UNIVERSITY OF SOUTHAMPTON

The Automated Translation of Integrated Formal Specifications into Concurrent Programs

by

Letu Yang

A thesis submitted in partial fulfillment for the degree of Doctor of Philosophy

in the

Faculty of Engineering, Science and Mathematics

School of Electronics and Computer Science

September 2008

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS
SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Doctor of Philosophy

by Letu Yang

The ProB model checker [LB03] provides tool support for an integrated formal specification approach, which combines the state-based B specification language [Abr96] with the event-based process algebra CSP [Hoa78]. The JCSP package [WM00b] presents a concurrent Java implementation for CSP/*occam*.

In this thesis, we present a developing strategy for implementing such a combined specification as a concurrent Java program. The combined semantics in ProB is flexible and ideal for model checking, but is too abstract to be implemented in programming languages. Also, although the JCSP package gave us significant inspiration for implementing formal specifications in Java, we argue that it is not suitable for directly implementing the combined semantics in ProB. Therefore, we started with defining a restricted semantics from the original one in ProB. Then we developed a new Java package, *JCSProB*, for implementing the restricted semantics in Java. The *JCSProB* package implements multi-way synchronization with choice for the combined B and CSP event, as well as a new multi-threading mechanism at process level. Also, a GUI sub-package is designed for constructing GUI programs for *JCSProB* to allow user interaction and run-time assertion checking. A set of translation rules relates the integrated formal models to Java and *JCSProB*, and we also implement these rules in an automated translation tool for automatically generating Java programs from these models.

To demonstrate and exercise the tool, several B/CSP models, varying both in syntactic structure and behavioural properties, are translated by the tool. The models manifest the presence and absence of various safety, deadlock, and fairness properties; the generated Java code is shown to faithfully reproduce them. Run-time safety and fairness assertion checking is also demonstrated. We also experimented with composition and decomposition on several combined models, as well as the Java programs generated from them. Composition techniques can help the user to develop large distributed systems, and can significantly improve the scalability of the development of the combined models of ProB.

Contents

List of Figures

List of Tables

Acknowledgements

First of all, I would like to thank my supervisor, Dr. Michael R. Poppleton, for his supervision and academic advice during my PhD study. Without his kind encouragement and constant support, I would never have finished. I am also thankful to Dr. Denis A. Nicole, my second-supervisor, who was always there to provide valuable advice. I thank Prof. Michael Leuschel for his supervision on the first year, and Prof. Michael J. Butler for his advice on my research topic.

I am grateful to all the rest of the academic and support staff of the Dependable Systems & Software Engineering group at the University of Southampton. Much respect to my officemates, and hopefully still friends, Andrew Edmunds, Divakar Yadav, Elisabeth Ball, Tossaporn Joochim, Nishadi De Silva, and Edwards Turner for putting up with me for almost four years. Also thanks to Prof. Peter Welch and Dr. Neil Evans for their advice on my research.

Lots of thanks goes to my friends Kan Huang, Ziheng Zhou and Xiaoli Li. They may have no idea about what formal methods are, but they still contributed to this thesis from talking jokes, cooking delicious food, and playing football with me.

Finally, I have to say 'thank you' to all my and family, particularly my Mum and Dad; and most importantly of all, my dear girlfriend Melody, who proclaimed herself as my third-supervisor, for everything.

To my dear mum, Yuan Ren ...

Chapter 1

Introduction

> *"Program testing can be used to show the presence of bugs, but never to show their absence. The only effective way to raise the confidence level of a program significantly is to give a convincing proof of its correctness."*
>
> *– Edsger W. Dijkstra*

Although there are countless computer systems running all around the world, the most serious issue in computer programming is the correctness of these systems. As both hardware and software systems have been rapidly growing in scale and functionality, it is harder and harder to identify or avoid errors in the development of computer systems. One significant aim of software engineering approaches is reducing errors and improving the reliability of systems. Techniques such as code modularization and code reuse, can help the developers to reduce the possibility of introducing errors in some sense. Familiar testing approaches can improve programs by detecting errors in the programs, however it is unlikely for testing approaches to cover all the examples in a large scale system to prove the correctness.

In [Dij65], Dijkstra suggests a definition of the correctness for programs, and an alternative approach for providing correctness to programs by establishing mathematical proofs. In [Flo67], Floyd provides formal definitions of the meanings of programming languages by defining them in flowcharts, and modes for verifying the flowcharts. Hoare [Hoa69] extends Floyd's work and establishes the famous *Hoare Logic*, which is a set of logical rules in order to reason about the correctness of computer programs with the rigour of mathematical logic.

However, directly applying mathematical proof rules to existing programs is apparently difficult and time-consuming. In [Dij68], Dijkstra introduces a constructive approach for developing programs. Rather than proving an existing program, it aims to develop a program in a sequence of steps. At each of these steps, proof is produced for the program at that stage. In [Wir71], Niklaus Wirth introduces stepwise refinement as a methodical

approach to computer programming. In an iterative step-by-step development process, a initial abstract and correct program can be developed into a more concrete version, and refinement rules can prove the later version is a correct refinement of the initial program. If this is done, finally, programmers end up with a proved program which is also concrete enough for execution. The stepwise refinement provides a realisation of mathematically proof of programs suggested by Dijkstra.

Concurrency is an important property of computer systems, in which multiple processes are running simultaneously and interacting with each other. A number of mathematically based concurrency models, such as *Petri nets* [Pet81] and *actor model* [HBS73], have been established for modeling and reasoning about concurrent systems. Process algebras, such as CSP [Hoa78], CCS [Mil80] and π-calculus [Mil99], also provide high-level specification methods for modelling concurrent systems, as well as algebraic rules for analyzing and reasoning concurrency properties in their specifications.

However, although formal concurrency models have existed for more than forty years, many conventional programming languages, e.g. Java, do not benefit from them. Concurrency in multi-threaded Java programming has always been seen as a problematic area [Pug00], to the extent that expert practitioner advice has been to avoid it where possible [MW00]. Safety properties have been made more tractable by the provision of a common cross-platform Java Memory Model [MPA05]. However, as the concurrency model of Java programs is described in natural language, it is still difficult to detect and avoid liveness problems in concurrent programs. The difficulty of concurrency in programming languages motivated the development of applying existing formal concurrency model to programming. One of the difficulties is that the formal concurrency models we mentioned above are based on the message passing mechanism, while the concurrency models of most computer systems and programming languages are based on shared memory. How to connect the two concurrency mechanisms is the key issue here.

1.1 Formal Methods, State- and Event- based, and Their Integration

Formal methods, which provide mathematically based techniques for software and hardware development, can be used to design, develop and prove the correctness of computer programs in a systematic manner. A typical formal methods approach introduces a formal specification language. Using the formal language, developers can produce mathematically-based formal models for computer systems. A formal method should also include formal development techniques for models described in the formal specification language. The development techniques of a formal method approach should include proof theories or verification techniques for mathematically proving the correctness of

these formal models. A good formal method should also come with a development strategy from abstract models towards implementations. Automated tools support for the development technique is also essential for formal method approaches, as many previous criticisms on formal methods concerned the lack of tool support.

In the past thirty years, many formal techniques have been developed for specifying system models and verifying their correctness. Generally, there are two kinds of formal techniques for specifying system model, which focus on different aspects of systems.

The state-based approaches, such as VDM [Jon90], Z [PST96] and B [Abr96, Sch01], focus on the data aspect of systems, such as data definitions and data transitions. Such a model can be viewed as a labelled transition system(LTS) [Mil89]. A Labelled Transition System (LTS) is a structure $(S, A, \rightarrow)$ with a set of states S, a set of actions A, and the transition relation $\rightarrow \subseteq S \times A \times S$. A state-based formal model is defined on the system states, where a state is an assignment of values to some set of state variables. It also needs to provide definitions of state actions, which changes the values of state variables with data substitutions and moves the system into a new state. At the same time, whether such an action is enabled or not depends on its guard, which is a conditional predicate on the state variables.

A state-based approach usually employs some form of stepwise refinement technique. Models at various abstract levels of a system are related by some formal refinement rules. Various tool sets have also been developed for these methods, e.g. *IFAD VDM-SL toolbox* [ELL94] for VDM, the *Community Z tools* (CZT) [MU05] for Z, and *Atelier B* [Ate01], ProB [LB03] for B.

Although state-based approaches are ideal for modelling the data aspect of systems, only a single-step atomic behaviour can be specified. It lacks expression for modelling behaviors of the whole system. For all the actions in a state-based model, the occurrence of actions is only restricted by precondition guards on these actions. The request for *divergence freedom* makes it hard to express a behaviour comprising a sequence of actions. Especially, in cases when the behaviour of a system turns complex, 'program counters', which are special state variables, need to be introduced to the model. Additional preconditions on these variables need to be added to actions for preserving a specific execution order of a number of actions. However, although additional variables and precondition guards can restrict the execution of actions, this approach still lacks an explicit system level view on system behaviours. This is apparently less expressive, not very easy for modelling. Also, it is usually not very easy to express or reason about behaviour properties using state-based approaches.

Event-based approaches, such as CSP [Hoa85, Ros98, Sch00] and CCS [Mil80, Mil89], focus on the behaviors of systems. There is also tool support for these approaches, for example, the FDR tool [For03] supports model checking for CSP specifications. Event-based approaches represent coarse-grained concurrency of programs. They focus

on behaviours of processes and interaction between processes, instead of the state of programs or data transitions on the state. An event-based model views a system as sequences of stateless actions the system may engage in. A sequence of events is modelled as a process, which can be defined using process operators, e.g. parallel composition, external choice. Shared events can be used among different processes for interaction and communication. Additionally, one event-based approach normally provides well defined formal semantics on behaviour properties, for example, the trace and failure-divergence semantics of *CSP*.

With these facilities mentioned above, event-based approaches are very good at specifying a high-level abstract model of system behaviours. However, it is usually not convenient to model the data aspect of systems using these approaches. A process maintains its own state and state variables, although the supported data types are usually very limited. In many event-base approaches, the data definition of these variables are not explicitly declared. It is very common for a process to use communicated data without knowing its data types.

For the whole system model, there is no global state. Also, as the events in an event-based model are regarded as stateless, it is not easy to specify complex state changes for the process variables. For high-level design of system behaviours and strategies, it may be possible to ignore the detailed data of systems. But when going to detailed design, especially for some data-intense systems, the expressivity for modelling data definition and transition is important. Furthermore, system properties concerning data safety can be very significant for many systems, and should be addressed in system models explicitly.

Either state- or event- based specification is a mathematical abstraction based on one perspective of the system properties. Although more detail of systems can be added in through a stepwise developing process, using a partial abstraction of a system to construct a system model still leaves out some properties of the system, or being less expressive on these properties. Directly modelling and reasoning temporal properties in a state-based model is difficult, also, defining and analysing the state of a globally shared or complex data in an event-based model can be difficult as well. However, developing some large-scale systems usually demands the expressivity of modelling both data and event features of systems.

In recent years, many efforts have been made to integrate the two kinds of formal specifications. These efforts include the integrations of CCS and Z [Gal96], CSP and Z [MS01], CSP and OZ [Fis97a, Fis00], CSP and B [ST03, But99, BL05]. In [BL05], an approach to combine B and CSP specification [1] is presented. The combined approach is also implemented in the ProB model checker [LB03] for automatic verification. The

[1] We will call this notation B+CSP for shorthand

CSP part of the combined specification describes the sequence of the system behaviors, while the B part specifies the data transitions inside these behaviours.

1.2 Formalizing Concurrent Java Programming

Formal approaches for modeling and developing concurrent computer systems, such as CSP [Hoa85], CCS [Mil80, Mil89], and occam [RH88, RGS94a, RGS94b], have been in existence for more than thirty years. Many research projects [WB04, CKK+00, BS05] and a number of real world systems [MST92, Law04, Pel04]have been developed from them. However, most programming languages in industry, which support concurrency, still lack formally defined concurrency models to make the development of such systems more reliable and tractable.

The concurrency model of Java is a multi-threading shared-memory model. Inside a concurrent Java program, multiple threads are running within a single process, and share the process's resource. When different threads try to access a shared data, Java offers the *monitor* mechanism to make sure when one thread holding the monitor, no other threads can execute any region of code protected by the same monitor lock.

The Java language has a painful history as it lacks explicit and formal definitions of its concurrency model. Before Java 5.0, the JMM (Java Memory Model) didn't explicitly define the read/write order that needs to be preserved in the memory model. This confused the developers of JVMs (Java Virtual Machines). The different JVMs developed under the old JMM could represent different behaviours, and lead to different results from running the same piece of Java code. To address this issue, Java 5.0 and the third version of the Java language specification had to redefine a new JMM.

Although the newly defined JMM clarified the safety of read/write orders at memory level, the Java concurrency model is still expressed in a natural language. It does not provide any method for evaluating general concurrency properties, such as safety, liveness and fairness. Developing a concurrent system under the Java concurrency model cannot guarantee the correctness of such programs. It still totally depends on developers' skills and experience to avoid concurrency problems.

Therefore, many approaches have been attempted to formalize the development of concurrent Java systems. Formal analysis techniques have been applied to concurrent Java programs. JML [LPC+05] and Jassda [BM02] provide strategies to add assertions to Java programs, and employ runtime verification techniques to check the assertions (There are also static analysis and verification tools for JML). Magee and Kramer [MK99] introduce a process algebra language, FSP (Finite State Processes), and provide a formal concurrency model for developing concurrent Java programs. JCSP [WBM+07] is a Java library offering the *occam* concurrency model, which is a low-level implementation

language of CSP. JCSP implements the main *occam* structures, such as process and channel, as well as key *occam* concurrency features, such as parallel, external choice and sequential composition, in various Java interfaces and classes. It bridges the gap between specification and implementation. With all the Java facility components in the JCSP package, developers can easily construct a concurrent Java program from its *occam* specification. All these approaches try to bring a formally defined concurrency model to concurrent Java programming. Such kind of models should provide a formal modelling language for constructing concurrency model, as well as development and verification techniques for building and reasoning Java programs.

1.3 Rationale for the PhD

A major criticism of formal methods is that many formal approaches lack industrial applications. The development of many formal approaches still largely rely on manual proof, and most lack substantial proof tool support, making system development with formal methods very hard for most developers.

The B method employs a stepwise refinement development technique. Both the B prover, B-Toolkit [BCo01], and the model checking tool, ProB [LB03], support this refinement development process for B method. The final refinement model would be a very concrete model, which is specified using a concrete $B0$ subset [Cle01] of the language. As the semantics of the $B0$ language is very similar to that of many conventional programming languages, it would not be very difficult to implement a final refinement of B method in a programming language. However composition and decomposition of B machines are still open research area, and it is not very easy to model or implement interactions and communications between distributed systems using B method. Most research on applying the B method in real world examples are either focusing well-defined small subsystems, or only developing models at abstract level. Therefore, the applications to which B development can be applied are still restricted. How to use the B method for developing a large, distributed system from design to implementation is still an ongoing research.

On the other hand, JCSP provides a Java implementation for the CSP specification. Although a manually constructed JCSP program can have a formal CSP model as its guidance, it cannot be systematically proved that the program correctly implements the formal model. Manual efforts involved here can be problematic. Also, the data manually introduced into the program usually cannot be easily specified in the CSP model. In some cases, such data can affect the behaviours of systems. An automated translation would represent a very useful contribution for JCSP. However, some reported experience [RRS03] has shown that, for automated translation, due to the lack of data expressivity

in CSP, it is sometimes not easy to directly use the Java programs translated from CSP. Usually, more data information still needs to be manually added.

Therefore, it would be an interesting contribution to implement integrated specifications. And with the help of existing techniques, e.g. B0 and JCSP, it could be very possible to develop an implementation strategy, as well as automated tool support, for the B+CSP model [BL05]. The implementation strategy, including the Java implementation of B+CSP semantics and the translation tool, should be carefully developed as their correctness is vital for this work. Tests would be applied to practically validate correctness of this work. Further work will target the formal verifications of the correctness of the Java implementation package and the translation.

Currently, the refinement rules for the integrated B and CSP specification are still not available. Although ProB can practically check the trace refinement between integrated models, a formal refinement proof method is not available. This means there is no development technique for building refinement models from the abstract one. Therefore, although this work mainly focuses on implementing concrete models, it would also support abstract semantics of the integrated specification languages. For an abstract model, the generated programs from the translation tool can either be used as a quick implementation, or be regarded as an animation mechanism for the model. For a concrete model, this implementation strategy would provide a prototype of the system.

Through this work, we try to develop an implementation strategy, as well as tool support, for integrated B and CSP specifications. Also, finding out an appropriate combination of B and CSP for developing large distributed systems is another important target.

1.4 Contributions

The first contribution of this work is a definition of the restricted semantics, based upon the existing semantics in ProB. The combined B+CSP specification language includes almost all the notations from both B and CSP specification languages. It is a large specification language with corresponding abstract semantics, which provide a very flexible way for specifying systems. However, some of the abstract semantics are very difficult to directly implement. Therefore, we develop a restricted semantics, which is concrete enough to be implemented into Java programs, and abstract enough to support modelling most general systems.

The second contribution of this work is implementing the restricted semantics in Java. Although JCSP has implemented *occam-π*, the semantics and concurrency model of B+CSP are different from those of *occam-π*. This means that we need to implement B+CSP semantics in a new Java package, *JCSProB*. In the development, we regard the structure of the JCSP package as a significant inspiration, and implement the B+CSP

concurrency model with similar process-channel structure to JCSP. A big difference between *JCSProB* and *JCSP* is that the new package supports combined B+CSP events, which include data changes, whereas *JCSP* communication channels are stateless [2]. The *JCSProB* package consists of new integrated event classes, new process classes, and a GUI sub-package. The GUI package allows users to interact with the generated Java programs, and it also provides interfaces for runtime assertion checking.

The development of translation rules and a translation tool is the third contribution of this work. Based on the implementation package, we formally define a set of translation rules to convert an useful and deterministic subset of the B+CSP specification to Java code. To make the translation more effective and stable, an automated translation tool is constructed as a functional component of the ProB tool. As the integrated specification includes both state- and event based views of the system, the generated Java code from the translation normally does not require manual modification (as other translation tools do, e.g. [RRS03]), and can be executed directly.

The above three contributions provide a development strategy with tooling support for the combined B+CSP models. Finally, the fourth contribution is to apply the strategy to some example models to evaluate the development of this work, and practically test and validate the Java implementation package and the translation. We also use the examples for evaluating and improving scalability of this work. This experimentation work includes modelling the example with the B+CSP specification, model checking the model in ProB, automatically translating the model to a Java application, testing and experimentally evaluating the generated Java programs. During the evaluation, we found the restricted B+CSP semantics used in this work is very close to the semantics of CSP‖B approach [ST03]. The composition strategy used in CSP‖B [STE05] can be used to compose B+CSP models. It is also possible for users of CSP‖B to make use of the JCSProB package and the translation tool.

1.5 Outline

This thesis starts with an introduction in Chapter 2 to state- and event- based formal methods, and the existing approaches for integrating them. The B method and CSP approach are introduced in Section 2.1 and Section 2.2. In Section 2.3, several integrated formal approaches are discussed and compared.

[2] JCSP provides Call channels. These are events from the caller's point of view whose acceptance is an "extended rendezvous" in which any state change, in the acceptor process, can be programmed. This is still different from the combined B+CSP event, in which any state change happens on the B machine

In Chapter 3, we discuss the problems of developing concurrent programs, especially in Java. In Section 3.2, various approaches to formalizing the development of Java concurrency are discussed. In Section 3.3, the JCSP package, which is important inspiration of this work, is introduced.

In Chapter 4, we present the restricted B+CSP semantics used in this work, which is the first objective of the development. In the discussion, we will first introduce the B+CSP specification language in Section 4.1, and operational semantics of ProB in Section 4.2. Then in Section 4.3, 4.4, and 4.5, the restricted semantics and its concurrency model are discussed.

In Chapter 5, the Java implementation of the restricted semantics is presented. As the new JCSProB package is inspired by and developed from JCSP, in Section 5.1, the reason for developing the new package instead of using the existing JCSP package is discussed. The JCSProB package includes implementation of guarded multi-way synchronization (in Section 5.4), and new implementation of multi-threading mechanism and recursions (in Section 5.5). To interact and control the Java programs constructed using JCSProB, we also provide a GUI package for building an user interface for users. This feature is presented in Section 5.7.

Chapter 6 describes the automatic translation from combined B+CSP specification to Java programs. In Section 6.1 some translation rules are presented and discussed. More translation rules can be found in Appendix A. In Section 6.2 the translation tool is presented.

In Chapter 7, the experimentation in this work is presented. Several example systems are modeled and implemented using this developing process, and the generated Java programs are tested and evaluated.

Finally in Chapter 8, we give the conclusion of this work, and propose some possible future directions based on this work.

Chapter 2

Integrated Formal Methods

Starting with the pioneering works from Dijkstra, Floyd, Hoare, and other computer scientists in 1960s, Formal Methods have been in existence for more than forty years, and many formal specification languages and development methodologies have been developed. However, formal approaches still have not been widely accepted or applied. A real system normally consists of many aspects of features, such as data, state, behaviour, and communication. One criticism on formal methods is that many formal approaches only focus on one aspect of views on systems, which limits their expressivity for large, complex systems. A restricted view of systems makes some formal methods lacking the expressivity to add enough details for constructing systems. Therefore, some formal method approaches may have sound and well-defined logic and specification languages, but using them to construct and implement complex systems can be very hard.

In this chapter, we introduce the state- and event based formal approaches, on which this work is based: the B method [Abr96] (Section 2.1), and CSP [Hoa85, Sch00, Ros98] (Section 2.2). In Section 2.3, we review several existing *integrated formal methods* approaches , which provide solutions for resolving the problem of restricted views on systems by combining different perspectives. Several integration approaches are discussed and compared. Finally, in Section 2.3.2, we focus on the combined B and CSP specification in ProB [BL05], which is selected for this work.

2.1 The B Method

The B method, developed by Jean-Raymond Abrial [Abr96], is a state-based formal approach for constructing computer systems. Jean-Raymond Abrial, who is also the inventer of the Z method [PST96], developed the B method based on successful projects [HK91] with the Z method.

It aims to cover the project life cycle from design to code generation. The B method is based on a version of first-order logic and set theory specifically tailored for it. The specification language of B method is defined in a form of *Abstract Machine Notation* (AMN), which is based ultimately on Dijkstras *guarded command language* [Dij97], and is very much influenced by Back's *action systems* [BKS83a]. The data and invariant of a machine is specified using predicate logic and set theory. In specification, the AMN and *Generalised Substitutions* are connected by the standard language of first-order logic and a restricted kind of set theory. Logically, a B machine can be understood as being a composition on four layers:

- *Level 0*, first order logic with equality
- *Level 1*, a typed set theory
- *Level 2*, the *Generalised Substitutions Language*
- *Level 3*, the AMN

2.1.1 AMN and *wp*

In the B method, an abstract machine is a specification of a system. The system specified by a B machine can be a part of a bigger software system. An abstract machine defines the state of a system based on a number of state variables, and invariants on these variables, which may type the variables and further constrain the values.

Operations modify the state under the constraints of the invariant. They can take zero or more parameters and return values. These are identifiers distinct from the state variables of the machine. Operations of a machine are specified as AMN substitutions. The semantics of AMN substitutions is given by *Generalised Substitutions*, modelled after Dijkstra's weakest precondition calculus [Dij97] and its later generalizations by Nelson [Nel89]. The weakest precondition is defined in the *wp-calculus* as $[S]P$. S is a statement of AMN, which may manipulate system states of the machine. The *postcondition* P of S is a predicate, which defines the set of states required to be true after executing S. Then, the notation $[S]P$ represents a precondition, which defines a maximal set of initial states from which after executing S, the *postcondition* P is guaranteed to be true. As $[S]P$ includes all the possible states that can achieve P after executing S, it is the weakest precondition. An operation of a machine can be regarded as a guarded and atomic command of Dijkstras *guarded command language*. The execution of a statement S is constrained by its precondition $[S]P$, and calling outside $[S]P$ is allowed but undefined.

