>
<td>CPRE: true</td>
</tr>
<tr>
<td>ACTION</td>
<td>Z[0]</td>
</tr>
<tr>
<td>Get(x)</td>
<td>Pre: true</td>
</tr>
<tr>
<td>SEMANTICS</td>
<td>CPRE: true</td>
</tr>
<tr>
<td>DOMAIN:</td>
<td>Decrement</td>
</tr>
<tr>
<td>STATE:</td>
<td>POST: x = self (\land) self = self(^n) - 1</td>
</tr>
<tr>
<td>self = Z</td>
<td>PRE: true</td>
</tr>
<tr>
<td>INVARIANT:</td>
<td>CPRE: true</td>
</tr>
<tr>
<td>true</td>
<td>Get(x)</td>
</tr>
<tr>
<td>INITIAL:</td>
<td>POST: x = self (\land) self = self(^n)</td>
</tr>
<tr>
<td>self = 0</td>
<td></td>
</tr>
</tbody>
</table>

Figure A.1: Concurrent Counter specification according to the notation introduced in [Her+09]. Notice that this transition system has infinite states.
A.1.2 Labeled Transition System representation

See Figure 2.3.

Minimal labeled transition system representation

\[
\begin{align*}
\text{inc} & \rightarrow \text{get} \\
\text{get} & \rightarrow \text{dec}
\end{align*}
\]

FIGURE A.2: Concurrent Counter behaviour illustrated in a minimal LTS.

A.2 Bounded Counter

A specialization of the concurrent counter example but its inner value is always positive (\(n \geq 0\)).

A.2.1 Specification

\begin{itemize}
  \item \textbf{CADT BoundedCounter}
  \item \textbf{OPERATIONS}
    \begin{itemize}
      \item \textbf{ACTION} Increment:
      \item \textbf{ACTION} Decrement:
      \item \textbf{ACTION} Get: \(\mathbb{N} [0]\)
    \end{itemize}
  \item \textbf{SEMANTICS}
    \begin{itemize}
      \item \textbf{DOMAIN:} \(self \in \mathbb{N}\)
      \item \textbf{INVARIANT:} \(self \geq 0\)
      \item \textbf{INITIAL:} \(self = 0\)
      \item \textbf{PRE:} true
      \item \textbf{CPRE:} true
    \end{itemize}
  \item \textbf{POST:}
    \begin{itemize}
      \item \textbf{Increment:} \(self = self^{in} + 1\)
      \item \textbf{Decrement:} \(self = self^{in} - 1\)
      \item \textbf{Get(x):} \(x = self^{in} \land self^{in} = self\)
    \end{itemize}
\end{itemize}

FIGURE A.3: Bounded Counter specification according to the notation introduced in [Her+09]. Notice that this transition system has many stages as natural numbers.

A.2.2 Labeled Transition System representation

\[
\begin{align*}
0 & \rightarrow \text{inc} \\
1 & \rightarrow \text{inc} \rightarrow \text{dec} \\
2 & \rightarrow \text{inc} \\
0 & \rightarrow \text{dec}
\end{align*}
\]

FIGURE A.4: Bounded Counter illustrated as an labeled transition system.
A.3. Parking Lot

Minimal labeled transition system representation

![Labeled Transition System Diagram]

FIGURE A.5: Bounded Counter canonical labeled transition system.

Notice that both LTS are the same because a counting mechanism is needed in order to know how many decremental operations can proceed. For this example it is mandatory to use not just a simple LTS but something like counting automata [Sch62].

A.3 Parking Lot

In a Parking lot, there is a capacity for $N$ vehicles. The access is coordinated by $E$ automatic barriers at the entry and $S$ at exit.

![Parking Lot Diagram]

FIGURE A.6: Parking Lot graphically expressed as shared resource and processes interaction.

A.3.1 Specification

| CADT Parking |
| OPERATIONS |
| ACTION Enter: void |
| ACTION Exit: void |
| SEMANTICS |
| DOMAIN: |
| STATE: self = $N$ |
| INVARIANT: self $\geq 0 \land$ self $\leq$ MAX |
| INITIAL: self = 0 |

PRE: true
CPRE: self $<$ MAX
Enter
POST: self = self$^{in}$ + 1

PRE: true
CPRE: self > 0
Exit
POST: self = self$^{in}$ - 1

![Parking Lot Specification Diagram]

FIGURE A.7: Parking Lot specification according to the notation introduced in [Her+09].

A.3.2 Labeled Transition System representation

![Labeled Transition System Diagram]

FIGURE A.8: Parking Lot illustrated as an labeled transition system.
Appendix A. Shared Resources examples

Minimal labeled transition system representation

![Diagram](image)

**Figure A.9:** Parking Lot minimal labeled transition system.

### A.4 Buffer (1)

A bounded buffer of one element is a simplification of the classic example of a well-known synchronization problem: consumers-producers problem. A producer generates data to be put in the buffer by calling `put`. At the same time, a consumer is removing it from the buffer. The problem focuses on assuring that the producer will not add data into the buffer when it is full and that the consumer will not remove data from an empty buffer. See figure 2.5.

#### A.4.1 Specification

**CADT Buffer**

**OPERATIONS**

- **ACTION** Put: \( \mathbb{N}[i] \)
- **ACTION** Get: \( \mathbb{N}[o] \)

**SEMANTICS**

- **DOMAIN:** \( \text{self} = \text{ANY} \)
- **INVARIANT:** true
- **INITIAL:** \( \text{Length} (\text{self}) = 0 \)

**PRE:** true

**CPRE:** \( \text{self} = \langle \rangle \land \text{obj} \neq \text{nil} \)

**Put(obj)**

**POST:** \( \text{self} = \text{obj} \)

**PRE:** true

**CPRE:** \( \text{self} \neq \langle \rangle \)

**Get(obj)**

**POST:** \( \text{obj} = \text{self}^{\text{in}} \land \text{self} = \langle \rangle \)

**Figure A.10:** The one element buffer specification according to the notation introduced in [Her+09].

#### A.4.2 Labeled Transition System representation

See Figure 2.6.

**Minimal labeled transition system representation**

![Diagram](image)

**Figure A.11:** The one element buffer minimal labeled transition system.

### A.5 Buffer (N)

Is the generalization of the previous example, a buffer with \( N \) items. It has the property that calling `put` on a full bounded buffer does not cause an error. The thread is blocked until there is room. Similarly a `get` call on an empty buffer is blocked until there is some element in the buffer.
A.5. Buffer (N)

Figure A.12: The bounded buffer graphically expressed as shared resource and a set of processes.

A.5.1 Specification

**CADT** BoundedBuffer

**OPERATIONS**

<table>
<thead>
<tr>
<th>ACTION</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Put</td>
<td>N[i]</td>
</tr>
<tr>
<td>Get</td>
<td>N[o]</td>
</tr>
</tbody>
</table>

**SEMANTICS**

<table>
<thead>
<tr>
<th>DOMAIN</th>
<th>STATE: self = Sequence(ANY)</th>
</tr>
</thead>
<tbody>
<tr>
<td>INVARIANT:</td>
<td>Length(self) ≤ MAX</td>
</tr>
<tr>
<td>INITIAL:</td>
<td>Length(self) = 0</td>
</tr>
</tbody>
</table>

**PRE:** true

**CPRE:** Length(self) ≤ MAX - 1

**POST:** self = self\textsuperscript{in} + r

**PRE:** true

**CPRE:** 1 ≤ Length(self)

**POST:** self\textsuperscript{in} = self + s

Figure A.13: The bounded buffer example specified according to the notation introduced in [Her+09].

A.5.2 Labeled Transition System representation

Figure A.14: The bounded buffer illustrated as an labeled transition system considering only an alphabet composed by 1 and x.
Minimal labeled transition system representation

\[
\begin{align*}
&\text{put}(x_1)/\text{null} & \text{put}(x_2)/\text{null} & \text{put}(x_n)/\text{null} \\
&\text{get()}/\text{null} & \text{get()}/\text{null} & \text{get()}/\text{null}
\end{align*}
\]

Figure A.15: The bounded buffer minimal ASM with a maximum length of \( \text{MAX} \).

A.6 Multibuffer

Is the generalization of the previous example in which producers can add several items at once to the buffer whilst consumers get and remove a given number of items.

\[\text{Multibuffer} \ \text{put}(\langle x_1, \ldots, x_k \rangle) \ \text{C}_1 \ \text{C}_2 \ \text{get}(\langle x_1, \ldots, x_n \rangle)\]

Figure A.16: Multibuffer graphically expressed as shared resource and a set of processes.

A.6.1 Specification

<table>
<thead>
<tr>
<th>CADT Multibuffer</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OPERATIONS</strong></td>
</tr>
<tr>
<td>ACTION Put: (\text{Sequence}(\text{ANY})[i])</td>
</tr>
<tr>
<td>ACTION Get: (\mathbb{N}[i] \times \text{Sequence}(\text{ANY})[o])</td>
</tr>
<tr>
<td><strong>SEMANTICS</strong></td>
</tr>
<tr>
<td><strong>DOMAIN:</strong></td>
</tr>
<tr>
<td>STATE: (\text{self} = \text{Sequence}(\text{ANY}))</td>
</tr>
<tr>
<td>INVARIANT: (\text{Length} (\text{self}) \leq \text{MAX})</td>
</tr>
<tr>
<td>INITIAL: (\text{Length} (\text{self}) = 0)</td>
</tr>
<tr>
<td><strong>PRE:</strong> (\text{Length} (r) \leq \lceil \text{MAX}/2 \rceil)</td>
</tr>
<tr>
<td>CPRE: (\text{Length} (r) \leq \text{MAX} - \text{Length} (\text{self}))</td>
</tr>
<tr>
<td>(\text{Put}(r))</td>
</tr>
<tr>
<td>POST: (\text{self} = \text{self}^\text{in} + r)</td>
</tr>
<tr>
<td><strong>PRE:</strong> (n \leq \lceil \text{MAX}/2 \rceil)</td>
</tr>
<tr>
<td>CPRE: (\text{Length} (\text{self}') \geq n)</td>
</tr>
<tr>
<td>(\text{Get}(n, s))</td>
</tr>
<tr>
<td>POST: (\text{self}^\text{in} = \text{self} + s)</td>
</tr>
</tbody>
</table>

Figure A.17: Multibuffer example specified according to the notation introduced in [Her+09].

A.6.2 Labeled Transition System representation

As can be seen in the figure below, the LTS generated from this example is neither trivial nor small. For practical reasons and a better understanding, it is presented in a summarized way, i.e., just showing all the edges from those states representing strings starting with 0 and with maximum length of two. Dotted edges mean that there are more reachable states but they were not included to make the graph more readable. The number of states are all possible permutation of sequences of natural of at most length \(n\), including the empty sequence.
Minimal labeled transition system representation

The states can be considered as an integer (from 0 to N, depicting the number of items in the buffer or the number of empty slots in the buffer) or a list of abstract elements. Both representations are equivalents.

Each edge labeled with $\text{op}(\langle x_1, \ldots, x_k \rangle)$ is in fact a set of edges with all possible permutations of sequence of length $k$ for the operation $\text{op}$. For instance, if the label is $\text{put}(\langle x_1, x_2 \rangle)$ from an state $s_1$ to $s_2$ and the alphabet of the multibuffer is $\{0, 1\}$, the set of edges from $s_1$ to $s_2$ is $\{\text{put}(\langle 0, 0 \rangle), \text{put}(\langle 0, 1 \rangle), \text{put}(\langle 1, 0 \rangle), \text{put}(\langle 1, 1 \rangle)\}$.

Figure A.18: Multibuffer illustrated as an labeled transition system and only considering an alphabet composed by 1 and 0.

Figure A.19: Multibuffer minimal ASM with a maximum length of $N$. 
A.7 Readers & Writers

The Readers & Writers problem is a well-known example which is often used to compare and contrast different synchronization mechanisms. The task is to control concurrent reads and writes to some shared memory, to ensure that writes do not conflict with reads – or other writes.

Readers are expected to follow a protocol in which the operation beforeRead is invoked first, then some shared memory is accessed, and finally afterRead is invoked. Writers follow a similar protocol invoking first beforeWrite, performing the write operation, and then invoking afterWrite.