Figure 2.1 demonstrates a very simple lift machine. An abstract B machine is defined under the header name **MACHINE**, which defines the identifier for the machine. Under the keyword **VARIABLES**, the machine variables are declared. The **INVARIANT**

clause provides all the information about the state variables, including their types and the values they could reach in this system. The initial state of the system is defined in the **INITIALISATION** clause. The operations of a machine is defined in the **OPERATIONS** clauses. All the B operations of a B machine do not happen in parallel, which means at a time, only one B operation can progress. The data substitutions inside an operation may change the state of the machine. The right-side references of B state variables refer to their pre-operation state, while the left-side references refer to the new state. In this case, the operation *inc* has a precondition $level < 10$ on its data assignment substitutions. When the precondition is not satisfied here, processing the data transitions would not reach any meaningful state in this machine.

MACHINE *lift*
VARIABLES *level*
INVARIANT $level : NAT \ \& \ level \geq 0 \ \& \ level \leq 10$
INITIALISATION $level := 1$
OPERATIONS
    $inc =$ ***PRE*** $level < 10$ ***THEN*** $level := level + 1$ ***END***;
    $dec =$ ***PRE*** $level > 0$ ***THEN*** $level := level - 1$ ***END***
END

FIGURE 2.1: An example of B machines: lift

As well as clauses defined above, abstract machines can also define given set types under the **SETS** clause. They are nonempty finite sets including a number of unique elements, and their elements can be enumerated. Constants can also be declared under the **CONSTANTS** clause. The types of the constants must be given in the **PROPERTIES** clause. Books from Abrial [Abr96] and Schneider [Sch01] are excellent text book on the B method for further reading.

2.1.2 The Development of the B Method

As a formal development methodology, the B method also provides an incremental development process, which is based on the *refinement calculus* [Bac80, HHS86, Mor88]. The refinement calculus provides a stepwise refinement method of program construction. A system specified in AMN is at a single level of abstraction of the system, which can be developed by adding details. The idea of stepwise refinement method starts with a very abstract model of a system. Details are gradually added to this first model by building a sequence of more concrete ones. The final production of this refinement procedure is a concrete *implementation* model.

Two successive models must preserve the relationship defined by the *refinement calculus* $\sqsubseteq$. A substitution S_1 is refined by a substitution S_2 ($S_1 \sqsubseteq S_2$), if any specification satisfied by S_1 is also satisfied by S_2. If P is the postcondition of S_1 and $[S_1]P$ holds, the predicate $[S_2]P$ for the refinement substitution S_2 should also hold. The refinement S_2 is either more deterministic than S_1, or has a weaker precondition than S_1.

The consistency of an abstract machine can be verified by proving proof obligations about it. The proving of proof obligations is traditionally called discharging them. A B machine, refinement, and composed machines(**INCLUDE** and **SEE** etc.) all have their own proof obligations. For operations, giving the invariants of two models as I and J, the proof obligation requires that the execution of S_2 must be matched by some execution of S_1, which means $[S_2]\neg[S_1]\neg J$ must be true at these states. The proof obligation is normally written as:

$$I \wedge J \wedge P \Rightarrow [S_2]\neg[S_1]\neg J$$

$\neg[S_1]\neg J$ means that not all transitions of S_1 make J false, which implies that there are some transitions of S_1 guarantee J to be true. So the left part of the formula means for any S_2, there are some transitions of S_1 to guarantee J.

To prove the refinement obligation for operations, gluing invariant *J(x,y)* sometimes need to be introduced to link the state variable x of the abstract machine, and the state variable y of the concrete machine.

The interactive proving tools for the B method, such as *Atelier B* [Ate01] and *B-Toolkit* [BCo01], can be used to help developers to developing systems with the B method.

2.1.3 The B0 Language

Abrial's book [Abr96] only gives an abstract syntax for his notation, and only gives faint hints on the concrete syntax. The concrete syntax used by the B-Toolkit or Atelier B is not described at all by Abrial. A definition of the $B0$ language can be find in [Cle01]. $B0$ is a concrete subset of the B specification language, describing operations and data of implementations. It only presents concrete data using concrete constants, concrete variables, operation input and output parameters, machine parameters, local variables, and enumerated sets. Conditions and terms are only defined on concrete data. Only concrete substitutions without non-determinism, which are called *instructions*, are allowed in $B0$. Instructions are used both in the initialization and the operations.

2.1.4 Model Checking and PROB

Formally proving a B model can be difficult and time-consuming. Model checking [EMCP99] provides an alternative technique to verify developed formal models. It tests whether a given formula in the propositional logic is satisfied by a given model by exploring the state space of the model. Model checking tools all face a scaling problem, commonly known as the *state explosion problem* [CGJ+01].

ProB [LB03] is an animation and model checking tool for the B method. The tool covers a large part of the B syntax and semantics. As an animation tool, ProB allows user to manually drive B models. As a finite state model checker, ProB supports automated consistency checking of B machines, and refinement checking between B machines:

- In consistency checking, ProB can check safety properties, such as violations of invariant and deadlock of the system. It can identify counter examples of required properties and provide traces leading to them. This information can help the developer to improve system specifications.
- In refinement checking, ProB automatically explores and compares the state spaces of two B models, and can find the exceptions of refinement.

The symmetry reduction technique [TB06] used in ProB can reduce the size state space and improve its model checking performance.

Furthermore, ProB also supports model checking for CSP specifications [Hoa85] or event combined B+CSP specifications [BL05]. We discuss this later in this chapter.

2.2 Communicating Sequential Processes

CSP (*Communicating Sequential Processes*) is a process algebra proposed by Hoare in [Hoa85]. It is a well-known event-based formal language for modelling concurrent systems. A concurrent system is viewed as a set of independent processes. Processes communicate with the environment and processes via atomic and stateless events.

2.2.1 A Simple Introduction to CSP Grammar

The set of all events that a process can engage in is called its *Alphabet*, usually named as αP. The behaviour of the process is defined in sequences of the events as combination with basic processes and process operators.

Basic processes in CSP includes *STOP* and *SKIP*. *STOP* means the process is in a state where no events can be engaged, and the process would stay at this state for ever. *SKIP* denotes a successful termination of a process. The process operator $\rightarrow$ used above describes the sequencing of events. An expression $a \rightarrow P$ means the process first engages in the event a, and then performs as process P.

The parallel composition of process is specified using process operator $\|$. A parallel composition of two processes P_1 and P_2 is written as $P_1 \parallel P_2$. The two processes can synchronize on events in the intersection of the alphabets of them. However, in many cases, a process in a parallel composition may not want to communicate with other

processes on all the events in its alphabet. The *alphabetized parallel* operator defines the parallel composition of processes with explicit declaration of the events that may be involved in the composition. The event sets of two processes P_1 and P_2 involved in the parallel composition are A_1 and A_2, where $A_1 \subseteq \alpha P_1$ and $A_2 \subseteq \alpha P_2$. The alphabetized parallel is written $P_1 \ {}_{A_1}\|_{A_2} \ P_2$. In this definition, P_1 can only engage events in A_1, and P_2 can only engage events in A_2, and the two processes need to synchronize on events in $A_1 \cap A_2$. A generalized version of parallel is defined as $P_1 \underset{X}{\|} P_2$, where $X \subseteq A_1 \cap A_2$. When $X = \emptyset$, which means the two parallel processes have no event to synchronize, the two processes are said to *interleave* with each other. The parallel composition here is written using interleaving operator $|||$ as $P_1 \ ||| \ P_2$. Semantically, the termination event $\surd$ always needs to synchronize in parallel processes, even in the interleaving processes, although it does not explicitly appear in the definition. This means the parallel composition terminates only when all participating processes successfully terminate.

Sequential composition of processes runs the participating processes one after the other. Only after a process has successfully terminated, can the successor process start performing. Using the sequential composition operator ;, a sequential composition of two processes P_1 and P_2 is written $P_1; P_2$.

The choice operator introduces a number of possible action paths for a process. In *external choice*, the control over the choice is external to the process, while *internal choice* is resolved inside the process. An external choice in process P between two process paths P_1 and P_2 is written $P_1 \Box P_2$. If, and only if the first event of a process path is ready to progress, the process path can be considered as a candidate for selection. This kind of external choice is also called *guarded external choice*. *Internal choice* is also known as non-deterministic choice. The decision of an internal choice $P_1 \sqcap P_2$ is made internally in process P, which leaves the outside environment with no control over the choice.

Other CSP operators, including *renaming*, *hiding* and *interrupting*, can be found in [Hoa85] or [Ros98].

2.2.2 Denotational Semantics of CSP

In the original CSP [Hoa78], the semantics of CSP were not clearly described. The traces [Hoa80], failures [BHR84] and divergences [BR85] semantics were later given for reasoning about properties of concurrent systems. In Roscoe's book [Ros98], the denotational semantics of CSP are summarized and discussed.

The traces model T captures the behaviours of a process in a set of non-empty prefix-closed event traces. The all set of finite traces of a process P is written *traces(P)*. For example, when $P = a \rightarrow b \rightarrow STOP$, $traces(P) = \{<>, < a >, < a, b >\}$. A set of

algebraic rules are defined for computing trace sets over all the process operators under traces model. For example, the rule for prefixing is defined as:

$$traces(a \rightarrow P) = \{<>\} \cup \{< a >^\frown tr \mid tr \in traces(P)\}$$

However, although the traces model can capture a range of system behaviours of concurrent system, it still not expressive enough to cover all the semantics of CSP. For example, the traces model does not distinguish between internal and external choice:

$$traces(P_1 \Box P_2) = traces(P_2) \cup traces(P_2)$$
$$traces(P_1 \sqcap P_2) = traces(P_2) \cup traces(P_2)$$

That means the non-determinism in internal choice cannot be expressed in traces model. Furthermore, some safety liveness properties, e.g. deadlock, cannot be expressed with trace semantics. Therefore, failures and divergences models introduced in [BHR84] and [BR85] to express these properties.

The failures model D is defined to express the deadlock state of a process. A process P which can make no internal progress is said to be *stable*, written $P \downarrow$. If after executing a sequence of events defined in trace tr a stable process P refuses to engage a set of events X, the observation of (tr, X) is called a *stable failure* of P.

The set $failures(P)$ includes all the failures of the process P. When it is possible for a process P to perform an infinite sequence of internal events and never reach a stable state, it is said to be *divergent*, written $P \uparrow$. The divergences model D of a process P is defined as $divergences(P)$, which includes all the traces that can lead the process P into divergence.

The full semantic model of a process is thus defined as a triple (T, F, D). The refinement of CSP specifications is also defined upon this semantics:

- A process P_2 trace refines a process P_1 if all the traces of P_2 are also traces of P_1

  $$P_1 \sqsubseteq_T P_2 \equiv traces(P_2) \subseteq traces(P_1)$$

- A process P_2 failure refines a process P_1 if P_2 traces refines P_1, and all the failures of P_2 are also failures of P_1

  $$P_1 \sqsubseteq_F P_2 \equiv traces(P_2) \subseteq traces(P_1) \wedge failures(P_2) \subseteq failures(P_1)$$

- A process P_2 failure/divergences refines a process P_1 if P_2 failures refines P_1, and all the divergences of P_2 are also divergences of P_1.

  $$P_1 \sqsubseteq_F P_2 \equiv traces(P_2) \subseteq traces(P_1) \wedge failures(P_2) \subseteq failures(P_1)$$
  $$\wedge divergences(P_2) \subseteq divergences(P_1)$$

2.3 Integrated Formal Methods

2.3.1 Some IFM Approaches

The integrated formal methods discussed here are formal specifications which try to integrate state-based methods with event-based methods. Some other approaches, such as UMLB [SB06], Real-Time Object-Z [SH00], B/VDM [BMRA98], will not be discussed here. Also, several previous works on combining event- and state- based methods have been widely mentioned, for example $ZCCS$ [Gal96] and TCOZ [MD99], but in this chapter, we only discuss a selection of more recent work.

2.3.1.1 CSP-Z

In [Fis97b, Fis98], an integrated formal specification language CSP-Z is introduced. It combines the syntax and semantics of CSP and Z [PST96]. The syntax of the CSP part is fully preserved, while the Z syntax subset used in CSP-Z is different from the original Z syntax. It defines the CSP-Z language with semantics subsets of CSP and Z languages. A CSP-Z specification starts with the declaration of *channels*(external and internal).

A CSP-Z channel can be associated with a Z schema type. The system behaviours are defined in a group of CSP processes. The Z operations, which correspond to the channels described above, are defined with the data transitions in the operations. As explained above, a CSP-Z specification is a parallel combination of the CSP and the Z parts via the channel names, such that on the occurrence of a CSP channel the corresponding Z schema operation is activated. Since the CSP-Z specification language is a semantic integration of CSP and Z, the failure-divergence semantics of CSP is also inherited into CSP-Z.

In [MS98, MS01], a strategy for model checking a CSP-Z specification in the FDR model checker is proposed. As FDR only works on the machine readable CSP specifications, [MS98] gives a solution which converts the Z part of the $CSP - Z$ specification into the CSP part of the specification. A CSP-Z specification is divided into the CSP part, and a CSP process which performs the Z part operations. As the result, a CSP-Z specification is translated into a machine readable CSP specification, which can be automatically checked in FDR.

The limitation of [MS98, MS01] is that the final model used for model checking is purely in CSP. Since the CSP specification lacks a convenient method to express complex data and data transitions, the Z part of the CSP-Z specification is restricted to very simple data types and data transitions. The final model of the strategy, which is purely in CSP, can also be specified directly with CSP. In [MS02b], the continuing research

employs data independence and abstract interpretation techniques to modify the *CSP-Z* model, and makes it possible to use more general *CSP-Z* model for model checking.

2.3.1.2 CSP/OZ

CSP-OZ [Fis97a] continues the work of *CSP-Z* . It aims at integrating *CSP*, and *Object-Z*, which is an object-oriented extension of *Z* [DRS95]. In a *CSP-OZ* specification, the *CSP* failure-divergence semantics is used to guide *Object-Z* classes.

In a *CSP-OZ* specification, firstly, the *CSP* style channels are defined in a *CSP-OZ* specification. The *CSP* part of the specification defines a process with the keyword *main* to present the behavior of the *CSP-OZ* class. It makes use of the channels defined above, and can include some *CSP* operators, such as *parallel*, *hiding*, *interleaving* and *choice*. The *Object-Z* part starts with the definition of types and constants. Then the definition of the state and initial state schemas are given. The data transition is defined in *Object-Z* operations. Like *CSP-Z*, each *Object-Z* schema operation corresponds to a channel defined in the *CSP* part.

In order to perform model checking on the *CSP-OZ* specification, as there was no existing *CSP-OZ* model checker, [FW99] presents a strategy to translate the *CSP-OZ* specification into machine readable *CSP* and check it with *FDR*. The *Object-Z* part and the *CSP* part of the specification are translated into different *CSP* processes. While the *CSP* part of *CSP-OZ* is directly converted to the *CSP* processes, the *Object-Z* part, especially the operations, need to be expressed in the *CSP* syntax. One of the most serious issues of this strategy is that the data domain of the *Object-Z* part can be too large to check in *FDR*. In [Weh99], data abstraction techniques are used to reduce the complexity of property checking on *CSP-OZ* specification.

To associate the abstract *CSP-OZ* specification with programming languages, in [Fis99, BFMW01], *Jass* (Java with Assertions) is introduced as an intermediate language. *Jass* is an assertion language which is written in a Java source code file as comments. It is motivated by *Design by Contract* [Mey92], which is a lightweight formal technique that allows for dynamic run-time checks of specification violation. The assertions can be tested when the program is executed.

2.3.1.3 CSP ‖ B

CSP ‖ *B* [TS99b, TS00, ST02, ST03] is integration of the *B* method and *CSP*. A *CSP* ‖ *B* specification includes *B* machines, and the *CSP* controllers(or processes) for them, which are expressed with a subset of the *CSP* syntax. Each *B* machine M_i of the system corresponds to a *CSP* controller P_i. The *B* machines do not communicate with each other directly. They may only allow communicate through their respective

CSP controllers, and then their controllers can communicate with each other. For each B operation $w \longleftarrow e(v)$, there is a corresponding CSP control channel $e.v.w$ defined in its controller. The execution of a B operation is strictly guarded by its CSP controller.

Figure 2.2 shows a B‖CSP example provided in [ST05].

```
MACHINE Even

VARIABLES even

INVARIANT even : NAT & even mod 2 = 0

INITIALISATION even := 0

OPERATIONS
        evenput(nn) =  PRE nn : NAT & nn mod 2 = 0
                       THEN even := nn
                       END;

        nn <-- evenget = BEGIN nn := even END

END
```

```
MACHINE Odd

VARIABLES odd

INVARIANT odd : NAT & odd mod 2 = 1

INITIALISATION odd := 1

OPERATIONS
        oddput(nn) = PRE nn : NAT & nn mod 2 = 1
                     THEN odd := nn
                     END;

        nn <-- oddget = BEGIN nn := odd END

END
```

```
EvenCtrl = oddpass?Z -> evenput!(Z+1) ->
        evenget?W -> evenpass!W -> EvenCtrl ;;
```

```
OddCtrl = oddget?X -> oddpass!X ->
        evenpass?Y -> oddput!(Y+1) -> OddCtrl ;;
```

FIGURE 2.2: The specification of the Odd-Even example

The CSP processes *OddCtrl* and *EvenCtrl* define a system which recursively increases numbers through the system. The B machines, on the other hand, define the states of the system with variables *odd* and *even*. The machine *Odd* accepts and maintains only odd numbers, while machine *Even* accepts and maintains only even numbers. The CSP processes work as controllers of the B machines. The execution of B operations are driven by their CSP controllers.

In [ST03], an approach is proposed which verifies each structure(a B machine M_i and CSP controller P_i pair) separately, and then obtains the correctness of the whole system by composition. In [ST02], a strategy for model checking CSP controllers via the FDR tool is discussed. Firstly, a B machine and its controller are checked together in the *B-Toolkit* to prove they are divergence-free. Then all the CSP controllers are checked in the FDR tool to prove they are deadlock-free.

One of the problems of the $CSP \parallel B$ approach is that there are no refinement rules or proof for the integrated approach. The case study in [TSB03] tried to establish a development strategy for $CSP \parallel B$ models. However, the verification step for refinement consistency between an abstract model and its refinement only focus on using FDR to model check refinement between two CSP controllers. It still cannot formally prove or model check refinement between whole integrated models. Another problem is the verification step involves different tools and several separate verification steps, which can be hard for users to apply the development.

2.3.1.4 csp2B

In [But99], *csp*2*B*, an alternative approach to integrate *B* and *CSP* specifications, is presented. It defines a set of *csp*2*B* notations. A *csp*2*B* specification is encapsulated into a *B* machine like structure, but the main component of the machine is a kind of CSP description. In the machine, a group of *CSP* channels each of which correspond to target *B* operations are defined under the keyword **ALPHABET**. Under the keyword **PROCESS**, *CSP* processes are defined to specify behaviours of the system.

The *csp*2*B* translation tool converts a *csp*2*B* model into a B machine. In the target B machine, state variables are introduced as *program counters* to explicitly express state of *CSP* processes from the *csp*2*B* machine. A B operation in the B machine includes guards, which are defined upon these program counters, and substitutions, which update program counters. In this way, system behaviours specified in the *csp*2*B* model are retained in the generated B machine. Furthermore, using the **CONJOINS** statement, a *csp*2*B* model can be used to constrains the behaviour of a existing B machine. The *csp*2*B* tool can also add behaviour constraints from the *csp*2*B* model to the B machine.

As the target of this translation is a solely *B* machine, it has existing refinement rules and development strategy for development. The generated B machine can be analysed and verified within the *B* model checkers, e.g. ProB, or interactive B provers.

One restriction for the *csp*2*B* approach is that it is restricted on a single CSP and B pair. There is still no compositional development or verification techniques for it.

2.3.1.5 Circus

Circus [WC01] is an integrated formal specification language, which is based on *Z* and *CSP*. It supports stepwise refinement [CSW03]. That means its final refinement of the system model is close to a general programming language.

A *Circus* program consists of many *Circus Paragraphs*, each of which can be a *Z* paragraph, a CSP-like channel(or channel set) definition paragraph, or a process definition. A *Z* paragraph uses purely *Z* notations, while a channel or channel set definition is written in *CSP* style. The two specification languages are associated with each other in the definition of *Circus* processes. A *Circus* process is defined with a process name and a process specification. In a process specification, *Z* paragraph and *CSP* like action paragraph can be used cooperatively. The *Z* schema defines the states of the process, which include the data and data operations. The *CSP* part in the *Circus* process defines how the process act the *Z* schema inside. So far as we know, *Circus* is probably the only combined state- and event-based formal method, which provides refinement rules [WC02] for the combined specification.

In [OC04], a set of translation rules, which translate *Circus* models into JCSP [WM00b] programs, is presented. Based on this rule set, an automated translator [FC06] is developed.

2.3.2 B+CSP in PROB

[BL05] presents a new integration approach to combine B and CSP, B+CSP. A B+CSP model explicitly provides the state-based view of a system in the B specification, and the event-base view in the CSP processes.

The B part of the combined model is used to specify the data used in the system, and the manipulations on the data. It uses the full set of B method notations. The CSP part provides the execution flow of the B operations. Each B operation of a B machine has a corresponding CSP channel defined in the CSP part. The parameters of the B operations are also mapped to the input and output parameters of the CSP channel. The **MAIN** CSP process uses the channel definitions, and defines the behaviors of the system. The execution of the B operations is not only guarded by the precondition of the operation, but also strictly guarded by the **MAIN** process.

The CSP part can also define the desirable and undesirable behaviors of the system. The desirable behaviour is defined in the **GOAL** process, while the undesirable behavior is defined in the **ERROR** process. Both of them are checked as trace properties of the combined system model. When the operation sequence of the **GOAL** process is found in the system model, the property is satisfied and the trace is returned. The property of the **ERROR** process is satisfied when there is no trace of it.

The combination of the $B + CSP$ model is supported by the ProB tool. The B part of the model is parsed and interpreted into *Prolog* rules via *jbTool* [Bru01] and the Pillow package [CH01]. The CSP processes of the model are translated into *Prolog* rules via the CIA (CSP Interpreter and Animator) tool. When doing model checking and animation on the combined model, ProB interpreter combines the two sets of *Prolog* rules together based on the operational semantics of B+CSP, which is discussed in Section 4.2. An important feature for ProB is that it can perform not only consistency checking for a B+CSP model, but also trace refinement checking between combined B+CSP models.

Table 2.1 compares B+CSP with three other integrated formal methods mentioned above regarding their development techniques.

The B+CSP can be regarded as a practical extension of the *csp2B* approach. The integration of B and CSP models in ProB is very similar to the conjunction of *csp2B* and B models in *csp2B*. Both approaches use CSP specification to constrain the behaviour of the B machine. In *csp2B*, the combination of B and CSP *csp2B* approach explicitly translates the CSP specification into B state expressions, while in ProB, the combination

	CSP+B	csp2B	CSP ‖ B	*Circus*
Spec languages	B, CSP	B, csp2B	B, CSP	Z, CSP
Refinement rule	no	yes, B	no	yes, Circus
Model checking	ProB	ProB	ProB, FDR	no
Animation	ProB	no	no	no
Composition	no	no	yes	yes
Implementation	JCSProB	no	no	JCSP

Table 2.1: The comparison of four IFMs

is presented in model checking. The final B machine generated by the *csp2B* tool can either be formally proved using theorem provers, but also can be model checked by ProB. On the contrary, currently there are no refinement rules for B+CSP, it can only be verified via model checking. Additionally, both of the approaches are restricted on one B and CSP specification pair. Composition rules or proofs are still missing. An advantage for *csp2B* is the final B machine generated by *csp2B* tool is solely a B machine, which means the stepwise development and proving techniques of B method can be applied to it.

Comparing to CSP ‖ B, B+CSP is a more practical approach. CSP ‖ B provides a more theoretical basis and systematical strategies for development and composition, while B+CSP has a better tool support from the ProB tool for animation and model checking. However, the semantics of the two approaches can be very close, and many research [STE05] think it is even possible to connect the two approaches together.

Circus is also a more theoretical approach than B+CSP. A significant advantage for *Circus* is that it has a set refinement rules [WC02] for the combined specification, while other approaches mentioned above do not have this support. By applying these rules, a new refined model can be constructed from the abstract one. Although a model checking tool is reported under construction, currently it is still unavailable.