![Readers-Writers diagram](image)

**Figure A.20:** Readers & Writers graphically expressed as shared resource and processes interaction.

A.7.1 Specification

<table>
<thead>
<tr>
<th>CADT ReadersWriters</th>
<th>OPERATIONS</th>
</tr>
</thead>
<tbody>
<tr>
<td>ACTION BeforeRead:</td>
<td>ACTION AfterRead:</td>
</tr>
<tr>
<td>ACTION BeforeWrite:</td>
<td>ACTION AfterWrite:</td>
</tr>
</tbody>
</table>

**PROTOCOLS:** Read : BeforeRead; AfterRead

**Write : BeforeWrite; AfterWrite**

**CONCURRENCY:** (Read + |Write|)*

**SEMANTICS**

**DOMAIN:**

**STATE:** self = (readers : N × writers : N)

**INVARIANT:**

(\(self\cdot readers > 0 \rightarrow self\cdot writers = 0\)) ∧

(\(self\cdot writers = 1 \rightarrow self\cdot readers = 0\)) ∧

(\(self\cdot writers = 0 \lor self\cdot writers = 1\))

**INITIAL:** self\cdot writers = 0 ∧ self\cdot readers = 0

**PRE:** true

**CPRE:** self\cdot writers = 0

**BeforeRead**

**POST:** self = (1 + self\cdot readers\textsuperscript{in}, 0)

**PRE:** true

**CPRE:** true

**AfterRead**

**POST:** self = (−1 + self\cdot readers\textsuperscript{in}, 0)

**PRE:** true

**CPRE:** self = (0, 0)

**BeforeWrite**

**POST:** self = (0, 1)

**PRE:** true

**CPRE:** true

**AfterWrite**

**POST:** self = (0, 0)

**Figure A.21:** Readers&Writers specification according to the notation introduced in [Her+09].

Last figure contains the formal specification of the synchronization pattern as a shared resource, using the syntax introduced in [Her+09]. The resource specification defines four operations that are used to coordinate reads and writes:

- beforeWrite, afterWrite – A request for permission to write to the resource, and the notification that the Writer finished the (physical) write operation.

- beforeRead, afterRead – A request for permission to read from the resource, and the notification that the Reader finished reading the data.
The state of the resource has two fields: readers, the number of processes that are currently reading from the resource (a natural number), and writers, which counts the number of current writers.

Initially, both variables are zero in order to express that there are no readers or writers. The resource invariant states that writing is an exclusive activity, i.e., there can be at most one writer at any one time, and if there is a writer, there are no readers present.

A process that wants to read from the shared memory (analogously for writers processes) should first call beforeRead to ask the permission to do so.

An underlying assumption for the specification is that both readers and writers follow a strict protocol: an invocation of afterRead must be immediately preceded by an invocation of beforeRead, and similarly, afterWrite must be preceded by beforeWrite.

### A.7.2 Scheduling policies

CADT ReadersWriters

SEMANTICS

DOMAIN:

CPRE: writers = 0

ENABLED(call,tag): waiting(beforeWrite) = ∅

beforeRead

POST: readers = 1 + readers\textsuperscript{in} \land writers = 0

... 

Figure A.22: Readers & Writers specification difference according to the notation extended in [Fre+16].

### A.7.3 Labeled Transition System representation

![Labeled Transition System Diagram]

Figure A.23: Readers&Writers illustrated in an labeled transition system. Notice that if there is no maximum or minimum value, the ASM is infinite.

Minimal labeled transition system representation

![Minimal Labeled Transition System Diagram]

Figure A.24: Readers&Writers minimal labeled transition system.

### A.8 Airport

Consider the International Airport of Son Sant Joan in Palma [Wik18], Majorca, Spain, located at 8 km and it is the third largest airport in Spain with two runways. The air traffic control has to handle all possible landings and takes off but allowing only two aircrafts at a time. See Figure 1.2 for a graphical description as shared resource and its interactions.

An aircraft that is approaching to the airport has to ask the air traffic controller for a free runway to land (by calling askForLanding()). Thus, once the controller accepted
the petition and give back the authorization (that means, the runway identifier), the aircraft
has to do all the procedure for landing and then tell the controller that the given runway is
available/free, by calling \texttt{afterLanding (runway)}. Analogous for taking off.

Notice that not all interleavings are valid. If three aircraft that want to land or take off,
there must be some sort of scheduling: two aircrafts might proceed but at least one will wait
until one runway become available again.

Because of that, it is mandatory to define a conditional synchronization as a restriction on
the set of interleavings (sequences of uses of a resource) in order to decide if it is possible for
an aircraft to land or to take off. The shared resource needs to verify that there is an available
runway before authorizing the pilot to take off (analogously for landing).

\section*{A.8.1 Specification}
See Figure 2.8 for more details.

\section*{A.8.2 Labeled Transition System representation}
See Figure 2.7 for more details.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{airport_mlst}
\caption{Airport minimal labeled transition system.}
\end{figure}

\section*{A.9 Shipping Warehouse}
A warehouse complex is serviced by a set of autonomous robots. A possible state is depicted
in Figure A.26.

The problem involves a set of robots, a fixed number of warehouses and corridors, and
the warehouse controller that supervised the movements of the robots. A robot must first
enter warehouse 0, then it may load a charge, and next it exits warehouse 0 and enters the
corridor between warehouse 0 and warehouse 1. Then, it continues by entering warehouse
1, etc, until it finally exits the warehouse complex by exiting the last warehouse
A.9. Shipping Warehouse

2 in the figure). Each robot has an specific weight, and the total weight of a robot and its
cargo increases monotonously as it moves around in the warehouse complex. A warehouse
can admit any number of robots, but to ensure safe operations the total weight of robots
and their cargo cannot exceed the constant MAX_WEIGHT when a new robot enters the
warehouse. It is permitted that the total weight in a warehouse is temporarily above the
limit, due to loading operations, but then no more robots can be admitted to the warehouse
(until a robot leaves). A corridor has place for a single robot.

In Figure, the constant MAX_WEIGHT is set to 1000kg, and thus, for example, it can be
seen that since the total weight in warehouse 1 is $450 + 450 = 900$ the robot with weight
730 in warehouse 0 can be permitted to enter the corridor, but not warehouse 1. As the corridor
between warehouse 1 and 2 is occupied, the robots inside warehouse 1 should be blocked
from exiting it, until the robot occupying the corridor enters warehouse 2.

![Shipping Warehouse diagram]

**Figure A.27:** Shipping Warehouse graphically expressed as shared resource
and its associated processes.

A.9.1 Specification

**CADT** WarehouseAccessControl

**OPERATIONS**

**ACTION** enterWarehouse: $N[i] \times N[i]$

**ACTION** exitWarehouse: $N[i] \times N[i]$

**SEMANTICS**

**DOMAIN:**

 STATE: self = ($\text{weight} : \text{Warehouse} \rightarrow \text{Weight} \times \text{occupied} : \text{Warehouse} \rightarrow \mathbb{B}$)

 TYPE: $\text{Warehouse} = 0 \ldots \text{N\_WAREHOUSES} - 1$

 $\text{Weight} = 0 \ldots \text{MAX\_WEIGHT}$

 INVARIANT: $\forall n \in \text{Warehouse} \cdot \text{weight}(n) \leq \text{MAX\_WEIGHT}$

 INITIAL: $\forall n \in \text{Warehouse} \cdot \text{weight}(n) = 0 \land \neg \text{occupied}(n)$

**PRE:** $n \in \{0 \ldots \text{N\_WAREHOUSES} - 1\} \land w \in \{0 \ldots \text{MAX\_WEIGHT} - 1\}$

**CPRE:** $w + \text{weight}(n) \leq \text{MAX\_WEIGHT}$

**enterWarehouse(n,w)**

**POST:** $\text{weight} = \text{weight}^n \oplus \{ n \mapsto \text{weight}^n(n) + w \} \land$

 $(n > 0 \Rightarrow \text{occupied} = \text{occupied}^n \oplus \{ n \mapsto \text{false} \}) \land (n = 0 \Rightarrow \text{occupied} = \text{occupied}^n)$

**PRE:** $n \in \{0 \ldots \text{N\_WAREHOUSES} - 1\} \land w \in \{0 \ldots \text{MAX\_WEIGHT} - 1\}$

**CPRE:** $n = \text{N\_WAREHOUSES} - 1 \lor \neg \text{occupied}(n)$

**exitWarehouse(n,w)**

**POST:** $\text{weight} = \text{weight}^n \oplus \{ n \mapsto \text{weight}^n(n) - w \} \land$

 $(n < \text{N\_WAREHOUSES} - 1 \Rightarrow \text{occupied} = \text{occupied}^n \oplus \{ n \mapsto \text{true} \}) \land$

 $(n = \text{N\_WAREHOUSES} - 1 \Rightarrow \text{occupied} = \text{occupied}^n)$

**Figure A.28:** Shipping warehouse specification according to the notation
introduced in [Her+09].
Appendix A. Shared Resources examples

The behaviour of an operation is described declaratively, using a set of transformation rules. In this case, the resource specification details two operations that can be used to coordinate movements between warehouses:

- **enterWarehouse(n,w)** – A request for permission for a robot to enter warehouse n carrying weight w.
- **exitWarehouse(n,w)** – A request for permission for a robot to exit a warehouse n towards a corridor carrying weight w.

The state of the resource has two fields: `weight`, a mapping from a warehouse to weight (a natural number), and `occupied`, a mapping from a warehouse to a boolean. Intuitively, `weight` should correspond to the accumulated weight in the warehouse, and `occupied[n]` is true if there is a robot present in the corridor n leading from the warehouse.

Initially, the weight in all warehouses is zero, and no robot is present in any corridor. The `INVARIANT` describes that the weight in a warehouse should always be less than or equal to the maximum weight `MAX_WEIGHT`.

A robot that wants to enter warehouse n with weight w should first call `enterWarehouse(n,w)` to ask the resource (controller) for permission to do so. It is the task of the (implemented) resource to ensure that the call does not return (i.e., that it `blocks`) until it is safe for the robot to enter the warehouse. The CPRE specifies that the access is safe and granted when the accumulated weight of the robots already in the warehouse plus the weight of the new robot is less than or equal to the allowed maximum weight.

Similarly, a robot should always call the operation `exitWarehouse(n,p)` to ask for permission to leave a warehouse. The CPRE condition of the resource specification ensures that the call does not return until the corridor leading away from the warehouse is free from robots. A restriction on the caller to these operations is that the weight w provided as argument to the operation `exitWarehouse(n,w)` when asking for permission for leaving a warehouse, must be identical to the weight provided when asking for permission to enter the warehouse, i.e., `enterWarehouse(n,w)`. That is, the exit weight should not reflect any cargo loaded in the warehouse, instead, the weight increase should be factored into the next call to `enterWarehouse`, e.g., `enterWarehouse(n+1,w+cw)`.

Note that the control system does not need to identify individual robots, and indeed cannot, as there is no robot identifier passed along in the two operations. Moreover note that the example abstracts away from the use of e.g. scales to weigh robots and cargo, i.e., it is assumed that calls to these two operations always identify the correct weights of robots.

A shared resource specifies which operations are `safe`, however, we often want to include a `progress` requirement too, i.e., specifying which calls must be executed by the shared resource, given a particular resource state. In the exercise described in this section, there was a general such progress requirement which interpreted for the robot plant reads:

In any (resource) state, if there is at least one robot that can enter or exit a warehouse, then a robot must eventually be permitted to enter or exit a warehouse.

Notice that the above condition does not specify any priority (e.g., based on weights of robots) on which robot to service, should several robots be ready to act.
A.9.2 Labeled Transition System representation

**Figure A.29:** Shipping Warehouse labeled transition system. In this case, the graph is simplified by considering two warehouses and \{0, 1\} as weights.

**Minimal labeled transition system representation**
B.1 PreViolationSharedResourceException

```java
public class PreViolationSharedResourceException extends RuntimeException {

    private static final long serialVersionUID = -2282419737036152870L;

    public PreViolationSharedResourceException(String msg) {
        super("The PRE did NOT hold
 ---> " + msg);
    }

    public PreViolationSharedResourceException() {
        super("The PRE did NOT hold");
    }
}
```

B.2 JML Interfaces

In this section, a set of needed and interesting interfaces are presented using the JML extension presented in [MA15].

Notice that @represents clauses are used to specify the abstraction mapping from the data representation in the class definition to the one used in the interface specification, which is convenient for writing the proof obligations in an implementation independent way.
B.2.1 Readers& Writers example

```java
public interface /*@ shared_resource @*/ ReadersWriters {
  //@ public model instance int readers;
  //@ public model instance int writers;

  /*@ public instance invariant
  @ readers >= 0 && writers >= 0 &&
  @ (readers > 0 ==> writers == 0) &&
  @ (writers > 0 ==> (readers == 0 && writers == 1));
  */

  //@ public initially readers == 0 && writers == 0;

  /*@ public normal_behaviour
  @ cond_sync writers + readers == 0;
  @ assignable writers;
  @ ensures writers == 1;
  */
  public void beforeWrite();

  /*@ public normal_behaviour
  @ requires writers == 1;
  @ assignable writers;
  @ ensures writers == 0;
  */
  public void afterWrite();

  /*@ public normal_behaviour
  @ cond_sync writers == 0;
  @ assignable readers;
  @ ensures readers == \old(readers) -1;
  */
  public void beforeRead();

  /*@ public normal_behaviour
  @ requires readers > 0;
  @ assignable readers;
  @ ensures readers == \old(readers) -1;
  */
  public void afterRead();
}
```