Chapter 3

Formalized Java Concurrency Development

"Java built-in support for threads is a double-edged sword."

– *Java Concurrency in Practice* [PGB+05]

The Java programming language supports a multi-threading concurrency mechanism in the core of the language. It also provides a series of classes to support Java concurrent programming. This feature helps Java developers to solve many problems, and many packages in JDK are developed using concurrency, e.g. AWT and Swing. However, at the same time, it is even more difficult to make concurrent programs correct than sequential programs. Concurrency issues, such as safety and deadlock, have existed since the idea of the concept of concurrency come into being. Concurrency in Java is also problematic.

Section 3.1 starts with an introduction to concurrent Java programming, and issues in Java concurrency. After that, we discuss several approaches on formalizing concurrent Java development in Section 3.2. Finally, in Section 3.3 we discuss the JCSP package, which is one of the main inspirations of our work.

3.1 Concurrent Java programming

Generally, there are two kinds of issues in concurrent Java programming.

Safety in Java programs consists of type safety and memory safety. Here we concern the safety which provides a consistent view of data. When multiple Java threads try to access the same data item, the developer has to make sure that these threads coordinate their access to the data so that all have a consistent view of the data without interfering

the others' changes. However, the previous version of the Java Memory Model (JMM), specified in Chapter 17 of [GJSB00], raised some concerns on the safety guarantee at memory level.

A memory model describes the relationship between data in a program and the low-level details of storing them to and retrieving them from memory in a real computer system. Consider a Java program fragment with two memory actions a and b, which change the state of data variables in the main memory.

... a; b; ...

Generally, action a should be processed before b in the program. However, the hardware system may permute the execution order for efficiency reasons. The actual execution order at hardware level or memory level can be b ; a. In certain circumstances, changing the processing order of the two operations will not effect the result of Java program. In those circumstances, the JVM may allow some of these reordering techniques in order to execute Java programs more efficiently. The JMM must explicitly specify in which occasions the execution order must be preserved as a ; b for all viewers of the memory, and in which occasions optimization techniques can be employed. Moreover, the JMM definition must be clear enough to be implemented on all platform without misunderstanding. If not, the different execution orders on different JVMs may lead to different results.

The previous version of JMM failed to deliver this clearly. In [Pug00], the problems caused by the implicit definition of JMM is discussed. In the new version of Java specification of JDK 1.5 [GJSB05], the Java Memory Model is revised. It redefines the semantics of synchronization, volatile variables, and final fields. Therefore, the memory level safety issues caused by JMM has been clarified.

The safety we are concerned with in this work is the concurrency safety in programs, not in the memory model. Inappropriate uses of Java concurrency primitives can cause safety problems, such as data races and deadlock, which can lead to unexpected results.

The other major issue in Java concurrency is liveness. Liveness problems can be caused by the Java concurrency primitives. The recommended synchronization pattern for a conditional wait [Jav] inside a Java monitor is shown in Figure 3.1.

Using the *while* loop with activation condition for synchronization may unnecessarily consume a lot of resource of computer systems. Furthermore, it can easily bring heavy overtaking or even starvation issues to the Java threads. The original wow-no-chicken example [Wel98] addressed the starvation issue of this strategy. The *notifyAll* method, widely used by many Java programmers, is very inefficient. It notifies all the waiting threads and lets them compete with each other for the shared resource. Only one thread can occupy the resource and progress. Apart from these problem in Java concurrency

```
synchronized(this){
   while(!<condition>) \*← overtaking*\
      wait();
   <assignment of shared data>
   notifyAll(); \*← inefficiency*\
}
```

FIGURE 3.1: Synchronization of Java Threads

primitives, liveness problems can also be easily introduced into concurrent Java programs by improper design from users.

For all the concurrency problems introduced by user design, Java provides no facility to address or avoid them. Developing correct concurrent Java programs mainly relies on manual skill and experience in development. [Goe03], [Goe04] and [Lea99] discuss concurrency issues in Java programming in detail. These disadvantages discourage some developers from using Java concurrency programming, especially for some large-scale systems where the concurrency model can be very complex. In [MW00], the developers of the Swing package even claimed: *If you can get away with it, avoid using threads.*

3.1.1 Concurrency in Java

The Java programming language supports shared-memory concurrency by the thread-monitor concept. A Java *thread* interface performs the basic process which represents independent control flow. The behavior of a Java thread is described in its *run* method.

The execution of a Java thread can be controlled or interfered by other threads using some methods provided by Java *thread*: *stop*, *suspend*, and *resume*. However, calling such methods to control the execution of a concurrent system is unsafe and deprecated. It is extremely difficult to control all the threads when there are a large number of them in a concurrent system. Therefore, Java provides a concurrency model based on the monitor concept to resolve the synchronization between threads. When more than one thread tries to access shared data in a *critical section*, which is marked with keyword *synchronized*, the lock of the critical section only allow one of these threads to access the protected section at a time. A thread uses the *wait* method to wait when it cannot exploit the critical section which it is already occupying. So it releases the monitor lock and hopes some other thread will acquire the monitor, fix things up for it and then wake the waiting thread. The user can either notify all the waiting threads using the *notifyAll* method, or notify only one of them using the *notify* method. In the *notifyAll* case, all the notified threads compete with each others to acquire the object lock.

The low-level monitor strategy of Java concurrent programming is not free from concurrency-related problems like data racing and deadlock. In Figure 3.1, some safety issues are

addressed using the recommended synchronisation pattern. Even then, the system remains exposed to liveness problems. The main concern is that the system developers have to identify all these concurrency problems by themselves. They need to use the Java synchronization primitives to construct a synchronization unit by themselves with concerns on the possible concurrency issues. As the concurrency strategy of a large-scale system can be very complicated, it is extremely difficult to build the system free from these issues. Furthermore, the efficiency of the concurrent system mostly depends on the design of synchronization. Even a building a simple semaphore unit can be inefficient or even cause deadlock. Therefore, building Java concurrent systems with the basic monitor primitives demands experienced skill from the developers.

3.1.2 J2SE 5.0

To avoid these problems, the new version of Java, JDK 5.0 introduced a new Java package *java.util.concurrency* [Goe04] to support higher level Java concurrent programming.

The new *util.concurrency* package also introduced some high-level thread-safe synchronization facilities to help in developing concurrent programs in Java. For example, the *Semaphore* class implements a classic Dijkstra counting semaphore. It has a certain number of permits, which can be obtained and released by threads. The Java threads with permits can access a shared resource and change the state of the shared resource. It changes the previous fashion of shared resource control which is described in Figure 3.1 with more efficient and safe class. Other new synchronization facilities include:

- `AbstractQueuedSynchronizer` class maintains synchronization state of Java threads. It replaces the *synchronizsed* block previously used in concurrent Java programming. It also maintains a FIFO queue for the blocked threads.
- `LockSupport` class blocks and unblocks threads. It replaces the *suspend()* and *resume()* method of Java *Thread* interface.
- `ConditionObject` class provides an alternative for the classic Java monitor-style synchronization by using the *Lock* interface of the new *concurrency* package.

Using the new concurrency primitives listed above, some new high-level facility classes are constructed. As these facility classes are carefully constructed and tested, using them to construct a concurrent system can prevent some fairness and efficiency issues. For example, the classical consumer-producer problem can be solved by Java monitor in Figure 3.2.

The consumer waits if the buffer is empty. After the producer added an object to the buffer, it notifies the consumer. Similarly, when a producer find the buffer is full, it

```
      //Consumer thread
      synchronized(buffer){
         while (buffer.size() == 0) {
            buffer.wait();
         }
         // -- Consume object from the buffer
   --
         notifyAll();
      }
      ...
      //Producer thread
      synchronized(buffer){
         while (buffer.size() == MAX) {
            buffer.wait();
         }
         // -- Produce object to buffer --
         notifyAll();
      }
```

FIGURE 3.2: Consumer-Producer Example: Java Monitor Solution

waits for the consumer to collect objects from the buffer, and send the notification. In Figure 3.1, the possible concurrency problems for the *wait-notify* approach are discussed.

The thread safe collection classes in J2SE 5.0 provides some straightforward implementations for thread-safe collections [Goe04]. The *java.util.concurrency* package provides a *BlockingQueue* interface and implementation classes, which is designed to be used primarily for producer-consumer queues. With this synchronization class, the consumer-producer problem can be easily and safely implemented in Figure 3.3.

```
   BlockingQueue q = new SomeQueueImplementation(size);
...
   //Consumer thread
   try {
      while(true) { consume(q.take()); }
   } catch (InterruptedException ex) { ...  handle
...}
   ...
   //Producer thread
   try {
      while(true) { q.put(produce()); }
   } catch (InterruptedException ex) { ...  handle
...}
```

FIGURE 3.3: Consumer-Producer Example: *BlockingQueue* Solution

The synchronization between the two threads are much easier and clearer in Figure 3.3. As the implementation classes of *BlockingQueue* are elaborately designed, it makes it

easier to write correct and thread-safe concurrent Java implementation. Carefully implementing these synchronizer classes improved the programming simplicity and scalability of concurrent Java program, and prevents a number of concurrency issues. Also, the Java built-in locks accessed with synchronization are not fair locks at all (See the wot-no-chicken example in Section 7.2). Instead, they provide weaker liveness guarantees that require that all threads will eventually acquire the lock. The low-level lock facilities in JDK 5.0, such as *ReentrantLock*, *Semaphore*, and *ReentrantReadWriteLock*, provide options to guarantee fairness to the lock [Goe04].

3.2 Formal Approaches to Java Concurrency

Even J2SE 5.0 with new Java higher level concurrency primitives is not free of concurrency problems. It also provides no checking or verification facilities to detect these problems. Moreover, the concurrency model is still defined only in natural language, and thus cannot be formally analyzed. The lack of a formal foundation for the Java concurrency model makes it difficult to analyze and resolve concurrency issues in constructing large-scale concurrent Java applications.

Many approaches have been proposed for providing formal foundations to Java concurrency. One important trend is using formal techniques to analyze and verify existing concurrent Java programs. The approaches for formal analyzing Java concurrency can be roughly divided into three kinds of techniques.

- **Static Analysis**. The static analysis approaches employ formal specification languages, such as Petri Net [LS03], CCS [Che00], CSP/FDR [Hoa78], occam/JCSP [WM00b, WM00a], Promela [DS98], JML [FLL+02], and CTL (computation tree logic) formulas [RS05], to model concurrent Java programs. The formal specifications derived from Java programs can be analyzed and verified by deductive analysis tools or model checking tools. The problem of this approach is that the abstraction from Java programs to formal specifications usually lacks of formal mapping rules and automated tool support. It is difficult here to fill the gap between Java programs to formal specifications with reliable formal connection. JPF1 (Java Path Finder) [HS00] is one of the exceptions. It provides automatical translation tools to translate a subset of Java language into Promela. The Promela programs can be automatical verified in SPIN model checker [Hol03].

- **Runtime Verification**. Runtime verification approaches are based on Meyer's *Design by Contract* concept [Mey92]. Assertion languages are introduced as pre-, postcondition, and invariants, which are used to specify the obligations that need to be satisfied during the execution of the Java programs. These assertions are inserted into the Java source code as comments, and can be evaluated and verified

when the programs are executed in the run-time verification tools. These runtime verification approaches are lightweight formal techniques which are easy to apply. JPF1, Jass/Jassda [BM02], and JML [LC03] are some featured approaches for Java runtime verification. It should be noted that the runtime verification approaches only prove that some certain assertions are preserved on the data operations during an execution of a concurrent Java program. This cannot guarantee that the program is free of concurrency issues.

- **Model Checking Programs** The JPF2 tool [VHBP99] is a very different tool from JPF1. It generates a state model of the Java program of reachable size using its own Java Virtual Machine JVM^{JPF}. Model checking concrete programs used to be impossible due to state explosion problem. JPF2 utilizes deduction techniques to reduce the size of the state space. The Java language features and the size of the Java applications that JPF2 can handle are of course limited.

The analysis and verification on existing concurrent Java programs can help the developers to find out the quality of their concurrent products, while a formal concurrency model may provide more support throughout the development process. As a typical formal methods strategy starts from specifying a system with an abstract specification and gradually makes it more concrete through refinement, applying formal techniques to build concurrent Java systems from formal models should also be a feasible solution. There are also different techniques established to build formal concurrency models for Java. The difference between these techniques is mainly on the different status of the concurrent models in developing.

- The traditional strategy starts from specifying the system with formal specification languages. Then the abstract model of the system is refined by refinement techniques to finally reach the concrete Java programs. These approaches include CSP-OZ to Java [CS02], FSP to Java [MK99], and Circus to Java [OC04].

- [MS02a] and [HL06] provide alternative approaches for using formal concurrency model to develop concurrent Java programs. For these kind of approaches, the concurrency model and primitives are all dropped. The developers build the system in Java without considering concurrency, and the concurrency of the system is expressed separately in formulae which control the execution of Java application. Automatic pre-processor tools are designed to translate the Java programs with formulae into normal concurrent Java applications.

Some formal approaches mentioned above are discussed in the following sections.

3.2.1 Runtime Verification: JML, Jass and Jassda

Run-time verification is a lightweight formal method approach. It is motivated by the *Design by Contract* technique [Mey92]. It employs assertion languages to specify system behaviour. Assertions can be checked automatically when the programs are executed in the run-time verification tools. These approaches include JML(Java Modelling Language) [LPC+05] and $Jass$(Java with assertions).

For each module of Java source code, a number of assertions are specified to define the allowed state of variables in the module. These assertions are usually written as a special format of comment in the source code, and can be understood by the run-time verification tool. Failure to satisfy these assertions can be detected by the verification tool at run-time. This kind of approach is good for specifying how to use the modules of a Java program, but lacks the abilities to specify and prove the correctness of the whole system. This disadvantage makes them difficult to specify and verify temporal properties of the whole system. For example, JML only supports sequential behaviour of Java code, and concurrent properties cannot be expressed using the assertion language. Although in [RDF+05], an extension for supporting multi-threaded programs verification in JML is proposed, it has not been implemented in major JML tools.

Like the other efforts which support *Design by Contract* in Java, $Jass$ supports the verification of pre-conditions, post-conditions and invariants. Furthermore, it supports a kind of *refinement checks* and *trace assertions*. The *refinement checks* in $Jass$ check if the trace of a subclass is in the trace of its superclass. For *trace assertions*, a Java object can be checked to discover if the sequence of its behaviors is in its $Jass$ trace assertions. The $Jass$ tool can translate the $Jass$ assertions, which are written as comments in Java source code file, into a Java byte-code program.

[Möl02, BM02] introduce the new $Jass$ Debug Architecture ($Jassda$). In $Jassda$, the assertions are written in CSP-like processes and checked at runtime via the Java Debug Interface(JDI), while the $Jass$ assertions are written in Java comments and need to be translated into Java source code before verifying. This technique gives a more flexible way to verify the trace properties of a Java class.

3.2.2 Model Checking Java Programs: Java Path Finder

The input languages used by model checking tools are usually simple and abstract to allow the state spaces of models to be restricted in scale. Programming languages, with rich data types, will in general cause the state space explosion in the model checker. Thus, it was generally believed in the past that model checking on concrete programming languages was very difficult to carry out due to the computational ability of existing computer systems.

[HS00] presents an automatic tool *Java Path Finder* (JPF2), which integrates model checking, program analysis and testing for Java programs. The JPF2 tool can generate a state model from a subset of the Java language via the support of its own Java Virtual Machine(JVM^{JPF}). Some reduction techniques, such as symmetry reduction, and abstract interpretation are applied to reduce the size of the state model in JPF2. Formal properties and assertions can be verified in the state model. Concurrency properties, such as data race and deadlock can be detected by JVM^{JPF}.

The Java programs checkable in JPF2 are in the 1000 to 5000 line range, and the Java language used in it is limited to a subset. The new 4.0 version of JPF claimed being able to check programs up to 10kloc, depending on their internal structure. Applying JPF model checking on a large scale concurrent system currently is still unpractical.

3.2.3 Semi-Formal Approach: The Magee approach

[MK99] presents a semi-formal strategy for build concurrent Java programs. A process algebra language, FSP (Finite State Processes) is used to specify the system. After that, the $LTSA$ (*Labelled Transition System Analyser*) tool is employed to translate the FSP specifications to an equivalent graphical description. The tool can check desirable and undesirable properties of the FSP model. To construct the Java application, the graphical version of the FSP specification is used as a guide for manual development.

This approach provide no formal translation from the FSP syntax to Java. The users must implement the model in Java through their own experience and skill. Although the concurrent model can be verified, there is no formal proof that the Java application is a correct implementation of the formal model. Therefore, the correctness of formal model cannot guarantee the correctness of the target concurrent Java programs.

3.2.4 Jeeg

Jeeg [MS02a] is a Java dialect which uses declarative Linear Temporal Logic (LTL) to replace the default synchronization mechanisms of Java. It tries to use aspect-oriented programming to fix the concurrency anomaly. Jeeg provides its own concurrency primitives to specify synchronization outside the methods of a Java class. This separates the methods, which express the actual job of the class, from its synchronization logic. This effectively limit the occurrence of the inheritance anomaly that commonly affects concurrent object-oriented languages. Also, synchronization constraints expressed in LTL make it possible to formally reason about concurrency properties.

The formulae which express the synchronization are placed in a *sync* section which is added to the class definition. The following code shows the basic structure of a Jeeg class.

```
public class MyClass{
              sync {
                 ....
              }
              // Standard Java class definition
              ....
            }
```

The *sync* section includes the LTL formulae in a form of:

m:ϕ

where m is an identifier of Java method, and ϕ is a formula expressed in a constraint language based on linear temporal logic. A pre-processor tool can automatically generate Java source files from Jeeg source files. The formula ϕ can be evaluated at run-time. When the Java programs runs in JVM, the execution of a guarded method m depends on the value of the LTL formula ϕ which gives the execution condition of the method.

3.3 JCSP

JCSP [WBM+07, WM00a] is a Java implementation of the *occam*/*occam*-π language. The *occam* language [Lim95] is an implementation language of CSP. It expresses a subset of CSP semantics. In *occam*, processes communicate with each other using communication channels. The *occam*-π language [WB04] extends the original *occam* language with π-calculus [Mil99]. It also support output guards and multi-way synchronization, which are not in the original *occam*.

JCSP inherits the same message-passing concurrency structure from CSP and *occam*. It provides various Java interfaces and classes for implementing *occam*/*occam*-π processes and channels, as well as basic processes, e.g. SKIP and STOP, parallel and sequential compositions, and external choice. Using JCSP, developers can easily construct a concurrent Java program from its CSP or *occam* specification.

The *JCSP* package implements the synchronization between the communicating processes inside channel classes. A Java application developed with *JCSP* consists of a number of objects from process classes. All the JCSP process classes implement a JCSP interface named *CSProcess*. Process objects communicate with each other through instances of JCSP channel classes.

The classical interaction between JCSP processes is a point-to-point communication channel. Figure 3.4 demonstrates a point-to-point communication.

Process P and Q synchronize on channel c, and communicate a data item X through the channel. The CSP specification of this communication is:

$$MAIN = P \underset{\{c\}}{\parallel} Q$$
$$P = c!X \rightarrow SKIP$$
$$Q = c?Y \rightarrow SKIP$$

and the *occam* language program of this is:

PAR
  c!X
  c?Y

In the JCSP implementation, for a communication channel c, there are two basic abstract $JCSP$ channel interfaces: *ChannelInput* and *ChannelOutput*. The *ChannelInput* interface defines a *read* method to read an object from the channel, while the *ChannelOutput* interface defines a *write* method to write an object to the channel. Figure 3.5 shows the process class which implements process P.

It gets the output end *out* of a channel in its constructor, and in the *run* method it sends out the data X through the output end. The communication channel is defined

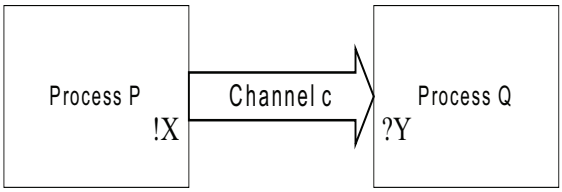

FIGURE 3.4: Channel and process of JCSP

```
class P implements CSProcess{
   private final ChannelOutput out;
   ......
   public P(ChannelOutput out){
      this.out = out;
   }
   public void run(){
      ......
      out.write(X);
   }
}
```

FIGURE 3.5: The JCSP process class implements process P

```
Class Main implements CSProcess{
   final private One2OneChannel c = Channel.one2one();
   ......
   public void run(){
      new Parallel(
         new CSProcess[]{
            new P(c.out()),
            new Q(c.in())
         }
      ).run();
   }
}
```

FIGURE 3.6: JCSP Parallel Processes

outside of the process. In Figure 3.6, the *JCSP* process objects P and Q are grouped in an array, and executed in parallel under the *Parallel* process.

In this case, the communication channel is declared as an one-to-one communication channel. In a parallel composition structure of the *Main* process, the output end of channel c is passed to process P and the input end is passed to process Q.

The communication only involves synchronization between one reader and one writer, which also means that there must be at least one reader and one writer. The reader and the writer processes synchronize with each other, and the writer process sends data to the reader. JCSP/occam also supports multiple writers and/or readers interleaving with each other to use a shared *any-to-any* channel. Note that the writers (respectively readers) do not synchronise with each other – only one reader with one writer. Therefore, the one-to-one, one-to-any, any-to-one, and any-to-any channel classes in JCSP are still point-to-point communication channels.

JCSP has implemented the barrier synchronization, which is a stateless multi-way synchronization, with the *Barrier* (and *AltingBarrier*) class. A barrier has an internal counter for all the synchronizing processes. When a process call the barrier, the process blocks and the barrier reduces the counter. Only when the counter is reduced to 0, which means all the synchronizing processes are ready, the blocked processes are released and can progress. The *Barrier* class in JCSP implements a multi-way synchronizing barrier. However, until version 1.0rc5, the *Barrier* class is not a guard, and cannot be used in external choice. We discuss this issue in Section 3.3.1.

To build a concurrent Java application, the developer can specify the system using *CSP*, verify the model using FDR [For03], and then develop a JCSP program from the CSP model. The benefit of JCSP is that the developer is protected from low-level synchronization issues during implementation. As the compositional semantics of CSP is carried over by JCSP, more often, people just develop the Java program directly with the JCSP library.

The producer-consumer solution given in Figure 3.2 and Figure 3.3 work for any number of producers and consumers. A JCSP version, almost identical that from Figure 3.3, is given in Figure 3.7.

```
Any2AnyChannel c = Channel.any2any(new Buffer(size));
...
//Consumer thread
while(true) { consume(c.read()); }
...
//Producer thread
while(true) { c.write(produce()); }
```

FIGURE 3.7: Consumer-Producer Example: The *JCSP* Solution

The correctness of the JCSP implementation of the *occam* communication channel to a JCSP channel has been formally proved [WM00a]: the CSP model of the JCSP channel communication is shown to be failures-divergences equivalent to the CSP channel.

3.3.1 The Limitation of JCSP 1.0rc5 and before

The JCSP packages of 1.0rc5 version and before mainly focus on implementing the point-to-point communication as their concurrency model. At a time, only one reader process and one writer process can synchronize and communicate data through the communication channel.

Although these packages have a multi-way synchronizing *Barrier* class implemented, the *Barrier* class is more like a separate synchronization facility as it cannot be used for guarded external choice. In JCSP, the *Alternative* class implement the alternative processes structure (`ALT`) of *occam-π*. In the 1.0rc5 version, the implementation of external choice has the same constraints as required by classical occam. Even for point-to-point communication channels, only the channel input ($ch?x$) can be used for guarded external choice. The output-end process must commit to a channel communication, which means it cannot use a channel output ($ch!x$) in external choice. Solving guarded external choice for both ends of the communication channel and multi-way synchronizing channel is complex and can be costly [McE06].

3.3.2 New JCSP versions

Recently, as new versions of JCSP have moved on to support the *occam-pi* language. In [WBM+07], new features in the new 1.0rc7 and 1.1 versions of JCSP are introduced.