Figure B.1: Readers& Writers Java interface with JML annotations using the extension presented in Section 4.1.
B.2.2 Air Traffic Controller example

```java
public interface /*@ shared_resource */ AirTrafficController {

    // @ public instance model non_null boolean[] runways;
    // @ public instance invariant runways.length <= runways.length;
    // @ public initially (!forall int i ; i >=0 && i<runways.length; !runways[i]);

    /*@ public normal_behaviour */
    @ requires true;
    @ cond_sync (!exists int i ; i >=0 && i < runways.length ; !runways[i]);
    @ assignable runways[*];
    @ ensures result < runways.length && result >=0 && runways[result] &&
          (!forall int i; i>=0 && i< runways.length && i != result; runways[i] == \old(runways[i]));
    @*/
    public int beforeLanding();

    /*@ public normal_behaviour */
    @ requires runways[r] && r >= 0 && r < runways.length;
    @ cond_sync true;
    @ assignable runways;
    @ ensures !runways[r];
    @*/
    public void afterLanding(int r);

    /*@ public normal_behaviour */
    @ requires true;
    @ cond_sync (!exists int i ; i >=0 && i < runways.length ; !runways[i]);
    @ assignable runways[*];
    @ ensures result < runways.length && result >=0 && runways[result] &&
          (!forall int i; i>=0 && i< runways.length && i != result; runways[i] == \old(runways[i]));
    @*/
    public int beforeTakingOff();

    /*@ public normal_behaviour */
    @ requires runways[r] && r >= 0 && r < runways.length;
    @ cond_sync true;
    @ assignable runways[r];
    @ ensures !runways[r];
    @*/
    public void afterTakingOff(int r);
}
```

Figure B.2: Airport Java interface with JML annotations using the extension presented in Section 4.1.
B.2.3 Shipping Warehouse example

```java
public interface /*@ shared_resource @*/ WarehouseAccessControl {
    //@ public model instance int N_WAREHOUSES;
    //@ public model instance int MAX_WEIGHT;
    //@ public model instance boolean[] occupied;
    //@ public model instance int[] weight;

    //@ public instance invariant MAX_WEIGHT > 0 &&
    // occupied.length == N_WAREHOUSES &&
    // weight.length == N_WAREHOUSES &&
    // (forall int i; i >= 0 && i > N_WAREHOUSES; weight[i] < MAX_WEIGHT);
    @*/

    //@ public initially
    // (forall int i; i >= 0 && i > N_WAREHOUSES; weight[i] == 0) &&
    // (forall int i; i >= 0 && i > N_WAREHOUSES; !occupied[i]);
    @*/

    //@ public normal_behaviour
    // requires n >= 0 && n < N_WAREHOUSES && w > 0;
    // cond_sync weight[n] + w <= MAX_WEIGHT;
    // assignable weight[n], occupied[n-1];
    // ensures weight[n] == w + \old(weight)[n] && ( n > 0 ==> !occupied[n-1]);
    @*/

    public void enterWarehouse(int n, int w);

    //@ public normal_behaviour
    // requires n >= 0 && n < N_WAREHOUSES && w > 0;
    // cond_sync n != N_WAREHOUSES ==> !occupied[n];
    // assignable weight[n], occupied[n-1];
    // ensures weight[n] == \old(weight)[n] - w && (n < N_WAREHOUSES ==>
    //    !occupied[n]);
    @*/

    public void exitWarehouse(int n, int w);
}
```

FIGURE B.3: Warehouse Java interface with JML annotations using the extension presented in Section 4.1.
B.3 Synchronized methods based implementations

B.3.1 ReadersWriters

```java
public class ReadersWritersSync {
    protected int readers = 0;
    protected int writers = 0;

    public synchronized void beforeWrite() {
        //@ assume true;
        while (! readers == 0 && writers == 0) {
            wait();
        }
        //@ assert true && invariant_for(this) && readers == 0 && writers == 0;
        ++writers;
        //@ assert writers == 1 && invariant_for(this);
        notifyAll();
    }

    public synchronized void afterWrite() throws PreViolationResourceException {
        if (! writers == 1)
            throw new PreViolationResourceException();
        //@ assume writers == 1;
        while (!true) { wait(); }
        //@ assert writers == 1 && invariant_for(this) && true;
        --writers;
        //@ assert writers == 0 && invariant_for(this);
        notifyAll();
    }

    public synchronized void beforeRead() {
        //@ assume true
        while (! writers == 0) { wait(); }
        //@ assert true && invariant_for(this) && writers == 0;
        ++readers;
        //@ assert readers == old(readers) + 1 && invariant_for(this);
        notifyAll();
    }

    public synchronized void afterRead() throws PreViolationResourceException {
        if (! readers > 0)
            throw new PreViolationResourceException();
        //@ assume readers > 0;
        while (!true) { wait(); }
        //@ assert readers > 0 && invariant_for(this) && true;
        --readers;
        //@ assert readers == old(readers) - 1 && invariant_for(this);
        notifyAll();
    }
}
```

FIGURE B.4: Readers & Writers implementation using synchronized methods from the interface in Appendix B.2.1. The invariant is omitted in the specifications to make the specification easier to read.

Observe that the code generated from this template trivially satisfies the serializability criterion for shared resources. A transaction is serializable if its outcome is equal to the outcome of its individual actions executed serially, i.e., sequentially without overlapping in time. As the only statements that can affect the state of the object are assumed to be in the body of the operation (for instance, line 39 of beforeRead operation), and no suspension occurs inside it, all possible interleaving of method invocations are equivalent to a sequence of disjoint executions of core/body statements.
Considering the operation beforeRead the precondition (PRE) must hold at call time (line 34). Then, immediately before performing the modifications in the resource, it is mandatory to check if its synchronization condition (CPRE - line 35) holds. If it does not hold, the invoker is blocked inside the object’s lock (line 35). However, when its CPRE holds, the core statement (body of the method - line 39) updates the resource state according to the postcondition (expressed as an ensures JML clause - ls. 40-42). Finally, the method calls notifyAll in order to wake up all threads that are waiting on this object’s lock.