A new and stable *AltingBarrier* class has been implemented in 1.0-rc7. The *AltingBarrier* class implements multi-way synchronization with guards. Therefore, it can be used

in external choice. In Section 5.4, we have a more detailed discussion on the implementation of multi-way synchronization.

In JCSP programs before version 1.1, Processes take channel-end types, such as *ChannelOutput* or *ChannelInput*, as arguments to process constructors. Instances of channels are passed directly to these processes. This allowed users to cast a *ChannlInput* as a *ChannelOutput*, this is now prevented by passing instances of channel ends. The *input-* and *output-end* can be extracted from a channel instance using *in* and *out* methods of the channel.

The 1.1 version also introduces extended rendezvous. A rendezvous allows the input-end process to execute extra code without scheduling the output-end process. The input-end process starts a rendezvous by calling the *startRead* method of a *ChannelInput* instance, and ends the rendezvous by calling the *endRead* method. During this period, the output-end process remains blocked.

Other useful features of the 1.1 version includes *poison* and *graceful termination*. These new features in 1.0rc7 and 1.1 versions provide better support of *occam-pi*, and improve the scalability of JCSP on various concurrency systems.

3.3.3 A Translation Tool for JCSP

Since the $JCSP$ package implements a subset of CSP syntax, it is possible to automatically translate a formal concurrency model specified with the subset of CSP syntax to a Java application. In [RRS03], a translation tool, which can automatically generate JCSP-based Java programs from CSP processes, is presented.

To develop a Java concurrent application with the tool, a concurrency model is specified with a subset of CSP notations. Then the specified model is checked in FDR to prove its correctness. After that, the concurrency model is translated to a Java program.

The formal model supported by this tool is a pure restricted CSP model. Therefore, the users have to add the extra data definition and operation, which cannot be supported by the tool, in the target Java program manually. However, the manually introduced data may affect behaviours of the system. Therefore, even if the translation is correct, the manual changes may make the final code inconsistent with its CSP model.

The CSP notations supported by Raju's tool are very limited. Furthermore, the tool has been found experimentally not to be robust enough to handle non-trivial examples.

3.3.4 From Circus to JCSP

In [OC04], the translation rules for translating *Circus* program to Java programs via the JCSP package are presented. In [FC06], a translation tool is introduced. The *Circus*

programs used for translation are refined to be more concrete. They are written in the executable subset of *Circus*.

As *Circus* is a combined specification from Z and a subset of CSP, the CSP part, which controls the execution flow of the specification, can be translated into JCSP as normal CSP specification. Each *Circus* process declared in the `ProcDecl` section is translated into a JCSP process class *CSProcess*. The main `Action` section of a *Circus* process, which determines how the process performs, generates the Java statements inside the *run()* method of the *CSProcess* class. Each CSP-like action of *Circus* corresponds to a JCSP channel.

In Chapter 8, we discuss and compare our approach and the *Circus* translation.

Chapter 4

The Combined B+CSP Specification

In ProB, the B part of the combined specification is essentially an action system [BKS83a]. It specifies the abstract state of the system based on a number of constants and variables. The system state is shared by a number of guarded B operations in the system model. The operations can change the state of the system by updating the values of system variables. In abstract models, the enablement of an operation is guarded by predicates on the state of the system. The CSP part, on the other hand, defines the behaviours of the system by specifying the possible operation sequences/traces. In CSP, the actions in the system are regarded as stateless channels. A process, a key concept, is defined in terms of possible behaviour sequences of those channels. Each process may also maintain variables which are only locally visible to the process itself.

Semantically, the B machine can be regarded as a special process, which is running in parallel with the CSP processes. The B operations are also in a parallel composition and synchronize with the corresponding CSP channels. The B and the CSP combine with each other through shared operational event names. A combined B+CSP event comprises a B operation and a CSP channel sharing name. It is only allowed to proceed when it is allowed both by the B and the CSP. The B specification can guard a combined event with precondition predicate. The combined event also needs all the CSP processes who synchronize on it to be ready.

ProB interprets the B and the CSP specifications into *Prolog*, and builds a state model from the Prolog representation. It also supports animation and model checking of the combined specification. The operational semantics of the B+CSP specification is introduced in [BL05] and provides a formal basis for combining the B and CSP specification.

In this chapter, we introduce the B+CSP specification, and how we make use of it in this work. This chapter starts with an introduction of the notations of the combined B+CSP

specification language in Section 4.1. In Section 4.2, the operational semantics of B+CSP and the combination strategy are discussed. As the semantics is developed for supporting abstract specification, it is too flexible for the implementation in programming languages. Therefore, in Section 4.3, the reasons for restricting the original semantics are discussed, and a restricted semantics of B+CSP combination is presented. In Section 4.4, we describe how to compute the restricted semantics in the implementation. Based on the new restricted semantics, the synchronization model is explained in Section 4.5. At the time of writing, the B+CSP approach is not methodologically complete and has its own limitations in the refinement and composition rules.

4.1 The Specification language

Table 4.1 gives a partial B and CSP syntax used in the thesis. We use quote marks as well as boldface to denote BNF terminal strings. A statement $S^{+\text{“;”}}$ means an symbol S can appears one or more times, and elements of S are separated by the terminal “;”. A statement S^* means S can appears zero or multiple times.

In the work, the target B specification is mainly the B0 subset. As introduced in Section 2.1.3, the B0 subset only supports concrete data and concrete data transitions. That makes it very close to programming languages and is easier to implement than many other abstract B features. We do support some abstract B language features beyond B0. One notable feature is the precondition substitution. Possibly subject to a precondition **PRE** - all of whose clauses must be satisfied to enable the operation - an operation updates system state using various forms of data substitution. These features are implemented to provide extra functions for rapidly implementing and testing an abstract specification in Java programs. In the implementation, preconditions are interpreted as guards, which will block the process if the precondition is not satisfied. The other one is parallel composition, which is normally resolved through nondeterminism in the abstract model. With the decomposition approach explained in Section 7.3, parallel composed B substitutions in a B operation can be decomposed into separate B operations of different parallel B machines.

4.2 The Operational Semantics of B+CSP

The operational semantics of B+CSP is introduced in [BL05]. It provides a formal basis for combining the B and CSP specification. The B and CSP specification are composed as parallel processes. A B machine is viewed as a special process in the system, which maintains and updates the system state through the data transitions in its operations.

B Machine	*Machine*	**MACHINE** *Header* *Clause_machine** **END**
	Clause_machine	... | *Clause_variables* | *Clause_invariant* | *Clause_assertions* | *Clause_initialization* | *Clause_operations* | ...
B Operation	*Clause_operations*	**OPERATIONS** *Operation*$^{+\text{“;”}}$
	Operation	*Header_operation* “=” *Substitution*
	Header_operation	[*ID*$^{+\text{“,”}}$ ←] *ID* [“(” *ID*$^{+\text{“,”}}$ “)”]
B Substitution	*Precondition*	**PRE** *Condition* **THEN** *Substitution* **END**
	Block	**BEGIN** *Substitution* **END**
	If-Then-Else	**IF** *Condition* **THEN** *Substitution* [**ELSIF** *Condition* **THEN** *Substitution*]* [**ELSE** *Substitution*] **END**
	Var	**VAR** *ID*$^{+\text{“,”}}$ **IN** *Substitution* **END**
	Sequence	*Substitution* “**;**” *Substitution*
	Parallel	*Substitution* ‖ *Substitution*
	Assignment	*ID*[(*Expression*)] “**:=**” *Expression*
CSP Process	*Prefix*	*ChannelExp* → *Process*
	Sequential Composition	*Process* “**;**” *Process*
	External Choice	*Process* “[]” *Process*
	Alphabetical Parallel	*Process* “[|” Ch_List“|]” *Process*
	Interleaving	*Process* “|||” *Process*
	Process call	*Proc_Header*
	If-Then-Else	**if** *CSP_Condition* **then** *Process* [**else** *Process*]
	Skip	**SKIP**
	Stop	**STOP**
CSP Channels	*ChannelExp*	*ID* [*Output_Parameter**] [*Input_Parameter**]
	Output_Parameter	“!” *CSPExp* | “.” *CSPExp*
	Input_Parameter	“?” *CSPExp*

TABLE 4.1: The main B and CSP specification syntax supported in JCSProB

Without CSP processes, a B machine process can fire all its operations freely. The data transitions can only be blocked by preconditions on the operation. However, in some cases, it may not be very convenient to define system level behaviours only with precondition guards. Normally, a B machine needs to define an abstract 'program counter' and use it in preconditions to control the execution of an operation [AM98]. However, this form of specification of behaviour is opaque, compared to process algebra approaches such as CSP.

To work with CSP processes, a B machine process needs to synchronize and communicate. The synchronization and integration of B and CSP processes are on the B operations and CSP channels.

- A B operation must have a corresponding CSP channel with the identical name. Together, they build up a combined B+CSP channel. The B operation is only ready to progress when the corresponding CSP channel is also ready.

- A CSP channel is combined with a B operation, or it can be a pure CSP channel which has no B counterpart. A pure CSP channel is only used in the CSP part for communication.

Figure 4.1 illustrates how the synchronization works. Operations A, B, and C all have corresponding CSP channels in the CSP part. Only when channel A is ready, operation A is able to progress the data transitions inside the operation. CSP channel D is only used by CSP processes, and has no counterpart in the B machine. In this way, the system behaviour specified in CSP can be used to control the execution of data transitions in the B machine.

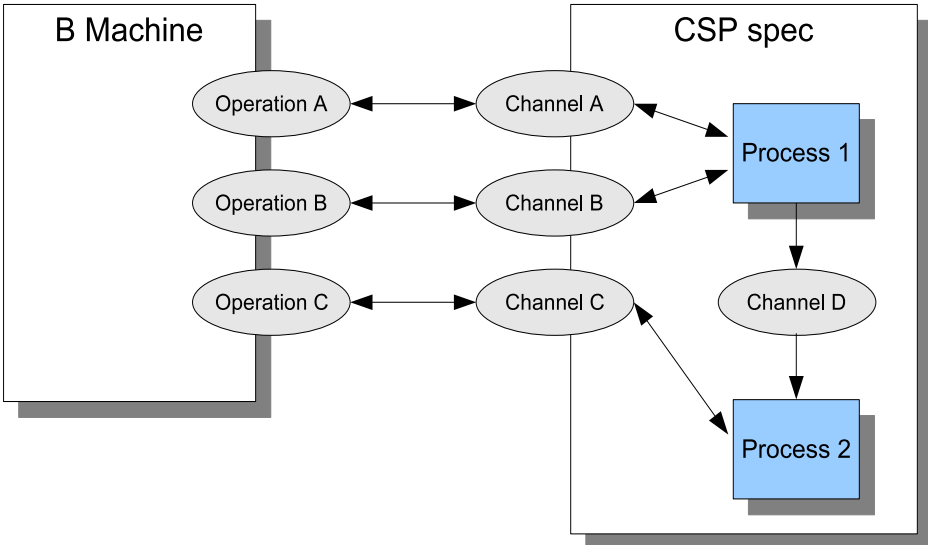

FIGURE 4.1: The synchronization between B and CSP specification

The combined B+CSP event is defined by the operational semantics. A state of a combined B+CSP specification is defined as a pair, which includes a B state and a CSP state.

In [BL05], states σ and σ' are the before and after B states for executing a B operation. The operation is defined with operation identifier op, return variable r_1, ... , r_m, and input variables a_1, ... , a_n, as r_1, ... , $r_m \leftarrow op(a_1, \ldots, a_n)$. The B operational semantics can thus be defined with a ternary relation $\rightarrow$ as $\sigma \rightarrow_{op} \sigma'$. That means in state σ, the operation op progresses with input variable a_1, ... , a_n, then returns output variables r_1, ... , r_m, and reaches a new state σ'. In the CSP part, P is a CSP process, and P' is the process after P processing CSP channel ch, which has the same identical name as the B operation op. Channel ch can be defined with a number of variables b_1, ... ,b_i as

$ch.b_1$. ... $.b_i$. The CSP operational semantics is give by a similar relation $\rightarrow$ as $P \rightarrow_{ch} P'$.

Therefore, the before and after state of B+CSP specification can be defined as (σ,P) and (σ',P'). We can now define the operational semantics of B+CSP specification by combining the two ternary relations into one form (σ,P) $\rightarrow_{ev}$ (σ',P'), where the combined event ev is an synchronization of B operation op and CSP channel ch.

The essential issue of this synchronization is how to define the data flows of B operations and CSP channels. A B operation can have input variables a_1, ... ,a_n and return output variables r_1, ... , r_m as result, while variables b_1, ... ,b_i of a CSP channel can have input(?), output(!) and dot(.) decorations to imply the data flow of the variables.

ProB supports a very flexible way to combine the data flows of B operations and CSP channels. In ProB, the synchronization is achieved by Prolog unification, which means data information can flow in from both B and CSP:

- CSP channels can provide concrete data values , which means the CSP part is used to drive the B part.
- B operations can provide data values, which means the B part is used to drive the CSP part.
- B and CSP can both provide concrete data values to each other. The mixed data flow allows B and CSP can drive each other at the same time.
- In the worst case, when both B and CSP do not provide concrete data values, ProB can enumerate the B datatypes of variables and drive the interpreter.

As an abstract model checking tool, ProB tries to explore all the possible states of a system. The power of enumerating data values from datatype definitions makes it capable of using the combined channels without caring about the input/output data flow on the channels. Therefore, it does not clearly distinguish the input and output variables of both B and CSP. For example, in ProB, *ch!aa* and *ch?aa* are all valid CSP channels for combining with B operation *op(aa)*. Moreover, it even does not force the numbers of variables on a B operation and its corresponding CSP channel to be same. For example, a CSP channel *ch.aa.bb* can have fewer variables than its corresponding B operation *op(aa,bb,cc)*. It ignores the missing variable *cc*, and either B part or the ProB interpreter can provide the concrete data value for it.

For a very simple B machine in Figure 4.2, we combine it with the following CSP process:

$$Proc = Set?Val \rightarrow Get!Val \rightarrow Proc$$

MACHINE Simple
SETS AA = aa,bb,cc
VARIABLES xx
INVARIANT xx ∈ AA
INITIALISATION xx := aa
OPERATIONS
Set(newval) =
PRE newval ∈ AA **THEN**
xx := newval **END**;
res ← Get = **BEGIN** res := xx **END**
END

FIGURE 4.2: A simple B machine: Simple

As both B and CSP require the data value of *Val* on combined event *Set*, neither of them can provide concrete data to drive the combined machine. In this case, PROB interpreter would provide the concrete data using its enumeration mechanism, and actually drive the combined model. The enumeration mechanism provides data values based on the data type of a variable. It also looks into the enumeration configuration in PROB for the size of the enumeration values it can provide. For example, if *Val* is a natural number, and the enumeration size setting in PROB is 3, the enumeration mechanism would provide three natural numbers 1, 2, and 3 as the result. The *Get* event here is more complex, as both the B operation and the CSP channel try to output data. The PROB interpreter would only combine them together when the output data values from B and CSP are equal.

We can also combine the B machine showed in Figure 4.2 with a different CSP process:

$$Proc = Get?Val \rightarrow Set!Val \rightarrow Proc$$

In the new combined model, the combined event *Get* has a B operation which outputs data, and a CSP channel which requires an input data. So, the B part is driving the combined event here by providing the data. On the other hand, the *Set* event is a combination with a CSP channel which outputs the value of *Val*, and a B operation which needs an input parameter. The data outputted from the CSP part would drive the combined event here.

The above two examples demonstrate how flexible the combination could be in the PROB tool. Actually, even when part of the parameter definitions are missing, PROB can still successfully perform the combination. For example, if the same B machine in Figure 4.2 is combined with a CSP:

$$Proc = Get \rightarrow Set!Val \rightarrow Proc$$

although the corresponding parameter on CSP channel *Get* is missing, ProB can still combine the B operation with it. In this case, there would not be any data communication between the CSP and the B parts in the combination. The two parts just synchronize with each other on the occurrence of the event.

4.3 The Restricted Semantics for Implementation

As a model checking tool, ProB aims to exhaustively explore all the states of an abstract finite state system, on the way enumerating all possible value combinations of operation arguments. The flexibility in combining the two formal models provides more power to the ProB tool to model check the state space of a model. The implementation of the semantics using Prolog unification is simple and efficient. Especially, in some cases, it allows the ProB interpreter, instead the B or the CSP, to drive the combined model. However, as our target is implementing the combined B+CSP specification in a concrete programming language, we cannot implement the involvement of the ProB interpreter or support the same flexible and abstract semantics as model checkers. Therefore, we have to restrict the original semantics in ProB to make it suitable and meaningful for a concrete programming language.

The ProB interpreter is used when neither the B nor the CSP provides full data information to drive the model. There are three kinds of cases, where this can happen:

- Both the B operation *op(aa)* and the CSP channel *ch?aa* request the data value of variable *aa*.
- The B operation *op(aa)* requests the data value of variable *aa*, while CSP channel *ch* does not provide the value.
- The CSP channel *ch?aa* requests the data value of variable *aa*, while B operation *op* does not provide the value.

In our restricted semantics, we prohibit all the three combinations. Furthermore, there is another combination of the B and CSP variables dropped from the semantics.

- Both the B operation $aa \leftarrow op$ and the CSP channel *ch!aa* output data values.

ProB can handle this with its Prolog unification. Only when the two output values from the B and the CSP are same, can the B operation and the CSP channel be combined. However, this violates the concurrency model of our approach. Section 4.5 will give a discussion of this in detail. The allowed argument combinations in this work are showed in Table 4.2.

JCSProB	B: input arguments ($c(x)$)	B: return arguments ($y \leftarrow c$)	B: no argument (c)
CSP output ($c!x$, $c.x$)	$\surd$	$\times$	$\times$
CSP input ($c?y$)	$\times$	$\surd$	$\times$
CSP none (c)	$\times$	$\times$	$\surd$

TABLE 4.2: The allowed arguments combination for B+CSP events

We thus define a restricted B+CSP operational semantics as follows. For a B operation o_1,...,$o_m \leftarrow$ op(i_1,...,i_n), its corresponding CSP channel must be in the form of $ch!i_1...!i_n?o_1...?o_m$. At CSP state P, a CSP process sends channel arguments i_1,...,i_n through the channel to a B operation as input arguments. After the data transitions of the channel complete - taking B state from σ to σ' - the CSP state changes to P'. The arguments o_1,...,o_m represent the data returned from B to CSP. The input arguments i_1,...,i_n only exist in state (σ,P), while the output arguments o_1,...,o_m are only available in state (σ',P'). The new restricted semantics of a combined event ev can be expressed as ((σ,P),in) $\rightarrow_{ev}$ ((σ',P'),out), where $in = i_1$,...,i_n, and $out = o_1$,...,o_m.

PROB also supports classical CSP communication channels. These channels exist only in the CSP part of the combined specification and have no B counterparts, which means that they cannot directly affect the system states in the B part. A channel output ($c!y$) synchronizes with one channel input ($c?x$) from a different process, and transfer a data. This synchronization is a point-to-point(p2p) communication pattern. It also supports multi-way synchronization, multiple processes can synchronize on one barrier channel c. Table 4.3 demonstrates CSP communication channels supported in this work.

JCSP	CSP input ($c?y$)	CSP output ($c!x$)	CSP none (c)
CSP output ($c!x$)	$\surd$ (p2p sync)	$\times$	$\times$
CSP input ($c?y$)	$\times$	$\surd$ (p2p sync)	$\times$
CSP none (c)	$\times$	$\times$	$\surd$ (Barrier)

TABLE 4.3: The allowed arguments combination for pure CSP event

4.4 How to Compute the Restricted Semantics

The restricted semantics described in Section 4.3 provides the formal basis of combining the B and the CSP models. The restricted semantics supports a two-way communication between a B operation and a CSP event, which means that the CSP can provide the data information to drive the B machine and/or vice-versa. Figure 4.3 shows how the semantics actually perform.

In step (1), the system is in state (σ,P). The CSP event ev can provide variables i_1,...,i_n to the B operation op as input arguments for the B operation. From the view of the CSP event, it outputs (!) this data to the B part, while the B operation sees this as a data input from the CSP event. Thus the CSP part uses the input arguments to drive the B machine and its actions on the system state. Even if there is no input data from the CSP, the CSP event still drives the B machine by invoking the execution of the B operation.

In step (2), the B operation op uses the input variables i_1,...,i_n (or without any input data), and processes some data transitions to move the state of the system from (σ,P) to (σ',P'). At the same time, it can produce some output data o_1,...,o_m.

In step (3), the B operation op send the output variables o_1,...,o_m back to the CSP event ev. The CSP event receives the variables as input (?). These variables can be further used by the CSP process to control the behaviour of the system. In this way, B machine can use the output variables o_1,...,o_m to drive the CSP part.

Accordingly, this semantics allows the system behaviour to drive and change the data aspect of the system, while the data aspect can also use the return variables to affect the behaviour of the system. As an example of the restricted semantics, Figure 4.4 shows a combined specification of a powered lift. The B machine has two variables to specify the system state: *level* indicates the floor of the lift, and *electr* marks the electric power left in the battery. The lift goes up and down through *inc* and *dec* operations. Each of these operation costs energy from the battery. When the power in the battery is below 40, the lift enters emergency mode. It uses the CSP channel *alarm* to call the supplier, and retraces to level 0 for resetting and charging. After the supplier is notified by the alarm, he recharges the battery. After the battery is recharged, the lift can return to normal functioning.

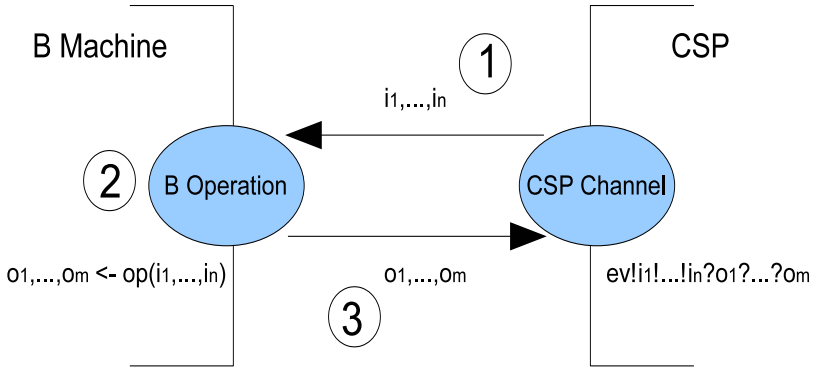

FIGURE 4.3: How to compute the restricted semantics

MACHINE *powered_lift*
VARIABLES *level, electr*
INVARIANT $level \in \mathbb{N} \wedge level \geq 0 \wedge level \leq 10 \wedge$
$electr \in N \wedge electr \geq 0$
INITIALISATION $level := 1 \parallel electr := 100$
OPERATIONS
$rr \leftarrow inc =$
PRE $level < 10 \wedge 2 < electr$
THEN $level := level + 1 \parallel electr := electr - 2 \parallel rr := electr - 2$
END;
$rr \leftarrow dec =$
PRE $level > 0 \wedge 2 < electr$
THEN $level := level - 1 \parallel electr := electr - 2 \parallel rr := electr - 2$
END;
$recharge =$
PRE $level == 0$ ***THEN*** $electr := electr + 150$ ***END***;
$reset =$
BEGIN $electr := electr - level \times 2 \parallel level := 0$ ***END***;
$rr \leftarrow test =$
BEGIN $rr := electr$ ***END***
END

———————————————————

$MAIN = Lift \ [|\{alarm\}|] \ Supplier$
$Lift = inc?Y \rightarrow LiftTest(Y) \ [] \ dec?Y \rightarrow LiftTest(Y)$
$LiftTest(X) =$ ***if*** $X<40$ ***then*** $alarm \rightarrow reset \rightarrow Emergency$ ***else*** $Lift$
$Emergency = test?Y \rightarrow$ ***if*** $Y \geq 40$ ***then*** $Lift$ ***else*** $Emergency$
$Supplier = alarm \rightarrow recharge \rightarrow Supplier$

FIGURE 4.4: Combined Specification of powered lift

In this model, the CSP part calls *inc*, *dec* and *recharge* channels to invoke the corresponding B operations for changing the system state. On the other hand, the return variables in channels *inc*, *dec* and *test* are further used in CSP processes as condition variables in the ***if-else-then*** structures. The behaviours of these CSP processes depend on the value of these variables.