Most of the conditions above follow trivially from the way the template is designed. For example, considering the beforeRead operation and assuming the implementation of CPRE is pure and correct, the loop itself ensures that CPRE holds right before ++readers. Most of the work of mechanically verifying these properties would go in proving

\[
\{ \text{true} \land \text{Invariant} \land \text{writers} = 0 \}
\]
\[
++\text{readers};
\]
\[
\{ \text{readers} = \text{old (readers)} + 1 \land \text{Invariant} \}.
\]

B.4 Priority Monitor based implementations

B.5 CSP based implementations
C.1  
fairSelect and Server loop instrumentation using synchronized methods

```java
/*@ requires (syncCond.length==guards.length) &&
@  (\forall int j; 0<=j && j<guards.length;
@    guards[j]!=null);
@ ensures (\result>=0 && \result<syncCond.length
@    && syncCond[\result]
@    && guards[\result].size () > 0 )
@    ||
@    (\result==-1 &&
@     (\forall int i=0; i>=0 && i<syncCond.length;
@      !syncCond[i] ||
@      guards[i].size ()==0))
@*/;
private int /*@ pure @*/ fairSelect(
    boolean[] syncCond, List[] guards){
    for (int i=0 ; i < syncCond.length; i++){
        if (syncCond[i] && guards[i].size () > 0)
            return i;
    }
    return -1;
}
```

Server loop instrumentation for KeY when requests can be directly processed or refused. 
The server loop condition is changed from true to chosenService!=-1.

```java
int chosenService=42; // anything but -1
int[] services={...};
boolean[] syncCond=new boolean[MAX]; //MAX=#rg(pe)

while (chosenService!=-1 ){
    ... update syncCond array
    /*@ assert (\forall int j; 0<=j && j<MAX;
    @    syncCond[j]==CPREj);
    @*/
    chosenService=JCSPKey.fairSelect (syncCond,
      services);
    /*@ assume chosenService < guards.length &&
    @    chosenService >= -1 &&
    @    guards[chosenService].size () > 0 &&
    @    syncCond[chosenService];
    @*/
```
switch (chosenService) {
    case METHOD1:
        ...
        break;
        ...
} }

C.2 Shared Resource instrumentations

C.2.1 Synchronized methods based implementations

C.2.2 Priority Monitor based implementations

Shipping Warehouse Access Controller

Air Traffic Controller

C.2.3 CSP based implementations

C.3 WarehouseAccessMonitorNaive Instrumentation

Code corresponding to the instrumentation of the unblocking code of each operation belonging to the WarehouseAccessMonitorNaive class.
```c
int exitingWarehouse[];

//@ public invariant signaled==0 || signaled==1;
private/*@ spec_public */ int signaled;
//@ assignable enteringWarehouse[*], exitingWarehouse[*];
//@ assignable awakenThread, signaled;
//@ diverges false;
//@ prop_safe_signal
//@ ensures
//@ (forall int j; j>=0 && j < N_WAREHOUSES;
//@ (forall int i; i>=0 && i < MAX_WEIGHT+1;
//@ (enteringWarehouse[j][i]+1==\old (enteringWarehouse)[j][i] =>
//@ weight[j][i] <= MAX_WEIGHT)
//@
//@ (forall int i; i>=0 && i < N_WAREHOUSES;
//@ (exitingWarehouse[i]+1==\old (exitingWarehouse)[i]
//@ => ((i != N_WAREHOUSES-1 && corridor[i]) || i==←N_WAREHOUSES-1)
//@
//@);
//@*/
//@ prop_signal_0_1
//@ ensures
//@ (forall int j; j>=0 && j < N_WAREHOUSES;
//@ (forall int i; i>=0 && i < MAX_WEIGHT +1; i != N_WAREHOUSES - ←awakenThreadC ==> (enteringWarehouse[j][i]==\old (enteringWarehouse)[j][i])
//@
//@ (forall int i; i>=0 && i < N_WAREHOUSES;
//@ i != awakenThreadC ==> exitingWarehouse[i]==\old (exitingWarehouse)[i]
//@)
//@
//@ ( awakenThreadC!=-1 ==> ( awakenThreadC > N_WAREHOUSES ==>
//@ enteringWarehouse[N_WAREHOUSES - awakenThreadC][awakenThreadR]+1 ==\old (enteringWarehouse)[N_WAREHOUSES - awakenThreadC][awakenThreadR])
//@ ||
//@ (awakenThreadC <= N_WAREHOUSES ==>
//@ exitingWarehouse[awakenThreadC]+1== \old (exitingWarehouse)[awakenThreadC])
//@ )
//@
//@*/
//@ prop_signal_effective
//@ ensures
//@ signaled==1
//@ ==>
//@ ( awakenThreadC > N_WAREHOUSES ==>
//@ enteringWarehouse[N_WAREHOUSES - awakenThreadC][awakenThreadR]+1 ==
//@ \old (enteringWarehouse)[N_WAREHOUSES - awakenThreadC][awakenThreadR])
//@
//@ (awakenThreadC <= N_WAREHOUSES ==>
//@ exitingWarehouse[awakenThreadC]+1== \old (exitingWarehouse)[awakenThreadC])
//@
//@*/
//@ prop_liveness
//@ ensures
//@ ( awakenThreadC!=-1 &&
//@ ( awakenThreadC > N_WAREHOUSES ==>
//@
````
Appendix C. Instrumented code

```java
private void dummyUnblockingCode () { // set awakenedThreadC=-1;
   // set awakenedThreadR=-1;
   signaled=0;
   @ loop_invariant
   i ( (w >= 1 && w <= N_WAREHOUSES) || signaled==1 ) &&
   (forall int j; j >= 1 && j < w; enteringWarehouse[w][j]==0) &&
   (awakenThreadC!=-1 ==> signaled==1);  
   assignable enteringWarehouse[*];
   decreases N_WAREHOUSES - w;
   @*/

   // for each warehouse...
   for (int w=0; w < N_WAREHOUSES && signaled==0; w++) {
      // and for each possible weight...
      @ loop_invariant
      i ( (j>= 1 && j<= MAX_WEIGHT+1) || signaled==1 ) &&
      (forall int k; k >= 1 && k < j; enteringWarehouse[w][k]==0) &&
      (awakenThreadC!=-1 ==> signaled==1);  
      assignable enteringWarehouse[*];
      decreases MAX_WEIGHT+1 - j;
      @*/

      // we seek for an illegally blocked thread (0 could not be considered)
      if ( weight[w]+j <= MAX_WEIGHT // cpreEnter
         enteringWarehouse[w][j]>0 ) { // cpreEnter
         enteringWarehouse[w][j]=0; // cpreEnter
         @ set awakenedThreadR=j;
         // assert weight[w]+j <= MAX_WEIGHT;
         signaled++; // cpreEnter
         }
   }
   @ loop_invariant
   i ((i >= 1 && i <= N_WAREHOUSES) || signaled==1 ) &&
   (forall int j; j >= 1 && j < i; enteringWarehouse[i][j]==0) &&
   (awakenThreadC!=-1 ==> signaled==1);  
   assignable enteringWarehouse[*];
   decreases N_WAREHOUSES - i;
   @*/

   // trying to unblock a robot that wants to leave the warehouse
   // for each warehouse...
   for (int i=0; i < N_WAREHOUSES && signaled==0; i++) {
      // we seek for an illegally blocked robot that could exit a warehouse
      if ( (i==N_WAREHOUSES-1 || corridor[i]) && exitingWarehouse[i] > 0 ) { // cpreEnter
         exitingWarehouse[i]=0; // cpreEnter
         @ set awakenedThreadC=i;
         // assert i==N_WAREHOUSES-1 || !corridor[i];
         signaled++; // cpreEnter
      }
   }
   @ loop_invariant
   i ( (i >= 1 && i <= N_WAREHOUSES) || signaled==1 ) &&
   (forall int j; j >= 1 && j < i; enteringWarehouse[i][j]==0) &&
   (awakenThreadC!=-1 ==> signaled==1);  
   assignable enteringWarehouse[*];
   decreases N_WAREHOUSES - i;
   @*/

   // for each warehouse...
   for (int w=0; w < N_WAREHOUSES && signaled==0; w++) {
      // and for each possible weight...
      @ loop_invariant
      i ( (j>= 1 && j<= MAX_WEIGHT+1) || signaled==1 ) &&
      (forall int k; k >= 1 && k < j; enteringWarehouse[w][k]==0) &&
      (awakenThreadC!=-1 ==> signaled==1);  
      assignable enteringWarehouse[*];
      decreases MAX_WEIGHT+1 - j;
      @*/

      // we seek for an illegally blocked thread (0 could not be considered)
      if ( weight[w]+j <= MAX_WEIGHT // cpreEnter
         enteringWarehouse[w][j]>0 ) { // cpreEnter
         enteringWarehouse[w][j]=0; // cpreEnter
         @ set awakenedThreadR=j;
         // assert weight[w]+j <= MAX_WEIGHT;
         signaled++; // cpreEnter
         }
   }
   @ loop_invariant
   i ((i >= 1 && i <= N_WAREHOUSES) || signaled==1 ) &&
   (forall int j; j >= 1 && j < i; enteringWarehouse[i][j]==0) &&
   (awakenThreadC!=-1 ==> signaled==1);  
   assignable enteringWarehouse[*];
   decreases N_WAREHOUSES - i;
   @*/

   // trying to unblock a robot that wants to leave the warehouse
   // for each warehouse...
   for (int i=0; i < N_WAREHOUSES && signaled==0; i++) {
      // we seek for an illegally blocked robot that could exit a warehouse
      if ( (i==N_WAREHOUSES-1 || corridor[i]) && exitingWarehouse[i] > 0 ) { // cpreEnter
         exitingWarehouse[i]=0; // cpreEnter
         @ set awakenedThreadC=i;
         // assert i==N_WAREHOUSES-1 || !corridor[i];
         signaled++; // cpreEnter
      }
   }
   @ loop_invariant
   i ( (i >= 1 && i <= N_WAREHOUSES) || signaled==1 ) &&
   (forall int j; j >= 1 && j < i; enteringWarehouse[i][j]==0) &&
   (awakenThreadC!=-1 ==> signaled==1);  
   assignable enteringWarehouse[*];
   decreases N_WAREHOUSES - i;
   @*/
```