4.5 The Concurrency Model

The behaviour of the system is specified in the CSP part of the combined specification. Processes in the CSP part can be in several kinds of processes compositions, e.g. interleave, parallel, sequential and choice. In particular, parallel CSP processes can synchronize with each other on certain CSP channels. In the restricted B+CSP semantics, B+CSP channels and pure CSP events have different concurrency models for synchronization.

A combined B+CSP channel has two levels of synchronization.

- The CSP level. In this level, All the CSP processes which call this channel need to synchronize on the channel. When a process calls a synchronizing channel, it blocks until all the other processes which synchronize on this channel are ready. The synchronization here is determined by the name of the channel, as well as the values of the variables on it.
- The combination level. The synchronization is between the CSP part (including all the CSP processes) and the B part. The call from the CSP part depends on the CSP level of synchronization, while the corresponding B operation can be guarded by preconditions. Only when both the B and the CSP are ready, the data transitions inside the channel can progress.

Figure 4.5 shows how four processes *p1*, *p2*, *p3* and *p4* synchronize on a combined event *ch*.

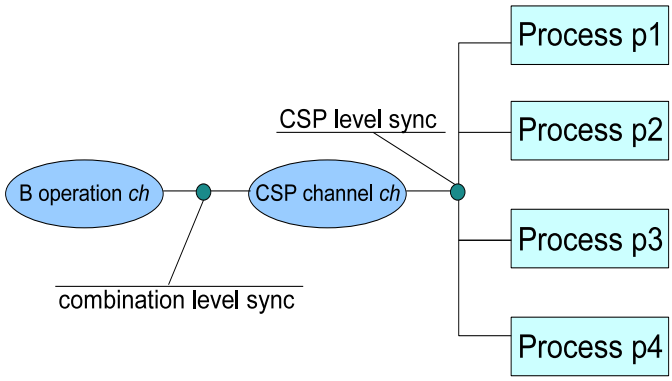

FIGURE 4.5: The synchronization of B+CSP channels

In the CSP level, the value of the variables on the channel is also an important factor in the synchronization, as well as the channel name. Only when parallel processes call the same channel with exactly the same values, do they synchronize with each other on that channel. For example, if process P and Q synchronize on channel $ch!x$, only when variable x_1 of the call $ch!x_1$ from process P equals variable x_2 of the call $ch!x_2$ from process Q, can the two processes synchronize.

Furthermore, the synchronization is only concerned with input data i_1,...,i_n. An input value must have a defined value at the time of synchronization because the value needs to be passed to the B operation. A B operation reads a defined value for its input data in order to trigger.

In ProB, the output variables o_1,...,o_m can also be used for synchronization, while in the restricted semantics, this is not allowed. The synchronization between the CSP

processes and B machine determines whether a B+CSP channel can progress. Therefore, the decision should be made in the state (σ,P), before the data transitions are actually processed. The output data o_1,...,o_m are only available in state (σ',P'), after the data transition progresses. However, when the values of input data of the combined event and the system state are fixed, a B operation would always move the system to a specific state, and produce a certain output data.

The synchronization between CSP processes on a channel can be viewed as a barrier synchronization with data. To simplify the concurrency model, it is suffice to say that processes calling the same channel with exactly the same data values perform a single barrier synchronization, which means it is possible to use a classical barrier synchronization to implement the CSP level of synchronization for B+CSP channel. The synchronization between a CSP channel and its corresponding B operation can be regarded as a two-way point-to-point communication: the CSP channel first sends the input data i_1,...,i_n to the B operation, and after the data transition is performed, the B operation sends output data o_1,...,o_m back to the CSP event. This gives the possibility of implementing the B+CSP synchronization using JCSP/*occam-π*. We will discuss this later in Section 8.2.5.

In ProB, pure CSP channels preserve the semantics of classical CSP. In this work, we use JCSP communication channels to implement pure CSP channels, which means the CSP channels here need to express the same semantics as JCSP/*occam-pi* does. The standard channel model of JCSP/*occam* provides point-to-point communication channel classes, which is introduced in Section 3.3. These channel classes are employed to implement the point-to-point communication channels in Table 4.3. As JCSP has implemented the barrier synchronization with the *Barrier* and *AltingBarrier* classes, we uses these classes to implement the barrier synchronization of pure CSP channels, which is also allowed in this work.

Chapter 5

JCSProB: A Java Implementation of B+CSP

Implementation is the final target for the development of all software systems. A typical formal development employs a formalized stepwise refinement techniques. The development starts with a very abstract non-deterministic system model, which is developed with high-level observation of the system behaviours. As the high-level abstract model lacks concrete details of the system, it is very hard to be implemented. In a series of refinement steps, low-level details of the system are added in the system model. Eventually, we can get a final refinement of the system model, which is concrete enough to be input to the compiler to generate an executable program.

In B development, the final refinement is defined in a deterministic subset of the B language, B0. Although B0 only includes concrete data and concrete data manipulations, it still lacks some implementation details and there is no compiler support for this language. Alternatively, as B0 is very close to conventional programming languages, it is possible to translate B0 into other well supported programming languages. The *Atelier-B* [Ate01] tool provides translation tools [Cle02] from B0 to Ada and C. In this work, we translate combined B+CSP specifications into Java programs.

The definition of the B+CSP semantics in Section 4.2 shows that the behaviours of a system are specified in the CSP part of the combined model. In B+CSP, the CSP part of a system is the driving force of system behaviour, while the B part specifies a reactive system with data and data manipulations. Although the data aspect can affect the system behaviours via the return variables of combined events, the invocation of events is controlled by the CSP processes. As the semantics of the B0 language is very close to conventional programming languages, it is not very difficult to translate the B0 specification into programming language. However, it is not common for a programming language to directly provide a mechanism to implement the semantics of

a process algebra. Therefore, how to implement the system behaviours specified by the CSP part of the combined specification is a key question of the work.

This chapter starts with a discussion on the reasons for developing the JCSProB package in Section 5.1. Then it gives an overview of the package structure in Section 5.2. In Section 5.3, we introduce our implementation of the combined B+CSP event. The implementation of multi-way synchronization of this event implementation is a very important feature for JCSProB, and in Section 5.4 we give a full discussion of our solution. Another important re-implementation is of the process mechanism. New thread, process and process composition classes are introduced in Section 5.5. A big difference between JCSP and JCSProB is that JCSProB has a state model defined in B specification. The implementation of the state model is in a variable class, which is presented in Section 5.6. In Section 5.7, a GUI package, which is used to interact with JCSProB programs, is introduced.

5.1 Why We Need JCSProB

The JCSP package [WM00b] provides a Java implementation of the *oocam-π* language, which expresses a subset of CSP semantics (see the discussion in Section 3.3). Because JCSP is a well constructed package and had the channel classes formally verified [WM00a], if we can implement the B+CSP event based on the JCSP package, correctness of the implementation should be easier to prove. Therefore, our first priority is using it to implement the CSP part of the system. However, some limitations of JCSP make it not very suitable for implementing the B+CSP event.

5.1.1 JCSP Channels and B+CSP Events

Both in JCSP and B+CSP, the synchronization between processes is through the channels/events. The difference between B+CSP and JCSP concurrency models is an important question in implementing B+CSP semantics using JCSP.

In Section 3.3.1, we discussed that the limitation of JCSP 1.0rc5 on multi-way synchronization and external choice, which was the available version of the JCSP package when we started this work. However, if we want to implement the multi-way synchronizing B+CSP events, external choice is an important feature which must be implemented. Therefore, we had to implement multi-way synchronizing events by ourselves. We will have a detailed discussion of this in Section 5.4.

Another big difference is the implementation of system state and the data transitions which change the system state. In JCSP/*occam-π*, each process maintains its own states, and the channels are regarded as stateless communication channels which have no

internal data transitions. The state changes of JCSP/*occam-π* happen in the processes, not in the channels. However, the B part of combined B+CSP events has data transitions inside to change the state of the B machine. This means we cannot directly use the JCSP channel classes to implement the B+CSP event.

Table 5.1 summarizes the differences discussed above.

Channels	JCSP channel	JCSP call channel	JCSP barrier	B+CSP event
Multi-way Sync	No	No	Yes	Need
External Choice	Yes	Yes (acceptor side only)	No	Need
Data on Channel	Yes	Yes (bi-direction)	No	Need
Data Transition	No	Yes (acceptor side only)	No	Need

TABLE 5.1: JCSP (1.0rc5) channel, barrier, and B+CSP event

Accordingly, the implementation of the B+CSP event needs to support multi-way synchronization, external choice, data on the event, and atomic data transitions inside the event. As the JCSP package before 1.0rc6 does not provide enough support for implementing the B+CSP event, we have to implement it in a new Java program.

5.1.2 CSP Process Call, Recursion and *occam-pi* Loop

A pragmatic difference between B+CSP and JCSP/*occam-π* is how recursion is managed. Although both *occam-π* and JCSP (through Java) allow recursion in processes, unbounded use (e.g. to express non-terminating cyclic behaviour) would lead to memory allocation failure. Many simple recursions in B+CSP are actually *tail* recursions and these are easily expressed with conventional `WHILE` loops in *occam-π* and Java (for JCSP). More complex B+CSP recursions, if they are unbounded, have to be transformed for memory safe implementation in JCSP/*occam-π*. In Section 5.5, this is addressed with a new mechanism within JCSProB, so that complex recursions may be implemented directly without memory problems.

The loop statements in both JCSP and *occam-π* represent a tail recursion inside a process, while the CSP part of the B+CSP specification supports more flexible, natural forms of recursions. A CSP process can call itself at the end of the process to do tail recursion. For example, a process P is defined as:

$$P = a \rightarrow P$$

After process P performs an event a, it enters a state P'. In state P', it calls process P. Then the process would perform as P again. In this way, event a is repeatedly called. Although the tail recursion described here can be easily implemented by the loop structures in JCSP and *occam-π*, they are still a bit different. In JCSP/*occam-π*,

the loop is inside a process, which means the loop structure maintains the state of the process. However, in general CSP *tail* recursion, when a CSP process P calls a new process P' at the end of the process P, the state within the process P is not directly accessible from the new process P'. In CSP, if we do want to maintain the data state within the recursion, one solution here is passing data between new process instance and its ancestor. For example:

$$P(x) = a!x \rightarrow b?y \rightarrow P(y)$$

Moreover, using branching structures, such as condition and choice, CSP can produce more complex recursion patterns than can be neatly modelled by ***while*** loops. In CSP, one process also can call other named processes. When process P calls process Q, the process would perform the behaviours defined in process Q. If we add branching structures into processes, one process can have different descendant processes depending on condition and choice. For example, the *Lift* process shown in the CSP specification of Figure 4.4 can further perform as process *Lift* or process *Emergency*. The decision is made by an external choice and the conditions of the ***if-else-then*** structures. As it involves a new process *Emergency*, the recursion here is complex enough to disable us from using a loop inside the *Lift* process to resolve it. It can be programmed just using loop or tail recursion, but that requires putting the *LiftTest* and *Emergency* processes inside the *Lift* process as a nested loop. For examples with more complex branching structures, the loop expression can be very hard to build, and harder to understand.

Therefore, it would be more reasonable to fully implement recursion in Java for CSP processes, than using the ***while*** loop. In Section 5.5, we give a in-depth discussion of this problem, as well as the solution.

The discussion in Section 5.1.1 and Section 5.1.2 shows that the channel classes in the old JCSP package (before 1.0rc6) are not capable to smoothly implement the B+CSP event. To deal with these limitations, we construct a new Java package, JCSProB, to implement the B+CSP semantics and concurrency.

5.2 An Overview of the JCSProB Package

The JCSProB package implements the B+CSP semantics in Java. It includes three sub-packages:

- The *jcsprob.lang* package includes the implementation of B+CSP specification.
- The *jcsprob.gui* package includes the Java interfaces and classes for constructing the GUI program.

- The *jcsprob.msg* package includes the implementation of the communication between the JCSProB process objects and the GUI program.

Some of the JCSProB classes inherit JCSP interfaces or classes, e.g. the process class *BCSProcess* implements the *CSProcess*, and many JCSP interfaces and classes can be directly used in a Java program together with JCSProB. Apart from that, a number of new event interfaces and classes are developed to implement the semantics of the B+CSP event. Table 5.2 shows the correspondence between the B+CSP model and its Java implementation in JCSP and JCSProB.

B+CSP	JCSP/JCSProB
Combined B+CSP event: including functions of synchronization, data transitions, and data input/output	JCSProB event interfaces and classes
Point-to-point communication channels	JCSP communication channel classes
Guarded external choice with multi-way synchronization for B+CSP events	*Alter* class
Guarded external choice of CSP/*occam-pi* communication channels	*Alternative* class of JCSP
B+CSP process, and the recursion	JCSProB thread and process classes
Process compositions of B+CSP, e.g. parallel and sequential composition	JCSProB process composition classes

TABLE 5.2: The Java Implementation of B+CSP model

There implementations of B+CSP event in JCSProB is a pure Java approach. This implementation was designed and developed before the recently published JCSP package with *AltingBarrier* and rendezvous. It includes a number of event interfaces and classes which are constructed without using any JCSP. All the semantics on the B+CSP event, such multi-way synchronization, are directly implemented in Java language. In Section 5.3, we demonstrates the structure of the event interfaces and classes. Then we continue with a solution for multi-way synchronization in Section 5.4.

As the process class *BCSProcess* is an abstract class which implements the *CSProcess* interface of JCSP, JCSP channel objects can also be used in an *BCSProcess* object. That means the *BCSProcess* can use JCSP channels and JCSProB events at the same time. The only problem is that external choice function for the B+CSP event classes is implemented in a new class *Alter*, which is different from the *Alternative* class of JCSP. Therefore, we cannot support a model which has JCSP channel classes and JCSProB event classes of the pure Java implementation as the first events of external choice paths at the same time. For example, if *ev* is a combined B+CSP event and *ch* is a pure CSP communication channel, the following CSP process P cannot be implemented with JCSProB event classes:

$$P = ch \rightarrow P \,\Box\, ev \rightarrow P$$

In the B+CSP semantics, a B machine is regarded as a special process which maintains the system state. The combined B+CSP events update the state variables and change the system state. The data transitions inside a B+CSP event need to be atomic, which means they cannot be interrupted by data transitions from other events. Therefore, although all the events are on offer in parallel, only one of them at a time can progress and its execution cannot be interrupted. Also, a B machine defines a number of invariants on the state variables. Any violation of these invariants brings the system into an unsafe state. Although on most occasions, the invariants are used for proving abstract models, it would be very useful if the Java implementation can check them at runtime. In Section 5.6, the implementations of the functions concerning system state, e.g. atomic access and the invariant check, are presented.

With the classes mentioned above, a B+CSP model can be translated into a Java program. However, the generated program just runs alone by itself and cannot be controlled. One important target for this work is using the generated Java programs to produce various traces, and comparing the traces with the traces in the B+CSP model. To control the execution of generated Java programs, a GUI interface is developed for the target Java program. Users can setup a configuration file to provide additional information for the Java program, and can use the GUI interface to control the execution of underlying JCSProB processes. Finally, in section 5.7, we demonstrate how the GUI program is constructed, and the communication protocol between GUI program and underlying Java programs.

5.3 B+CSP Event Classes

The implementation of semantics of the combined B+CSP event includes a series of Java interfaces and classes.

One basic Java interface is *BCSPGuard*, which declares several methods used for implementing guarded external choice. All event classes need to implement this interface and its methods. In Section 5.4.3, implementations of this interface is discussed in detail.

An abstract *PCChannel* class provides a method for setting the number of synchronizing processes on the event(similar to the way to set enrolled process number of JCSP barrier), and some other methods for basic event functions. This class is inherited by all event classes, including two direct subclasses: *CCChannel* and *ICChannel*. Both of them implement external choice, precondition check and synchronization for the combined event. The difference is that the *CCChannel* class is designed for implementing events without input data, while the *ICChannel* implements events with input data.

The *CCChannel* class has two subclasses *CChannel* and *OutCChannel*, and the *ICChannel* class has two subclasses *InCChannel* and *OutInCChannel*. The four new classes

implement ready calls for different input/output combinations. The *OutCChannel* and *OutInCChannel* classes also provide methods for producing output data for events. With the four basic event classes, users can construct implementations for combined B+CSP events.

Classes	Input Data	Output Data
CChannel	no	no
InCChannel	yes	no
OutCChannel	no	yes
OutInCChannel	yes	yes

TABLE 5.3: Basic event classes and their input/output types

To construct an event class for a combined B+CSP event, the first thing is to choose a basic event class from the four according to the input/output type of the event. The implementation class would inherit the chosen basic event class, and implement or extend some methods of it. The *run* method must be implemented by the implementation class with the data transitions of the B part of the combined event. If the combined event has input data, the implementation class needs to implements the *assign_input* method for assigning input data with input variables of the event. If the event has output data, the implementation class needs to implements the *make_output* method for make a Java vector, which contains all the output data. Furthermore, if the event has a precondition on it, the precondition needs to be implemented using Java conditional statements, and put in the *preConditionCheck* method.

5.4 Implementing Multi-way Synchronization with Choice

In Section 3.3.1 and Section 5.1.1, we addressed the problem of multi-way synchronization in JCSP. In this section, we discuss the difficulties in implementing multi-way synchronization, and give our solution for this.

5.4.1 The Difficulty in Implementation

Normally, without considering external choice, when a process calls an event, no matter whether the event is ready, it commits to the progress of this event, and cannot withdraw its call. For an event with more than one process synchronizing on it, a calling process would block when the whole set of processes is not ready. For example, a philosophy process *PHIL* synchronizes with a fork process *FORK* on events *picksup* and *putsdown*.

$$MAIN = PHIL\ [|\{picksup,putsdown\}|]\ FORK$$
$$PHIL = picksup \rightarrow eats \rightarrow putsdown \rightarrow thinks \rightarrow PHIL$$
$$FORK = picksup \rightarrow putsdown \rightarrow FORK$$

When process *PHIL* calls the event *picksup*, it commits to the progress of this event and cannot withdraw its call. When both *PHIL* and *FORK* processes are ready, the event can progress.

For guarded external choice, only one process path is selected from all the paths. The external choice structure makes non-commit calls to first events of all the process paths, and the decision is only made among the paths whose first events are ready to progress. When evaluating the first event of a choice path, although the current process is ready to progress the event, it does not commit to it. A process can withdraw its ready call on an event even after it previously became ready for the event. After the decision is made, the process would commit to progress the selected event, and would not go for first events of all the unselected paths.

Furthermore, when an event has more than one process synchronizing on it, and it is used in external choice, the situation is more complex. The choice decision on one event in a process would affect the decisions on that event in other processes.

- If an event is selected by one process, it must make sure that all the other processes which synchronize on this event also know the decision and select the event.
- If a process withdraws from an event, it must make sure that the other processes which synchronize on this event are aware of the change and cannot select this event after it withdrew.

Figure 5.1 shows a version of the classical dining philosophers model that demonstrates this situation.

$$
\begin{aligned}
&MAIN = PHILS[|\{picksup,putsdown\}|]FORKS\\
&PHILS = |||x{:}0..4@PHIL(x)\\
&FORKS = |||x{:}0..4@FORK(x)\\
&PHIL(x) = picksup.x.x \rightarrow picksup.x.((x+4)\%5) \rightarrow eats \rightarrow\\
&\qquad\qquad putsdown.x.((x+4)\%5) \rightarrow putsdown.x.x \rightarrow thinks \rightarrow PHIL(x)\\
&\qquad\qquad \Box\\
&\qquad\qquad picksup.x.((x+4)\%5) \rightarrow picksup.x.x \rightarrow eats \rightarrow\\
&\qquad\qquad putsdown.x.x \rightarrow putsdown.x.((x+4)\%5) \rightarrow thinks \rightarrow PHIL(x)\\
&FORK(x) = picksup.x.x \rightarrow putsdown.x.x \rightarrow FORK(x)\\
&\qquad\qquad \Box\\
&\qquad\qquad picksup.((x+1)\%5).x \rightarrow putsdown.((x+1)\%5).x \rightarrow FORK(x)
\end{aligned}
$$

FIGURE 5.1: The Dining Philosophers Example

There are five philosophers in this story. They are sitting around a table, and there are five forks, each placed between two neighbouring philosophers. A philosopher may only eat when he holds two forks from both left and right sides. When a philosopher is ready to pick up a fork, he has to make a decision on which fork he would pick if the forks from both sides are ready. A fork can also be picked up by either its left or its right

side philosopher. A *picksup.i.j* event is synchronized by a *PHIL* process i and a *FORK* process j. When process *PHIL(1)* is ready to pick up a fork, and both *FORK(1)* and *FORK(0)* are ready to be picked, that makes both of the two events *picksup.1.1* and *picksup.1.0* ready to progress. However, event *picksup.1.1* is not only in the external choice of process *PHIL(1)*, but also in the process *FORK(1)*. If one of the two processes, *PHIL(1)* or *FORK(1)*, makes its decision, it must let the other process know the result. That means:

- If *picksup.1.1* is selected, we must make sure both of the two processes, *PHIL(1)* and *FORK(1)*, select *picksup.1.1*.
- If the process *PHIL(1)* selected *picksup.1.0*, not *picksup.1.1*, it must let process *FORK(1)* know that the process *PHIL(1)* has withdrawn from the event *picksup.1.1*, and the event is not ready anymore.

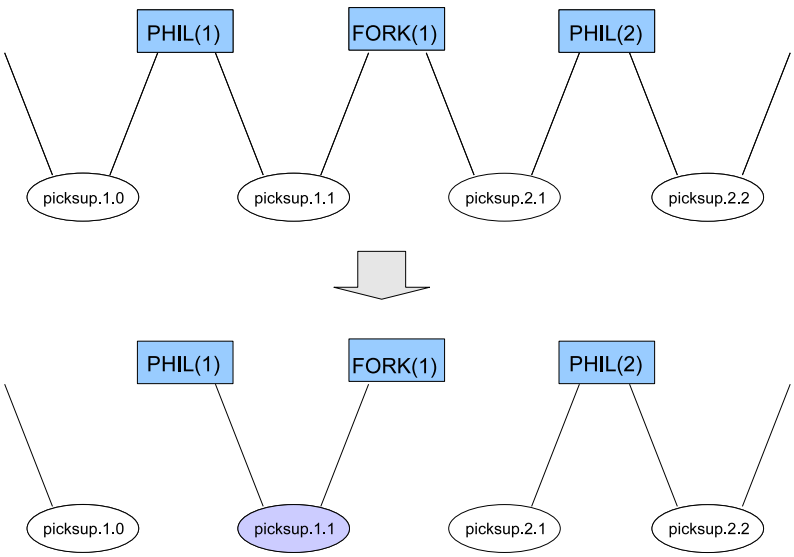

FIGURE 5.2: The External Choice involving processes *PHIL(1)* and *FORK(1)*

Since the choice decisions in processes *PHIL(1)* and *FORK(1)* are produced separately in two parallel processes, it is not very easy to make the algorithm correct. If the decisions in the two processes are made independently, the two processes cannot know each others' decision. If *PHIL(1)* selects the event *picksup1.1*, it cannot make sure that *FORK(1)* also selects the event, which does not make sense as the progressing of an event means that all the synchronizing processes should commit to it. To avoid this problem, when making choice decision, a process must consider the decisions on other processes which synchronize on the event selected by the current process. And when the decision on this process is made, it should also let all the related processes know the result. However, the order of the decisions is important as well. As the processes are

running in parallel, it is still possible that *PHIL(1)* chooses the event *picksup.1.1*, while at the same time *FORK(1)* decides to go *picksup.2.1*, which leaves the event *picksup.1.1* not ready. Figure 5.3 shows such a kind of situation.

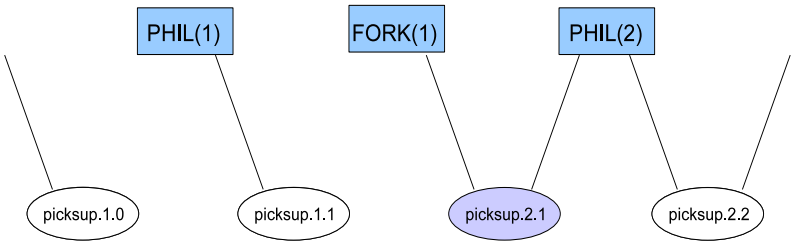

FIGURE 5.3: *PHIL(1)* commits to an unready event *picksup.1.1*

5.4.2 Existing Solutions on Implementing External Choice

In classical *occam* and JCSP (1.0rc5), only the channel input can be used for external choice. The point-to-point communication channel only involves two parties (readers and writers), which means the channel output is not considered when making the choice. A process first informs the input channels of all the choice paths that it is ready for the synchronization. When the input guard of a channel is ready, the channel is regarded as a ready channel for selection, no matter whether the process on the output end of the channel is ready or not. Also, only the input end of the channel can withdraw the previous offer to the channel. As the implementation of external choice here only considers one end of the channel, a choice decision in one process would not affect the decisions in other processes. This constraint makes the implementation of external choice easy and fast, but as output guard and multi-way synchronization channel cannot be used for external choice, this restricts the concurrency models that can be expressed.

In [McE06], a two-phase commit protocol is introduced. The protocol only uses the point-to-point communication channels to construct a multi-way synchronizing channel, and only uses input guards of the channels in external choice. Figure 5.4 demonstrates the state machine of resolving external choice with multi-way synchronization. An external choice has m paths, and a client P_i represents a process, which makes withdrawable call on the multi-way synchronizing event e_i of a path i ($i \in 1 \,..\, m$). At the initial state S_1, the client uses the *offer* message to indicate that it is ready for the execution of event e. Once the offer has been made, in state S_2, P_i can either be interrupted by an event *interrupt* (by the selection of other paths), or commit to the synchronization. If the event *interrupt* occurs, which means the process will not select this path any more, the *withdraw* event withdraws its previous offer. Otherwise, when the client commit to the synchronization, it can either do the synchronization if the synchronization check of event e is ready, or release from its commitment.

In the implementation, two one-to-one communication channels are employed between a client and the event e to implement this protocol, and only the channel input ends of the two channels are used in external choice. Although the two-phase commit protocol is a correct implementation of the external choice with multi-way synchronization, the cost of this solution can be heavy and unpredictable. As all the participating clients P_i of the event e are running in parallel, it is possible that each client keeps choosing to commit to an event, while the other participating clients have withdrawn. Even if this worst case does not happen, the cost for computing this algorithm may be significant.

In [WBP06], a new algorithm is developed to resolve external choice with multi-way synchronization. It is a fast implementation which is not a two-phase commit. Currently, this solution needs shared memory, and is supported only for a single JVM. This algorithm is used to implemented external choice for *AltingBarrier* [WBM+07].

5.4.3 A Solution of Multi-way Synchronization

We argue that the parallel order of making the choice decisions on multiple parallel processes is the main problem to bring the overload. When a process is making a choice decision on an event, a choice decision on another process may change the synchronization state of this event. Although the two-phase commit protocol can prevent the parallel choice decisions from going wrong, the parallel recursive commit/release (or offer/withdraw) actions are not the fast and simplest way to do it. If we can keep a section of code which makes the choice decision atomic, and do not allow decisions on other processes to interrupt it, we can easily prevent a lot of extra overload discussed above. That also implies that the choice decisions in this code section are in a sequential order, instead of a parallel order. In [Bag87, Bag89], such a kind of algorithm has been developed. Our implementation uses a very similar algorithm to [Bag89].

The implementation of multi-way synchronization guards in JCSProB has an exclusive lock for every B machine. All the processes in a B+CSP model need to compete with each other to get the lock, before they can start evaluating synchronization guards

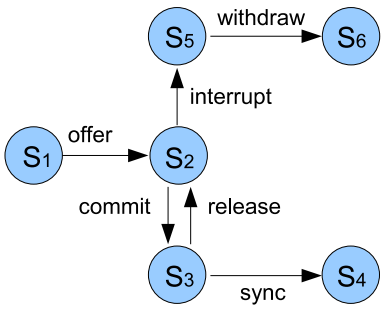

FIGURE 5.4: The state machine of a client P_i

and making choice decision. This lock mechanism forces the choice decisions on a single machine to be atomic and in sequential order. Therefore, it can avoid the cost of parallel *commit-release* actions in the two-phase commit protocol.

The algorithm consists two parties: a set of processes P and a set of event E. The process set P consists a number of parallel processes, which are marked from p_1 to p_N.

$$P = p_1 \parallel p_2 \parallel .. \parallel p_N$$

Such a process p_i $(i \in 1..N)$ may have a M-paths external choice structure. The first events of all these M paths in this external choice are grouped into an event set E, and these events are marked from e_1 to e_M. For a path j $(j \in 1..M)$, the choice is resolved on the first event e_j. After the process performed event e_j, it would further progress as the process res_j.

$$p_i = e_1 \rightarrow res_1 \,\Box\, e_2 \rightarrow res_2 \,\Box\, ... \,\Box\, e_M \rightarrow res_M$$

Figure 5.5, from the view of a process p_i, briefly illustrates the work flow on dealing with an external choice consisting three paths(M = 3). From the initial state S_i, the process first uses the *ready* message to make the first-phase commit calls on all the events in E. The first commitment just increases the event counter, which indicate if the event is ready to progress. Then it cyclically tests these events and looks for a choice decision. The cycle would eventually terminate after all the involved processes start to commit to the events. This will be discussed later in this section. At each state S_j $(j \in 1..M)$, a series of checking steps are applied to test if the event e_j is ready. These checking steps are discussed later with Figure 5.6. If the event e_j is ready, it is selected with message *select(e_j)*. Then the p_i system reaches its end state S_e, and the external choice is resolved. Otherwise, the next event $e_{j\%M+1}$ will be tested.

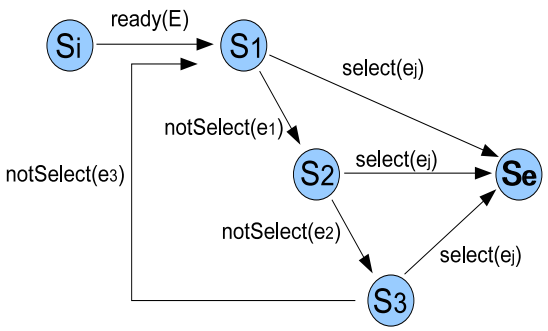

FIGURE 5.5: The state machine a process p_i with three choice paths

The checking steps mentioned in the last paragraph checks the synchronization status of an event e_j. Figure 5.6 shows how such a checking works on an event e_j. From the view of an event e_j, the process p_i represents one of those *client* processes which synchronize on it. The states C_1 to C_6 here represent the internal states of a state S_j ($j \in 1..M$) in Figure 5.5. States C_t and C_s and both represent the terminal state S_e in Figure 5.5, where a resolution has been reached. At the state C_s, the external choice is resolved by selecting the event e_j, whereas at C_t, a different event e_k, other than e_j is selected as the result. The state C_a represents the state where no resolution has been reached during the checking.

Previously, the *ready* message informs the event e_j that the process is ready for the synchronization. The global lock is a key unit for this algorithm. The first commitment *ready* message does not need to obtain the lock before it can make the commitment. But for the second commitment, before actually calling the event e_j, the client needs to obtain the exclusive lock first using the *getLock* message. After the call is finished, no matter whether the event progresses or not, the client needs to *unlock* the lock. This means the shadow region in Figure 5.6 is an exclusive section, and at a time, only one such section can progress.

At state C_2, the client can either be informed for the choice decision and directly go to terminal states (C_s or C_t), or try to make the second-phase commitment to the event e_j:

- The event *interrupt1* informs the client that the synchronization of the event e_j was satisfied, and selected by one of the other involved processes. The event e_j has progressed, and this client p_i has been withdrawn from all the events in the external choice. So the current process would go to state C_s, and performs as res_j.
- The event *interrupt2* informs the client that a different event e_k ($k \in 1..M, k! = j$) in this M-paths external choice, was selected by another process. The event e_k has progressed, and this client has been withdrawn from all the events in the external choice. So the current process would go to state C_t, and performs as res_k.
- When no interruption messages has been received, the client tries to make the second-phase commitment call to e_j.

Before make the second-phase commitment, the client first needs to check if the synchronization status of e_j is ready at state C_3. This check performs as a method call on the event. The event would check if the synchronization counter has reached its capacity and if the precondition on the event is satisfied. Only when both conditions are satisfied, the event would return a true value to the client. If yes, the client would finally *select* the event as the result of the external choice. Otherwise, the client releases the lock, and reaches the state C_a, where no resolution has yet been reached. The exit

from C_a represents state transitions in from one checking state S_j to a new checking state $S_{j\%M+1}$ in Figure 5.5. It lets the alternative paths in the external choice structure to progress.

After the event e_j is selected, as all the other candidate events in E cannot be selected in this choice, the process p_i should withdraw itself from these events. The *ack* message is received from the event, which is discussed in next paragraph. Between the *select* and *ack* messages, the event also informs other clients, which synchronize on event e_j, about the choice decision.

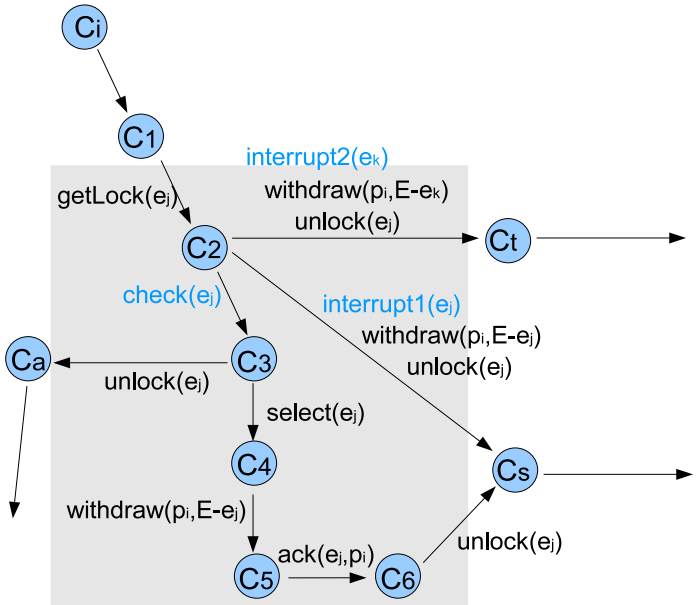

FIGURE 5.6: The state machine of a client E_i

A multi-way synchronizing event is an important unit in this algorithm. In JCSProB, the combined B+CSP event is implemented more like a special process, than a communication channel. It maintains the commitment records of all involved processes by receiving commitment messages from them.

The process set on an event e is Q_e. The event reacts to the commitment calls from external choice clients. Figure 5.7 shows how an event e reacts to commitment messages. After the event receives the first-phase commit call (*ready* message) from a client q ($q \in Q_e$), it increase the synchronization barrier by 1. After that, it checks the barrier to see if all the synchronizing processes in Q_e are ready. If the result is yes, the event is selected and finally committed. Then the decision on the event e needs to be broadcast to all the other processes ($Q_e - q$). The broadcast will not race with broadcasts from other events as at the time of broadcasting, the exclusive lock is hold by the client who interacts with the current event. The *inform* message sent to those processes would cause *interrupt1* or *interrupt2* actions in them. After broadcasting the selection, the

event moves on to processing the communication or data transitions inside the event e, and then resets the synchronization barrier. Finally, it sends an acknowledgement back to the calling client and waits for new commit calls.

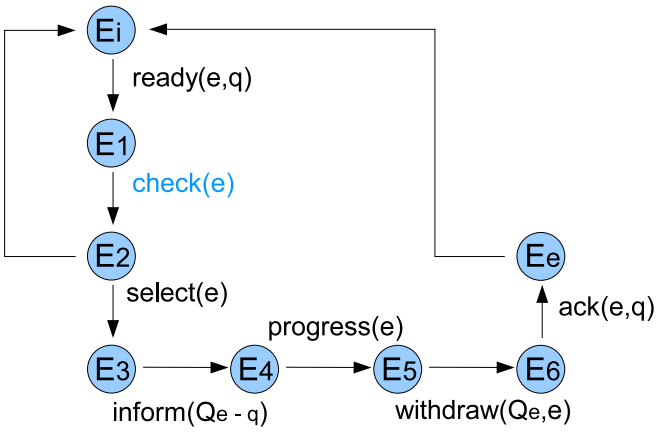

FIGURE 5.7: The state machine of an event E processing commitments

5.4.4 An Example: Dining Philosophers

In Section 5.4.1, we discussed guarded external choice and multi-way synchronization in the dining philosophers example of Figure 5.1, and the difficulties in resolving them together. Here we demonstrate how to employ our solution to resolve the external choice in dining philosophers, and how it can avoid the erroneous situation showed in Figure 5.3.

The process *FORK(1)* consists of two possible paths. The events *picksup.1.1* and *picksup.2.1* are the first events on the two paths. *FORK(1)* needs to synchronize with *PHIL(1)* on *picksup.1.1*, or synchronized with process *PHIL(2)* on *picksup.2.1*. The process *FORK(1)* first makes the first commitment calls, the *ready* messages, to both of the two events. If the two *PHIL* processes, *PHIL(1)* and *PHIL(2)*, also made the *ready* calls on the two events, that makes the synchronization status of the two events both ready. Figure 5.8 shows the ready calls on the two events.

After that, as we showed in Figure 5.5, a process would try to recursively check the synchronization states of all the candidate events. Before it can start the checking, it needs to obtain the exclusive lock first. Figure 5.9 demonstrates that *PHIL(1)* starts with requiring the lock for checking *picksup.1.1*, while at the same time, *FORK(1)* tries to get the lock for checking *picksup.2.1*. The two *getLock* actions compete with each other, and in this occasion, the *PHIL(1)* gets the lock.

From Figure 5.6, we know that after getting the lock, a process would start to check the synchronization status of the event. In this example, as the *picksup.1.1* is ready for synchronization, the checking succeed, and *picksup.1.1* is selected as the solution for this external choice. The first two steps in Figure 5.10 illustrate this action.

Also in Figure 5.6, after the choice decision has been made, the process client withdraws its first commitment on other candidate events. Therefore, the *PHIL(1)* process here withdraw itself from *picksup.1.0*. At the same time, the event unit *informs* other client processes about the choice decision. Such an action would later cause an *interrupt* message in these processes. After the third step in Figure 5.10 finished, both *PHIL(1)* and *FORK(1)* are aware of the choice decision. Then finally, the selected event *picksup.1.1* can progress, and for a combined B+CSP event, this means the data changes inside the event can be processed now.

When the execution of the event is accomplished, the event withdraws all the client processes on it through the $withdraw(Q_e, e)$ messages. In Figure 5.11, the *picksup.1.1* event withdraws commitments from the *FORK(1)* and *PHIL(1)* processes. Then there is an internal synchronizing step, in which the event sends an acknowledgement back to the calling process. After received the acknowledgement, the process *PHIL(1)* releases the exclusive lock.

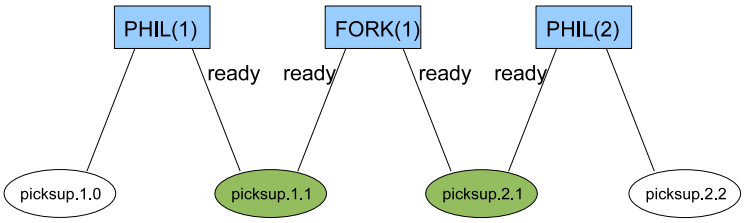

FIGURE 5.8: The dining philosophers: ready calls

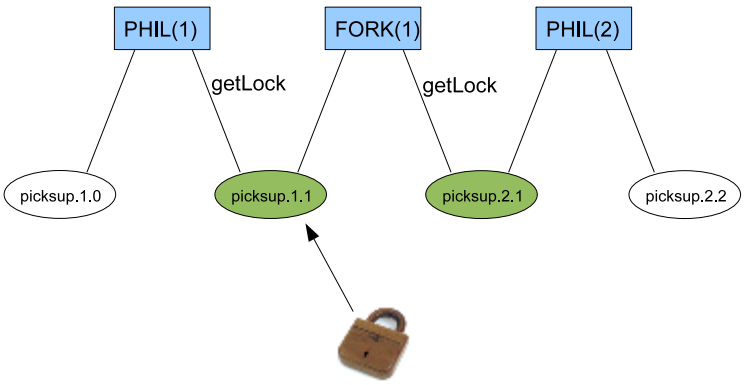

FIGURE 5.9: The dining philosophers: compete for the lock

When the *FORK(1)* process obtains the lock later, the *interrupt* message informs the process about the choice decision, and that leads the process to withdraw its commitment to other events. In Figure 5.12, *FORK(1)* withdraws its first commitment to event *picksup.2.1*, and continues to perform as alone the choice path led by *picksup.1.1*.

Finally, the system enters a state showed in Figure 5.13. Both *FORK(1)* and *PHIL(1)* agree to select the path led by *picksup.1.1*, and both of them withdrew their first commitment calls on other candidate events.

The lock mechanism prevents the situation showed in Figure 5.3 to happen. It puts the *FORK(1)* process waiting while *PHIL(1)* are checking on event *picksup.1.1*. When the choice decision comes out, *PHIL(1)* is also aware of that. Therefore, it cannot select *picksup.2.1*, as *PHIL(1)* decided to take *picksup.1.1*. Also, comparing with the two-phase commit protocol, there could only be one commit-withdraw action in this algorithm, and at a time, there is only one second-commit action can progress. There cannot be a situation that multiple processes keep on committing and releasing, but never get a result.

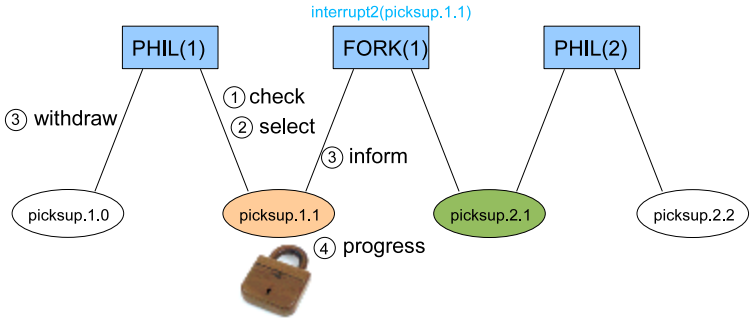

FIGURE 5.10: The dining philosophers: *picksup.1.1* is selected and progress

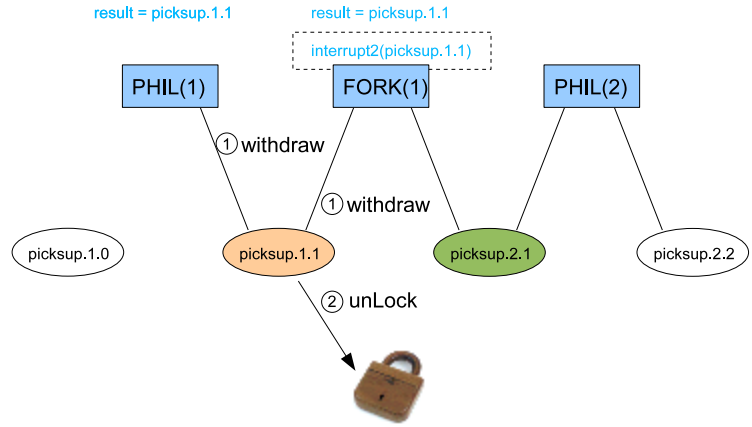

FIGURE 5.11: The dining philosophers: withdraw and unlock

However, a limitation for this algorithm is that it can only be applied on a single memory, and cannot be directly applied on distributed systems. In [WBM+07], similar restriction is placed on the *AltingBarrier* of JCSP. Both algorithms try to make sure there is no redundant commit-withdraw action, which means all the decisions of the final commitment must be made in a sequential order. To ensure this, there has to be a global facility, such as a lock, which can be accessed by all the clients. The implementation on a distributed environment needs an exclusive token to be passed through all the subsystems to make sure all the decisions lie in a sequential order.

A part of Java code, which includes several major classes in this example, is presented in Section B.2.

5.5 Process Classes: Thread, Process Calls and Recursion

In Section 5.1.2, we briefly discussed the differences between the management of recursion in JCSP/*occam-π* and CSP. CSP (through Java) and *occam-π* allow recursion in their processes, so any CSP recursive expressions could be directly implemented. However, unbounded recursion will lead to memory overflow. Hence, *tail* recursion in CSP

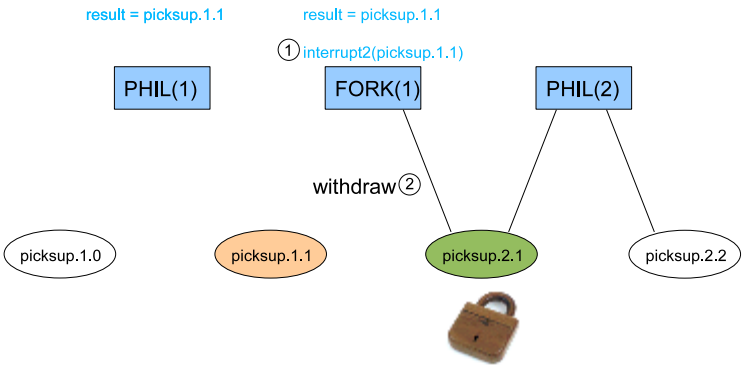

FIGURE 5.12: The dining philosophers: interrupt in *FORK(1)*

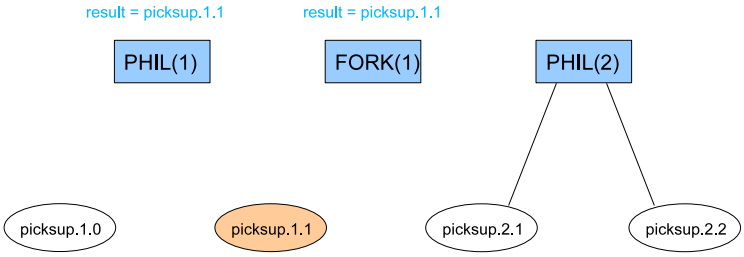

FIGURE 5.13: The dining philosophers: final state

(which is very common) is always expressed through (`WHILE`) loops, which cause no memory problems. More complex recursions require more complex transformation if they are to be made memory safe.

5.5.1 Calling a Process in JCSP

In *occam* and *occam-π*, it is not possible for a named process to call a new named process or even a new instance of itself in the way that CSP can. Conventionally, JCSP translates the **WHILE** statement of *occam-π* into a Java ***while*** statement. The loop steps in a recursion take place in a single JCSP process object, without introducing any new process objects. For example, a CSP process

$$P = a?x \rightarrow b!x \rightarrow P$$

would be translated into a JCSP process class as:

```
class P implements CSProcess{
      ......
      void run(){
            while(true){
                  x = a.read();
                  b.write(x);
            }
      }
      ......
}
```

To support more general CSP recursions and process calls, an existing JCSP process should be able to call a new process object. This requires the existing process object to declare a new process object and call the *run* method of it.

```
class P implements CSProcess{
      ......
      void run(){
            x = a.read();
            b.write(x);
            new P(a,b).run();
      }
      ......
}
```

However, it is very dangerous to do this in JCSP, because the old process object cannot be released while the new process object is running. Figure 5.14 demonstrates the call on a new process object P' from the existing JCSP process object P. The new process object P' runs in the same Java thread *thr* as the existing process object P, which would not release until the new process object complete. Recursively doing that may

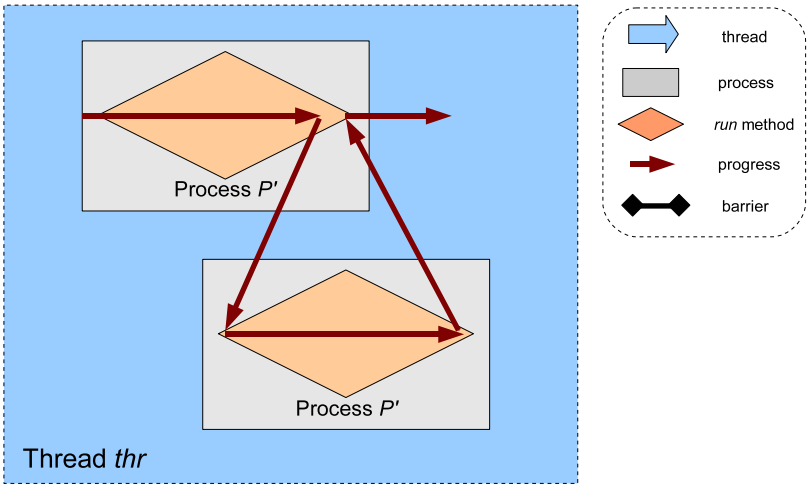

FIGURE 5.14: Calling a new process object in JCSP

cause a Java error, as JDK has a limitation on the number of objects. If new process objects are recursively produced in this way, it would generate too many process objects without releasing them. Eventually, this would arose the Java error *StackOverflowError*! Therefore, we cannot directly use the current JCSP process mechanism to implement the recursion and process calls in CSP. In CSP, when the process P called a new process P', the process should continue to perform as the process P', and the system would have no further concern with the existing process instance P. In the Java implementation, it means that after calling the new process object, the existing process object should be able to release itself to Java garbage collection, and the Java thread would perform as the new process object. At a time, there should be only one process object running. As the *CSProcess* interface does not provide this function, we need to implement it in the JCSProB package.