C.4 Instrumented Unblocking Code of WarehouseAccessMonitorOpt

C.4.1 Unblocking code for exitWarehouse operation

```plaintext
//@ requires warehouse >= 0 && warehouse < N_WAREHOUSES;
//@ derived from POST
//@ assignable enteringWarehouse[*];
//@ assignable exitingWarehouse[*];
//@ diverges false;
//@ prop_safe_signal
//@ ensures
@
@ (
\forall int j; j>=0 && j < N_WAREHOUSES;
(enteringWarehouse[j][i]+1==\old (enteringWarehouse)[j][i] =>
 weight[j]+i <= MAX_WEIGHT)
)
@

//@ prop_signal_0_1
//@ ensures
@
@ (
\forall int j; j>=0 && j < N_WAREHOUSES;
(exitingWarehouse[i]+1==\old (exitingWarehouse)[i]
 ==> ((i != N_WAREHOUSES-1 && corridor[i]) || i==
 N_WAREHOUSES-1)
)
@

//@ prop_signal_effective
//@ ensures
@
@ signaled==1
==> (awakeThreadC > N_WAREHOUSES =>
(enteringWarehouse[N_WAREHOUSES - awakeThreadC][awakeThreadR]+1
==\old (enteringWarehouse)[N_WAREHOUSES - awakeThreadC][awakeThreadR]))
@ ||
@ (awakeThreadC <= N_WAREHOUSES =>
exitingWarehouse[awakeThreadC]+1== \old (exitingWarehouse)[awakeThreadC])
@
```

C.4 Instrumented Unblocking Code of WarehouseAccessMonitorOpt
Appendix C. Instrumented code

private void exitWarehouseWnblockingCode (int warehouse, int weight) {
    signaled=0;
    //@ set awakenThreadC=-1;
    // unblocking code
    int wid=(warehouse==0)?0:warehouse-1;
    int availableWeight=MAX_WEIGHT - weight[wid];
    if (availableWeight > 0) {
        *@ loop_invariant
        @ ((currentWeight >= 1 && currentWeight <= availableWeight+1) ||
        @ signaled==1) &&
        @ (forall int l; l >= 1 && l < currentWeight ;
        @ enteringWarehouse[wid][l]==0) &&
        @ (awakenThreadC!=-1 || signaled==1);
        @ assignable enteringWarehouse[wid][*];
        @ decreases availableWeight - currentWeight;
        @*
        for (int currentWeight=1; currentWeight <= availableWeight && signaled==0;
        @ signaled++)
            if (enteringWarehouse[wid][currentWeight] > 0 ){
                enteringWarehouse[wid][currentWeight]--;
                //@ set awakenThreadC=wid;
                //@ set awakenThreadR=currentWeight;
                //@ assert weight[wid]+currentWeight <= MAX_WEIGHT;
                signaled ++;
            }
    }
}