5.5.2 Multi-threading in JCSP

Direct tail recursion in CSP is handled in JCSP through `WHILE` loops. If the process being invoked at the tail of an equation branch is not the process being defined, indirect mutual tail recursion may be happening – for example, the Lift system in Figure 4.4. Implementing such equations directly using Java recursion and JCSP will lead to stack overflow. Transforming out the mutual recursion to leave a direct *tail* recursion (implementable through a loop) will distort the expression of the process and may lead to

errors and maintenance problems. Here, we consider two ways to implement mutual tail recursion directly and with no memory problems.

- The first way is to allow replacing the caller process object P with the callee process object P' in the same Java thread container. There is a synchronization barrier in a thread which is used for inform the environment about the termination of this thread.
- The second way is to create a new thread container T' for the new callee process object P'. The caller process object P and its container thread T terminate after the new thread starts. There is also a synchronization barrier in every thread container. If the current process creates no new process, the thread container calls the barrier when it terminates. Otherwise, to keep the new thread T' reachable for the environment, the synchronization barrier in T should be passed to T'.

In JCSP, the *ProcessManager* class, which provides a new thread container, implements this second solution already. The *ProcessManager* creates a new thread container to run a new process. The user needs to call the *start* method to run a *ProcessManager*.

In our implementation, we employ the first solution because it creates fewer objects and is less complex in implementation. To implement each of the two solutions, we need to have the access of the multi-threading mechanism in JCSP to make the change. However, this is not so easy to achieve.

As JCSP targets producing concurrent Java programs, it makes use of the Java multi-threading architecture. JCSP package has a thread class *ParThread* for running JCSP process objects. The *ParThread* class only appears in the *Parallel* class, which implements the parallel composition of processes. Process classes, and other processes composition classes, cannot affect the behaviour of their container threads, which means the idea of replacing the caller process object with the callee process object in a thread cannot be supported under the current multi-threading architecture of JCSP. Furthermore, not all processes are running in instances of the *ParThread* class. The core process of a JCSP program runs in the main Java thread of this program. A direct and easy solution for implementing the switch between the caller and callee processes is to extend the *ParThread* class and implement the existing *CSProcess* interface with the function of processes switching inside a thread. This solution would allow us to continue using the process composition classes provided by JCSP.

However, as the multi-threading of JCSP is not designed to implement the CSP process, it is not possible to reuse this threading facility to implement the CSP process. The *ParThread* class is a *package-private* class of the JCSP package. It is not allowed to be accessed from the outside of this package. Although JCSP is an open source package, it is not feasible to change the existing access control of it. Furthermore, the idea of

replacing the caller process object with the callee process object in a thread also cannot be support by processes composition classes in JCSP, e.g. *Parallel* and *Sequence*. The *Parallel* class is the only class in JCSP which explicitly use the *ParThread* class. For a parallel composition in a process P which includes N processes, the *Parallel* class generates *N-1* new *ParThread* threads. It runs the first *N-1* processes in the *N-1* new threads, and uses the container thread of the current process to run the last process P_N. All the threads in a *Parallel* composition share a common barrier. When a thread finishes the run of the process object it contains, it calls the barrier for synchronization. After all the threads and the process in the container thread synchronized on the barrier, the parallel composition finishes. Figure 5.15 shows a parallel composition with three participating JCSP processes.

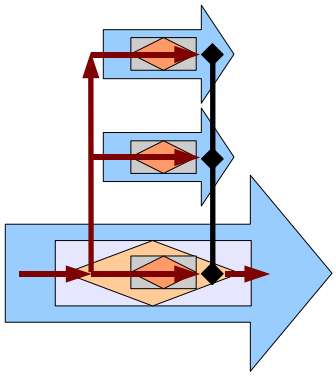

FIGURE 5.15: Parallel composition in JCSP

If the last process object P_N in the parallel composition calls a new process object P_X, it is not correct to use P_X to replace the current process in the thread, because the actual process running in the thread is P, not P_N. If P_X replaced P in the thread, we would lose all the information about P, which hasn't finished yet. Especially, as the barrier and the parallel composition are defined in process P, the barrier synchronization of between the parallel processes would also gone, which would leave the runs of the parallel threads/processes uncontrollable.

Similar problem also can be found in the sequential composition class *Sequence*. The *Sequence* class does not directly access the *ParThread* class. A sequential composition in a process P runs the participating composition processes one by one inside itself. Figure 5.16 shows a sequential composition including three participating processes. If one of these processes calls a new process object P_X, it is also dangerous to use P_X to replace the existing process P in the thread.

Accordingly, the multi-threading architecture in JCSP does not support the idea of process switching, and the processes composition implementations in JCSP, e.g. *Parallel* and *Sequence*, cannot adopt to the idea either. Therefore, in order to implement the CSP process with processes switching in Java threads, we need to build new thread and

process classes, as well as the processes composition facilities. In Section 5.5.3, new implementations of the thread and process classes are proposed, and in Section 5.5.4, the parallel and sequential composition classes using the new thread and process classes are discussed.

5.5.3 Implementations of the CSP Process in JCSProB

The implementation of the CSP process in JCSProB consists of two parts: an abstract process class *BCSProcess* which implements the *CSProcess* interface of JCSP, and a thread class *RecurThread* which is the thread container for the new process class.

In the JCSProB implementation of the CSP process, every process object needs to run in its own thread container, which is an instance of the *RecurThread* class. Even for the core process, the Java program needs to produce an extra thread to run it. The *RecurThread* class consists of two fields: one is the *BCSProcess* process object which runs in it, and the other is a JCSP *Barrier* object which needs to synchronize before the thread terminates. Figure 5.17 demonstrates the state machine of a *RecurThread* instance. The constructor of the class passes a barrier object and a process object to initialize a thread object. When the thread starts running, it calls the *run* method of the process object. The run of the process object may produce a new process object. If there is a new process object, the thread gets the new process and runs it. At the same time, the previous process object is released to Java garbage collection. If there is no new process object, the process terminates, and the thread object synchronizes on the barrier before it finally finishes its run.

The abstract process class *BCSProcess* has the function of communicating with its thread container on the new process it has created. When a process object calls a new process object, it does not directly run the new process. Instead, it passes the new process object to the thread container. When the run of the current process is finished, the thread container would checks whether it has produced a new process object.

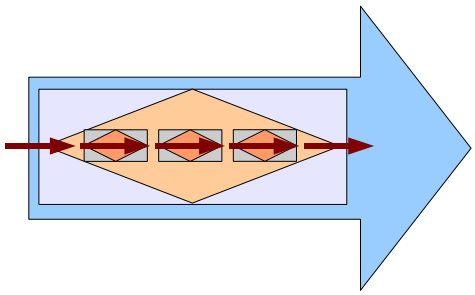

FIGURE 5.16: Sequential composition in JCSP

This new implementation of processes allows a process to call a new process, and safely release itself after the call. Also, as the new process class is an extension of the JCSP process interface, it can continue working with many JCSP classes, e.g. channel classes, except the processes composition facilities. In implementation, the *BCSProcess* class provides a new *callNextProc* method for its implementation classes. What the user need to do is creating a new process object and using it as a parameter to call the *callNextProc* method. When the current process terminates, the thread container can get the new process object and run it. Therefore, the example in Section 5.5.1 is implemented in JCSProB as:

```
class P implements BCSProcess{
      ......
      void run(){
            x = a.read();
            b.write(x);
            callNextProc(new P(a,b));
      }
      ......
}
```

5.5.4 Sequential and Parallel Composition in JCSProB

The new thread class *RecurThread* provides a fundamental thread implementation for the B+CSP process. As the multi-threading mechanism in JCSP does not support this new implemented thread, we need to build a new one for JCSProB. Process composition functions, such as parallel composition and sequential composition, are re-implemented for JCSProB processes and threads.

The parallel processes composition of B+CSP is implemented in the *CSParallel* class. For a parallel composition consisting N processes, the *CSParallel* object generates N new thread objects of the *RecurThread* class to run all the N processes. All the new N threads, and the current thread running the parallel composition, share a barrier whose counter is set to $N+1$. When all the threads terminate, the parallel composition structure completes its run. Figure 5.18 illustrates a parallel composition of three processes.

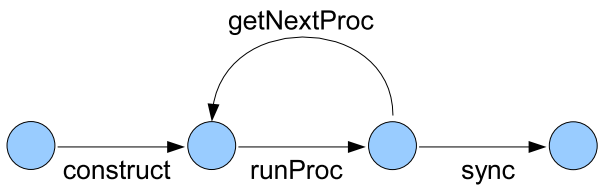

FIGURE 5.17: The state of the *RecurThread* class

This new implementation of parallel composition allows the process objects in all the parallel threads to be replaced by new process objects, this bypassing the problems of the *Parallel* class of JCSP.

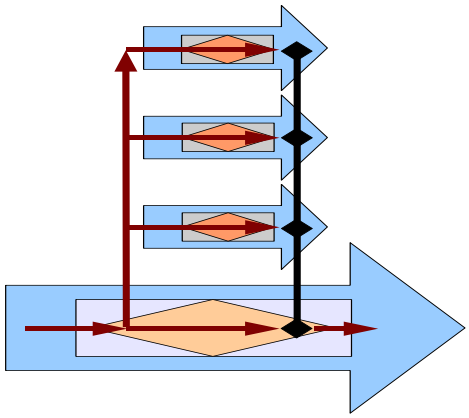

FIGURE 5.18: Parallel composition in JCSProB

The new implementation of sequential composition introduces threads, while the JCSP implementation of it does not create new threads. For every participating process object in a sequential composition, a new thread container is produced. Every thread container shares an unique barrier with the core process. Only after a thread terminates and is synchronized on its barrier, is a new thread started by the composition object. In this way, the participating processes progress one by one, and the process replacing mechanism in the thread class can be used without collision. Figure 5.19 shows a sequential composition object of JCSP with three participating processes.

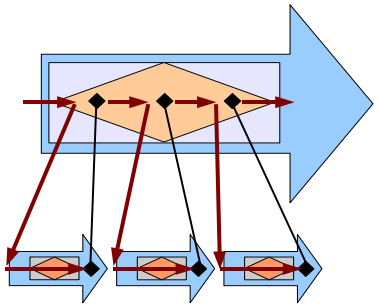

FIGURE 5.19: Sequential composition in JCSProB

The new thread class *RecurThread* and abstract process class *BCSProcess* provide an implementation of the CSP process, while the original JCSP process implementation is more close to the semantics of *occam-π* processes. The new process implementation *BCSProcess* provides a method for calling a new process in an existing process, and supports more flexible recursion structures. Together, the two classes implement the CSP process and its recursion semantics. The process composition classes implements the multi-threading mechanism for the implementation of the CSP process.

Comparing the cost of this solution with the *ProcessManager* class of JCSP, the two solutions produce extra cost on different situations. The JCSProB solution produces one extra thread container and one extra barrier object for each sequential composition. If the child process does not have recursion, the extra thread container is actually unnecessary. In JCSP, all child processes of a sequential composition just run in the same thread container. The JCSP solution (*ProcessManager*) produces one extra thread container and one extra barrier object at every step of a recursion. If there are more sequential composition in the system, using *ProcessManager* would be more efficient; while if process calls and recursions are more frequently used, the JCSProB solution would produce less cost. In Section 8.2.5, we will discuss a possible solution to reduce the cost of JCSProB.

5.6 The State Variable Class

In the B-method, the data transitions of a B operation must be kept atomic in order to preserve the consistency of the state model. The combined B+CSP model also has this requirement. The JCSProB packages provides a *JcspVar* class for implementing this feature in the Java implementation. It explicitly implements an exclusive lock to control the access to the B variables. Only one event object can have the lock at a time. This lock is also used for the implementation of multi-way synchronization and guarded external choice, which is discussed in Section 5.4.

In the implementation of combined B+CSP events, the call on the *run* method of an event object is guarded by the lock. An event object needs to obtain the lock before it can call the *run* method, and it is also required to release the lock after the *run* method finishes.

When constructing a Java implementation from its formal specification, the *JcspVar* class needs to be extended, and all the global B variables should be implemented in the new constructed class.

5.7 GUI

A Graphical User Interface package (*org.dsse.jcsprob.gui*) is designed for concurrent Java programs constructed using the JCSProB package. It is a sub-package of JCSProB, and consists of a number of Java graphical components corresponding to JCSProB events and processes. When developing a concurrent Java program using JCSProB, users can build a graphical interface for the underlying Java program, and use the constructed GUI to control the execution of the Java program. It also provides runtime checking

for invariants of B machines, as well as external programming interfaces for runtime assertion checking.

In [MK99], most Java examples are presented with GUI. However, those GUI programs are manually constructed. Although they are used to demonstrate the state models, there is no formal association between them. In [FC06], the automated translator for *Circus* generates Java programs with GUI. The GUI program simply presents all *Circus* channels as buttons. The system information presented for interaction is very limited.

5.7.1 Overview of a GUI Program

Figure 5.20 demonstrates the structure of a GUI program, which includes four major parts:

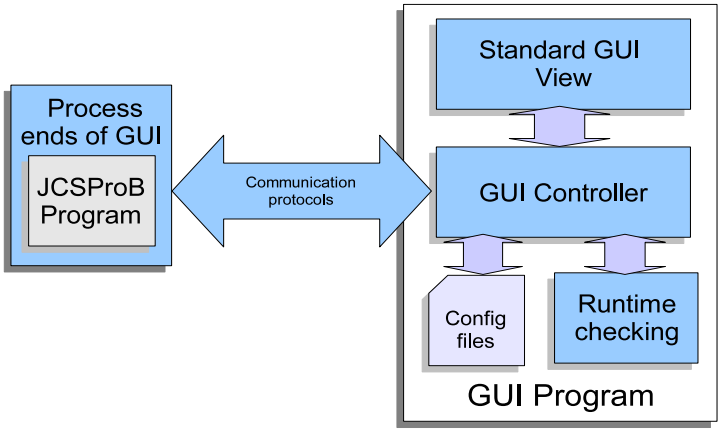

FIGURE 5.20: The structure of a GUI program

- The process end of the GUI programs.
- A GUI controller unit.
- Communication between the the processes end and the GUI controller.
- A standard GUI view of the B and CSP model.
- Configuration files.
- Runtime assertion checking interface.

The modules represented as blues squares are components from the GUI package. The items in light gray color, including the *JCSProB* program and the configuration files, can be derived from combined *B+CSP* models. The automated translation discussed in Section 6 provides the mechanism for generating *JCSProB* programs with the process

end of GUI, and the configuration files, which carry further information from the formal models.

The configuration files, composed in XML, contain various kinds of information, such as, a succinct version of the CSP specification, runtime assertions, and settings of visual components. Users are allowed to change some settings in the configuration file to control the GUI, while many other settings, which concern the expression of B and CSP models, are not allowed to be changed. The *configuration.xml* contains the information for communication messages between the GUI controller and the process end, and runtime checking assertions. Generally, communication messages in the configuration file are not allowed to be changed, unless the user wants to display his own message in the on-screen log module. For a combined model with the B machine name *machine*, a configuration file *machine.xml* is generated by the translator. It contains the CSP specification and default GUI settings on the CSP specification. Another configuration file is *machineGUI.xml*, which contains the setting for painting the GUI frame and panels.

To construct a GUI program from Java program using JCSProB, the JCSProB processes are encapsulated in the process end of GUI package. When a JCSProB process tries to perform certain activities, such as enabling a B+CSP event, making an external choice, or starting a number of parallel processes, it uses the process end of GUI to communicate with the GUI controller.

Communication protocols are two-way message passing communications between process end and the GUI controller. They are also implemented as a part of the GUI package (*org.dsse.jcsprob.gui.msg*). Each scenario of CSP process or event activities, for example calling an event, is associated with a certain communicating pattern, implemented both in the process end and the GUI controller. We use JCSP communication channels to implement the communication protocols. The detail of these communication protocols is discussed later in Section 5.7.2.

The GUI controller is the centeral unit of the GUI program as it communicates with all the other four components of the GUI program.

- The controller reads the configuration file, which contains the specification of CSP processes, and uses it as guidance for communicating with the process end. As the CSP specification used by the underlying JCSProB program and the GUI controller unit are from the same B+CSP model, it makes sure that the control unit uses the same communication pattern as the *JCSProB* program.
- The GUI controller communicates with the GUI view to allow the interaction with users. It sends information to the GUI view, and receives users commands and inputs data from it.
- The GUI controller also send out runtime record information to the runtime assertions checking module.

The GUI controller provides a number of interfaces for building GUI views. In the GUI package, we implemented a standard GUI view for all the processes and process operators. However, using the interface provided by the GUI controller, users can construct their own GUI views of processes and events.

The runtime assertion checking module can record runtime data about the Java program through GUI controller. Users can use the configuration file to indicate specific points where the records are needed, and what kind of information are needed in the records. With the runtime record information, the checking module can verify assertions upon the information. In the package, we provide checking for several kinds of assertions. Users can also add their own assertions and runtime checking programs using the provided checking interface. In Section 5.7.3, we discuss this function in detail.

With the support of GUI package, the *JCSProB* program can be regarded as an alternative animation tool for B+CSP models. As the users are allowed to develop their own GUI view, it is also possible to be used for implementation. Moreover, the runtime assertion checking module allow users to evaluate some properties of system models, especially the properties are not easy to specify using B or CSP, e.g. fairness.

5.7.2 The Development of Communication in GUI

The communication between the process end and the controller of GUI is implemented using *JCSP* communication channels. The two sides use communication channels to send and receive messages from each other. The messages sent between them are implemented in a sub-package (*org.dsse.jcsprob.gui.msg*) of the GUI package. Finally, we define the communication protocols upon the orders of communicating messages through communication channels. We developed different protocols for all the CSP semantics elements that we supported for the translation. In this section, we will demonstrate the protocol for an event call.

In a *JCSProB* program, when a process object tries to call a combined B and CSP event, it calls the *ready* method of the event object. The *ready* method may need some input data, and may produce some output data. To construct a GUI object to control the call, we need the GUI object to give permission for calling the *ready* method. That decision can either be made by GUI itself, or an user action on the GUI. Also, to help the user to make decision, the user should be informed about the input data before the call, and the output data afterwards. In the implementation, we use two one-to-one *JCSP* channels for communicating between the process end and the GUI: the *configure* channel is used to send messages from the process end to the GUI, while the *control* channel is used for the opposite direction.

Figure 5.21 shows a very abstract sequence diagram of the calling.

The process first sends a message including the input data IN to the GUI through the *configure* channel. After that, an action X, which can be an user action or an internal GUI action, enable the call of the event, and the GUI uses the *control* channel to send the action message X back to the process end. Having received the message, the process would perform the call on the the event using the *ready* method. When the synchronization on the event is satisfied, the *run* method would be processed. So the event call requires the inpu data IN, and produces output data OUT. After the call finishes, it encapsulates the output data into a message OUT, and send it back to the GUI through the *configure* channel. The CSP specification of this communication model is:

$$System = Actor \underset{\{interaction\}}{\parallel} GUI \underset{\{configure,control\}}{\parallel} Process(State)$$
$$Actor = interaction!X \rightarrow Actor$$
$$GUI = configure?IN \rightarrow interaction?X \rightarrow control!X \rightarrow configure?OUT \rightarrow GUI$$
$$Process(IN) = configure!IN \rightarrow control?X \rightarrow ready.IN.X!OUT \rightarrow configure!OUT \rightarrow Process(OUT)$$

In the implementation of the process end of GUI, we provide an abstract *BGProcess* class. It extends the *BCSProcess* class, and provides a number of methods for implementing the communication to the GUI controller. For the event call, it provides *channelCall* and *channelRtn* methods for calling an event (*channelRtn* for calls with output data). A call *ev.xx* on an event *ev* with input data *xx*, which was translated into *JCSProB* without GUI support as:

```
ev_ch.ready(new Vector(Arrays.asList(xx)));
```

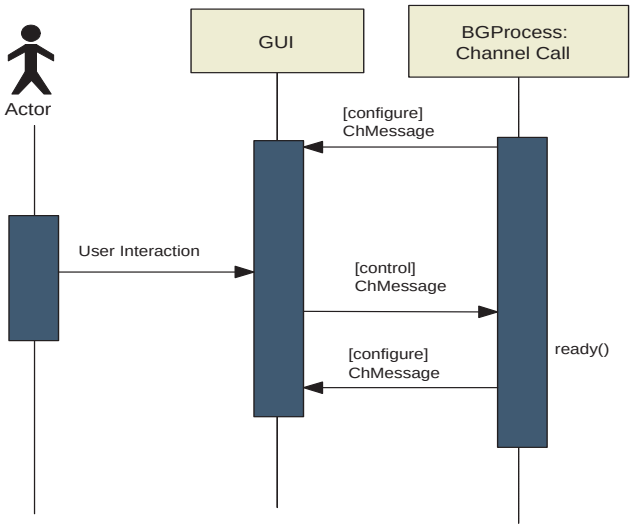

FIGURE 5.21: The GUI communication of event call: level 0

would be translated into a call with GUI control as:

```
channelCall(ev_ch,new Vector(Arrays.asList(xx)));
```

The communication with GUI is implemented and encapsulated inside the *channelCall* method. Therefore, when we try to build a *JCSProB* program with GUI support, instead of extending the *BCSProcess* class, all the process classes should extend the *BGProcess* class. Also, when a process class performs process activities, such as calling events, parallel or sequence compositions, it should make use of corresponding methods provided by the *BGProcess* class. As communications with GUI are well encapsulated inside these methods, a process class with GUI support, which is constructed using the *BGProcess* class and its methods, can be built in a similar structure as the process classes without GUI support, and the Java code is as clean and straightforward as the latter one. The differences are only on syntax level.

However, apart from the process end, a GUI program is more complex than we illustrated in Figure 5.21. Two main parts of a GUI program is the GUI controller and the GUI view. As we aim to separate the implementation of communication protocol from the implementation of GUI views, giving freedom to the user to implement his own GUI component, we clearly separate the two parts. The GUI controller provides certain interfaces and implementation policies for constructing a GUI view module. Figure 5.22 demonstrates an updated version of event calling in the GUI, where the controller and the view of GUI are separated.