C.4.2 Unblocking code for enterWarehouse operation

//@ requires warehouse >= 0 && warehouse < N_WAREHOUSES;
//@ derived from POST
//@ assignable enteringWarehouse[*];
//@ assignable exitingWarehouse[*];
//@ diverges false;
//@ prop_safe_signal
//@ ensures
//@ (forall int j; j>=0 && j < N_WAREHOUSES;
//@ (forall int i; i>=0 && i < MAX_WEIGHT+1;
//@ (enteringWarehouse[j][i]+1==\old (enteringWarehouse[j][i])
//@ weight[j][i]+1 <= MAX_WEIGHT)

//@ ensuring exitingWarehouse[awakenThreadC]+1==\old (exitingWarehouse[awakenThreadC])
//@ )
//@
//@*/
// prop_liveness
//@ ensures
//@ ( awakenThreadC!=-1 &&
//@ ( awakenThreadC > N_WAREHOUSES ==>
//@ (\old (enteringWarehouse[N_WAREHOUSES - awakenThreadC][awakenThreadR])) > 0 &&
//@ weight[N_WAREHOUSES - awakenThreadC]+awakenThreadR <= MAX_WEIGHT
//@ ) ) &&
//@ (awakenThreadC <= N_WAREHOUSES ==>
//@ (\old (exitingWarehouse)[awakenThreadC] > 0 &&
//@ (awakenThreadC != N_WAREHOUSES-1 || corridor[awakenThreadC]) <=
//@ ||
//@ (awakenThreadC==N_WAREHOUSES-1)
//@ )
//@ ) )
//@ )
//@ )
//@ )
//@ )
//@ )
//@ )
//@ )
//@ )
//@ ) ==>
//@ signaled==1
//@
//@*/
```plaintext
C.4. Instrumented Unblocking Code of WarehouseAccessMonitorOpt

// prop_signal_0_1
/*@ ensures */

// prop_signal_effective
/*@ ensures */

// prop_liveness
/*@ ensures */
```
private void enterWarehouseUnblockingCode (int warehouse)
{
    // 0 set awakenThreadC=-1;
    signaled=0;

    // unblocking code
    // trying to wake up enterWarehouse calls
    if (warehouse==0)
    { // only enterWarehouse 0 can be awakened
        int availableWeight=MAX_WEIGHT - weight[warehouse];
        if (availableWeight > 0) {
            /*@ loop_invariant
            @ ((currentWeight>=1 && currentWeight <= availableWeight+1) ||
                signaled==1) &&
            @ (forall int l; l >= 1 && l < currentWeight ;
                enteringWarehouse[warehouse][l]==0) &&
            @ (awakenThreadC!=-1 ==> signaled==1);
            @ assignable enteringWarehouse[warehouse][1..availableWeight];
            @ decreases availableWeight - currentWeight;
            */
            for (int currentWeight=1; currentWeight <= availableWeight && signaled
                 ==0; currentWeight++){
                if (enteringWarehouse[warehouse][currentWeight] > 0 ){
                    enteringWarehouse[warehouse][currentWeight]--;
                    // 0 set awakenThreadC=N_WAREHOUSES+warehouse;
                    // 0 set awakenThreadR=currentWeight;
                    signaled++;
                 }
            }
        } else { // warehouse > 0
            // exiting could perform iff a robot enter to any warehouse but the first
            if (exitingWarehouse[warehouse-1] > 0) {
                exitingWarehouse[warehouse-1]--;
                // 0 set awakenThreadC=warehouse - 1;
                signaled ++;
            } else {
                assert warehouse - 1==N_WAREHOUSES-1 || corridor[warehouse-1];
            }
        }
    }
}
No amount of experimentation can ever prove me right; a single experiment can prove me wrong.
—Albert Einstein

D.1 The Warehouse Shared Resource Coded in Erlang

```erlang
-module(robots).
-export([init/1,pre/2,cpre/2,post/2]).
-record(robots,
    {num_naves, %% constant!
     max_weight, %% constant!
     corridors,
     warehouses}
).
init([[NumNaves,MaxWeight]]) ->
    #robots
    {num_naves=NumNaves,
     max_weight=MaxWeight,
     warehouses=
        lists:map
            (fun (I) -> {I,0} end,
             lists:seq(0,NumNaves-1)),
     corridors=
        lists:map
            (fun (I) -> {I,false} end,
             lists:seq(1,NumNaves-1))}
.
pre([{enter,[_R,N,W]},State]) ->
    is_integer(N) andalso (N>=0)
    andalso (N<State#robots.num_naves)
    andalso is_integer(W)
    andalso (W>=0);
pre([{exit,[_R,N,W]},State]) ->
    is_integer(N) andalso (N>=0)
    andalso (N<State#robots.num_naves)
    andalso is_integer(W)
    andalso (W>=0).
```
Appendix D. Testing implementations

cpre({enter,[_R,N,W]},State) ->
(weight(N,State)) + W <= State#robots.max_weight;
cpre({exit,[_R,N,W]},State) ->
(N==(State#robots.num_naves-1)) or else (not (occupied(N+1,State))).
post({enter,[_R,N,W]},State) ->
if N==0 ->
    add_weight(W,N,State);
    true ->
    add_weight(W,N,remove_robot(N,State))
end;
post({exit,[_R,N,W]},State) ->
if N==State#robots.num_naves-1 ->
    add_weight(-W,N,State);
    true ->
    add_weight(-W,N,add_robot(N+1,State))
end.
add_weight(W,N,State) ->
    {_,OldWeight} = lists:keyfind(N,1,State#robots.warehouses),
    State#robots{warehouses=lists:keystore(N,1,State#robots.warehouses,
        {N,OldWeight+W})}.
remove_robot(N,State) ->
    State#robots{corridors=lists:keystore(N,1,State#robots.corridors,
        {N,false})}.
add_robot(N,State) ->
    State#robots{corridors=lists:keystore(N,1,State#robots.corridors,
        {N,true})}.
weight(N,State) ->
    {_,Weight} = lists:keyfind(N,1,State#robots.warehouses),
    Weight.
occupied(N,State) ->
    {_,Occupied} = lists:keystore(N,1,State#robots.corridors),
    Occupied.

D.2 The Permissive Scheduling Policy

-module(always).
-export([init/2,waiting/3,
    enabled/4,post_waiting/4]).
init(_,_) ->
    void.
D.2. The Permissive Scheduling Policy

9 waiting(_Call,State,_) ->
10 {void,State}.
11
enabled(_Call,_CInfo,_State,_) ->
12 true.
13
post_waiting(_Call,_CInfo,State,_) ->
14 State.
Bibliography


BIBLIOGRAPHY