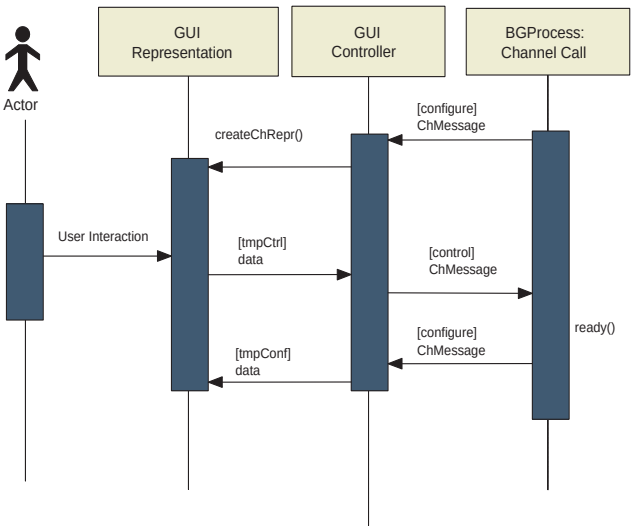

FIGURE 5.22: The GUI communication of event call: level 1

In the Java implementation of GUI, the GUI controller is implemented in an abstract Java class *CtrlProc*. The *CtrlProc* class provides a number of methods which implement

the communication protocol at the controller end. They communicate with corresponding methods in the *BGProcess* class. For example, for the event call method *channelCall* in *BGProcess*, which implements the process end of the communication protocol, the *CtrlProc* also provides a *channelCall* method, which implements the controller end. While the process end is driven by the method calling order in the *run* method of the process class, the controller end uses the CSP specification in configuration files to drive its progress. A call on an combined event *ev* is recorded in the automatically generated configuration file as:

```
<Channel>
      <ChName>ev_chclass</ChName>
      <IOType>InCChannel</IOType>
      <ChType>ChannelCall</ChType>
      <UserControl>false</UserControl>
      <Wait>200</Wait>
      <Record>false</Record>
      <LogInArgs>true</LogInArgs>
      <LogInArgsText>Args In</LogInArgsText>
      <LogOutArgs>true</LogOutArgs>
      <LogOutArgsText>Args Out</LogOutArgsText>
</Channel>
```

The XML file not only includes specifications of CSP processes, but also the configuration for driving the GUI. The $<IOType/>$ element carries the information of input/output type of the event. *InCChannel* here means the *ev_chclass* event only has input data, and its Java class extends the *InCChannel* class. The $<UserControl/>$ element contains the setting for user interaction on the event call. If it is true, user interaction is required for calling the event, otherwise the event is directly called by the GUI. In some cases, we may want the GUI to wait a while before it automatically calls an event. So the $<Wait/>$ element contains the timing for calling the event. The *channelCall* method of the *CtrlProc* class receives the whole $<Channel/>$ element as an parameter, and uses it to instruct the GUI controller and the GUI view.

The GUI view module of a process should be implemented as a subclass of *CtrlProc*. In the *CtrlProc* class, the *channelCall* method starts the GUI view of event calls by calling an abstract method *createChRepr*, which is regarded as an interface for the GUI view module. The GUI process class should implement this method in order to build GUI view of event calls. According to the sequential model, which can be used as an instruction for constructing the *createChRepr* method, the implementation of the method should:

1. Display the data of input variables, as well as a GUI component for user interaction

2. Capture the user interaction
3. Send the data input by the user interaction to the channel *tmpCtrl*
4. Receive and display the data of output variables from the *tmpConf* channel, and display it.

In our standard GUI view implementation, two labels are used to display the data of input and output variables, and a button is used for user interaction with the event name displaying on the button. To activate the call of an event, the user simply clicks the button. Figure 5.23 illustrates a standard GUI view for calling an event *thinking!0*. The Input label here shows the value, 0, of the input argument. The GUI can display any kind of data through the default *toString* method supported by all Java objects.

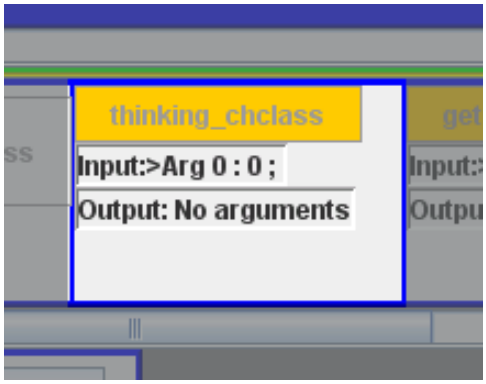

FIGURE 5.23: The standard GUI view of an event call

Also, the GUI controller is in charge of sending the runtime log to the log module, which is a component in the GUI frame to display runtime logs, and the runtime record to the assertion checking module. Therefore, we need to introduce the log module and record module in this model of event calls. Also, in the XML expression of the event call we used above, there are several settings, e.g. *<Record/>* and *LogInArgs*, for sending out log and record information. Figure 5.24 demonstrates the new sequence model of event calls with log and record.

The CSP specification for the final model of this GUI function for event calls would be:

$$System = Actor \underset{\{interact\}}{\parallel} GUI \underset{\{configure,control\}}{\parallel} Process \underset{\{recordOut\}}{\parallel} Record \underset{\{logOut\}}{\parallel} Log$$

$$Actor = interact!X \rightarrow Actor$$

$$Process(In) = createMsg!In?InMsg \rightarrow configure!InMsg \rightarrow control?X \rightarrow ready.In.X?Out \rightarrow createMsg!Out?OutMsg\ configure!OutMsg \rightarrow Process(Out)$$

$$GUI = configure?InMsg \rightarrow logOut!InMsg \rightarrow getMsg!InMsg?In \rightarrow (GUICtrler \underset{\{tmpCtrl,tmpConf\}}{\parallel} GUIRepr(In))$$

$$GUICtrler = tmpCtrl?X\_DATA \rightarrow control!X\_MSG \rightarrow configure?OutMsg \rightarrow getMsg?OutMsg!Out \rightarrow tmpConf!Out \rightarrow logOut!OutMsg \rightarrow recordOut!RcdMsg \rightarrow SKIP$$

$$GUIRepr(In) = display!In \rightarrow interact?X \rightarrow GUICtrl!X \rightarrow SKIP \underset{\{guiCtrl\}}{\parallel} GUICtrl?Y \rightarrow tmpCtrl!Y \rightarrow tmpConf?Z \rightarrow display!Z \rightarrow SKIP$$

$$Log = logOut?LogMsg \rightarrow displayLog.LogMsg \rightarrow Log$$

$$Record = record?RcdMsg \rightarrow putRecord.RcdMsg \rightarrow check \rightarrow Record$$

If users want to implement their own GUI view for event calls, they should follow the CSP specification of the *GUIRepr(In)* process to implement the *createChRepr* method. One notable point in the *GUIRepr(In)* process is that we actually use two parallel processes to specify the GUI view: one to create and display the GUI component, and the other one to handle the interaction. We use a specific process to display the GUI component because a Java *Swing* component cannot be displayed properly if its thread blocks. In this case, if we only use a single process here, it would block on reading from the channel *tmpConf*.

Other process activities, such as parallel composition, external choice, and process calls, are implemented in the GUI package in very similar ways, but using different GUI components for interaction.

5.7.3 Runtime Assertion Checking

Generally, there are two kinds of properties for B+CSP models:

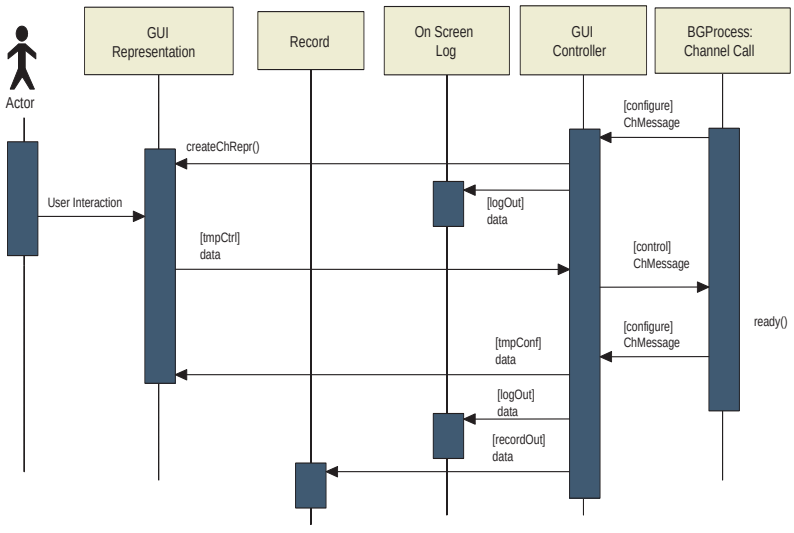

FIGURE 5.24: The GUI communication of event call: level 2

- **Known properties**. These properties, e.g. safety and deadlock, can be checked in the ProB tool for the B+CSP model. The test on the generated Java programs checks whether the verified properties are also preserved in the Java implementation. This provides a partial evidence for the correctness of the Java implementation strategy.

- **Unknown properties**. For other properties, e.g. fairness, which cannot be specified or verified in ProB, we provide alternative experimental methods to evaluate them in the Java programs at runtime. In these circumstances, the generated Java program runs as simulators for B+CSP models. It produces runtime traces, and experimentally verifies the properties on these traces.

ProB provides a mechanism for detecting deadlock in the state space. When the system reaches a state where no further operation can progress, it deadlocks. Stronger liveness properties, such as livelock-freeness and reachability, are difficult to detect in model checking, and are not supported by ProB. Fairness, which involves temporal logic, is an even more complex property for model checking. Many approaches [HOS+93, LMC01, TS99a] have been attempted for extending model checking of B or CSP specifications to temporal logics.

The runtime assertion checking module (RACM) in JCSProB is designed to check user-defined assertions on the JCSProB GUI programs. It receives runtime information from the GUI controller, and keeps a record of the information. Based on the record information, the user can define runtime checks user customized assertions.

All runtime records received by the RACM are stored in a sequential-order record log. Also, each of these records contains runtime information, such as process name, channel name, and variable values. The RACM dynamically loads several assertion objects at runtime. The configuration file tells the RACM which assertion objects to load. For each assertion, users pick up some fields of these information, and compare two records upon the values of these fields. If two records are equal on all these user-defined information fields, they are identical. For each assertion, all the unique records defined on these information fields are the target situations to be discussed. For example, if a unique record has three instance in the record log, that means the situation happened three time in the record log. Assertions can be defined on the occurrence of these situations.

The assertions are defined under the *Assertions* element of the configuration file *configuration.xml*. Here is an example of a frequency assertion:

```
<Assertions>
     <Assertion>
          <AssertionName>Frequency</AssertionName>
          <CheckingClass>Frequency</CheckingClass>
          <RecordType>
```

```
                <ProcName>true</ProcName>
                <ChName>false</ChName>
                <ChInArgs>none</ChInArgs>
                <ChOutArgs>none</ChOutArgs>
                <ProcArgs>1</ProcArgs>
            </RecordType>
            <AssertionBody>
                <Duration>15</Duration>
                <Occurrence>3</Occurrence>
            </AssertionBody>
        </Assertion>
    </Assertions>
```

For an *Assertion* element, the structures of the first three sub-elements, *AssertionName*, *CheckingClass* and *RecordType* are fixed, and the user is only allowed to change the values of all the fields, whereas the last one, *AssertionBody*, can be customized by the user for defining different assertions.

- The *AssertionName* element provides the name of the assertion.
- The *CheckingClass* element gives the name of the Java class, which implements the assertion checking. The implementation class needs to be a subclass of the JCSProB class *Checking*. The runtime checking module loads an object of this class at runtime. In this case, an implementation class, *Frequency*, implements the checking algorithm for this assertion.
- The *RecordType* element defines how the record information be used by this assertion. As most assertions we defined concern fairness properties of the program, a very common work is comparing two records. In *RecordType*, the data used for comparing records are defined. In this example, two records are thought to be identical when their *ProcName* are same, and the first of process variables are equal.
- The *AssertionBody* element is an user defined element. The user can customize the data structure of this element for a certain assertion checking class. Also, the data setting in the assertion data structure can be changed. How to use the assertion information defined here should be defined in the implementation class of this assertion, which is discussed later in this section.

The *JCSProB* package provides an abstract class *Checking*, which is used to define assertions. When defining a new assertion, the user needs to implement two abstract methods, *initAssertion* and *check*, from the *Checking* class. The *initAssertion* method takes the XML element *AssertionBody* from the configuration file, which defines the

assertion, as its argument. Inside the method, the user need to assign the structured data in the element into the form which can be used by the assertion checking. In the *check* method, the user needs to implement the assertion checking algorithm.

In the above frequency example specified in XML, the fairness property it tries to tested is:

$$!(i).(i \in ProcID \ \&\ card(record) > 15 \Rightarrow \\ \mathbf{card}(\mathbf{card}(record) - 15..\mathbf{card}(record) \lhd record \rhd \{i\}) > 3)$$

The symbol !(i) here means "for all i", *card()* is a cardinality operator, and *ran()* returns the range of a function. The symbol $\lhd$ represents domain restriction, while the symbol $\rhd$ represents range restriction. The assertion means that for the last 15 records the checking module received, all the involved processes should at least progress three times. In Appendix B.1, the *Frequency* class is presented. The *check* method of it implements the checking algorithm in Java.

The implementation class of runtime checking has a vector of such *Checking* objects. After received a new record from the GUI program, it calls the *check* method of all these *Checking* objects. The module also maintains a history of all the checking failures. When the program exits, all the records and failures information are saved into a log file.

5.7.4 A Example of the Standard GUI View

The automated translation tool, which is introduced in Section 6.2, can generate necessary files from combined B and CSP models for GUI. The GUI program uses the standard GUI view as default. Figure 5.25 shows the interface of the translator in ProB. If the option 'Generating GUI programs' is selected, the translator would generate *JCSProB* programs with GUI support, as well as the three configuration files for guiding the GUI. Figure 5.26 shows a GUI program directly generated for the *wot-no-chicken* example [Wel98].

- The dark gray panel, marked with number 1, is a thread container which represents a Java thread of the *JCSProB* program. The blue panel in it is an object of the *GUIProc* class, which represents a *JCSProB* process. This structure simulates the thread/process structure of *JCSProB* we introduced in Section 5.5, which means recursive processes run in the same thread container. A process container contains a number of light grey internal panels to represents the elements in the expression of the process. A *GUIProc* object starts with a button marked with the process name for starting the process. In this example, it is the button with the text *XPhil_procclass*.

- The process *Phils_procclass*, marked with number 2, contains a parallel composition of several processes. A yellow panel (not very visible in this figure) contains a number of thread containers for running the process components.

- The buttons with number 3, 4, and 5, demonstrates three states of event representation. The light grey button (3) means the corresponding event has not been called by the GUI, while the dark grey button (5) represents the event which has been called by the GUI, but still has not responded to the GUI. After the event completes, the process end sends a message to the GUI, then the button would turn into orange color (4).

- The text panel marked with number 6 on the figure is the on-screen log module. It displays all the log information sent by the GUI controller. Users can use the configuration files to modify the log information displayed here.

- The table component marked with number 7 display the values of B variables. The values are updated at runtime.

- The table component, which is in a separate frame and marked with number 8, shows the information of runtime records. The *process* column display the records received, which including the process name, the process position from where the record was sent, and the value of process parameters. The *occurrence* column contains two numbers which are separated by a semicolon: the first number is the occurrence of the record, and the second number counts the violation of runtime assertions.

- The button marked with number 9 is a two-state button used for user control. The button is in *automatic* state as default, which means the GUI can run automatically

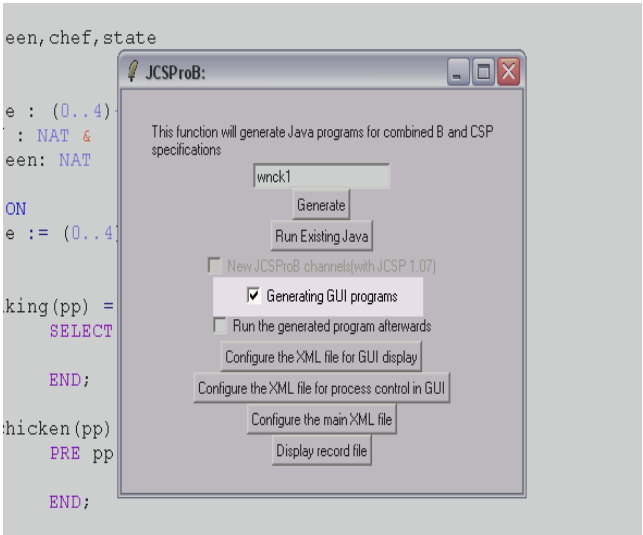

FIGURE 5.25: The interface of translation tool in PROB

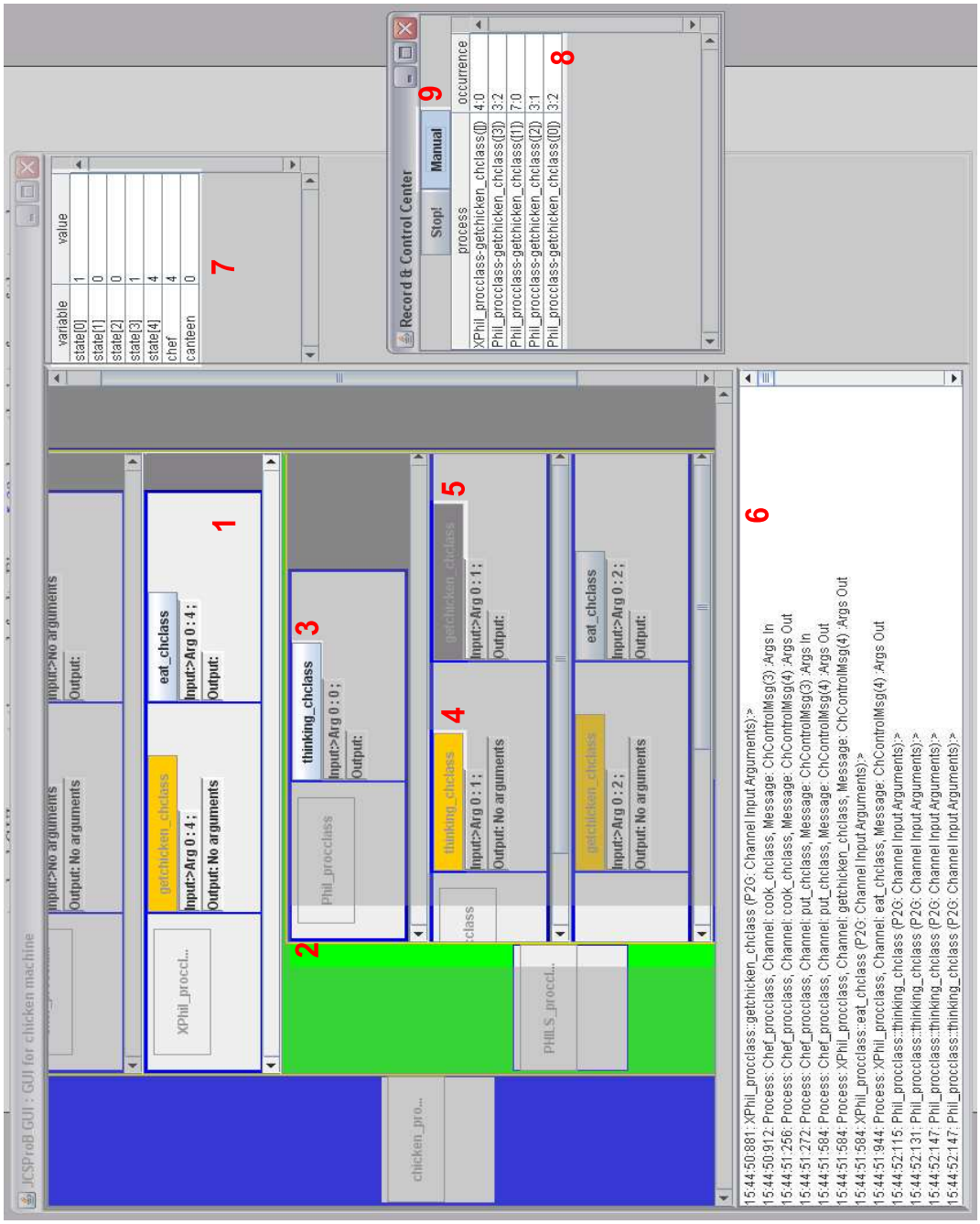

FIGURE 5.26: The interface of translation tool in PROB

without user interaction. When the button is pressed, it changes to *manual* state, in which the user has to manually drive the GUI.

Chapter 6

Translation from B+CSP to Java

The *JCSProB* package provides basic facilities for constructing concurrent Java applications from B+CSP models. However, there is still a big gap between the combined specification and its Java implementation. Manually constructing the Java implementation is still very complex, and cannot guarantee the Java application correctly implements its formal model. To close the gap, a set of translation rules are developed to provide an explicit connection between formal models and the target Java code. The translation rules can be recursively used to generate a concurrent Java application from a B+CSP model. In Section 6.1, we discuss a number of translation rules concerning the translation of CSP processes. An automated translation tool, which implements the translation rules, is also developed as an extension of the ProB tool. It can save effort and avoid careless errors that are very common in writing code. In Section 6.2, the tool is discussed in detail.

In Appendix A, more translation rules are presented.

6.1 Translation Rules

To define the translation rules *Tr*, we first use the BNF (Backus Naur Form) notation to define a subset of the B+CSP specification language that can be implemented in the Java/JCSProB programs. The translation uses language elements in the definition of the two specification languages, and specifies their translation to Java code. In the translation rules:

- The items with fat font in translation rules, e.g. **GuardsList**, are names of translation rules.
- The items with italic font, e.g. *Chs*, are language elements from B+CSP specifications.

- The items with type writer font are terminals, which are Java code. The symbols from Java code are marked with single quotation marks, e.g. `';'`, in order to avoid confusion with BNF symbols.

To understand a translation rule, users need to refer the B+CSP language elements involved, as well as the rule itself. For example, the following translation rule generates a Java class for a B set definition:

Rule 2.1. $\mathbb{B}\mathbb{S}\mathtt{et}\mathbb{C}\mathtt{lass}\lceil BSet \rfloor \Rightarrow$
$\mathbb{P}\mathtt{ackage}\mathbb{D}\mathtt{ef}$
$\mathbb{C}\mathtt{lass}\mathbb{H}\mathtt{eader}$
`public enum` $\mathbb{P}\mathtt{rint}\lceil BSetName \rfloor$ `'{'`
$\mathbb{P}\mathtt{rint}\lceil Iden \rfloor^{+',}$
`'}'`

B+CSP: $BSet \vdash BSetName' = \{' \ Iden^{+',}\ '\}'$

□

The name of the rule, $\mathbb{B}\mathbb{S}\mathtt{et}\mathbb{C}\mathtt{lass}\lceil BSet \rfloor$ is defined first, between the rule number **Rule 2.1**, and the symbol $\Rightarrow$. After $\Rightarrow$, the rule body, which defines how to translate the rule, is given. The rule body can include:

- Names of other rules, such as $\mathbb{C}\mathtt{lass}\mathbb{H}\mathtt{eader}$, which can be further translated using their rule definitions.
- Terminals, such as `public`, `'{'`.
- Extended BNF notations, such as $+$. $\mathbb{P}\mathtt{rint}\lceil Iden \rfloor^{+',}$ here means that the rule can be applied more than once, but each of them needs to be separated by a *','*.

The additional B+CSP language definition after the main rule body, labelled as **B+CSP**, provides information of formal models involved in this translation rule. However, in some cases, the language definition of B+CSP cannot clearly provide enough information for the translation. To make the translation rules more explicit, supplements are introduced to express the information which cannot be easily specified in the BNF definition of B+CSP. For example, in the Java implementation of *external choice*, a Java `switch` statement is used to express all the choice pathes:

Rule 3.6.10.3. $\mathbb{C}\mathtt{hoice}\lceil ProcBPrefix^{*} \rfloor \Rightarrow$
`case` $\mathbb{I}\mathtt{nteger}$ `':'`
$\mathbb{C}\mathtt{hoice}\mathbb{R}\mathtt{tn}\mathbb{E}\mathtt{v}\lceil ProcBPrefix \rfloor$

| $\mathbb{C}\mathtt{hoice}\mathbb{R}\mathtt{tn}\mathbb{E}\mathtt{v}\mathbb{V}\mathtt{alue}\lceil ProcBPrefix \rfloor$
| $\mathbb{P}\mathtt{roc}\mathbb{E}\lceil ProcBPrefix \rfloor$
`break';'`
`'}'`

CSP: $ProcBPrefix \vdash ChCall \rightarrow ProcB$

Supplement: When *ChCall* is a B+CSP event without output data, choose $\mathbb{C}\mathtt{hoice}\mathbb{R}\mathtt{tn}\mathbb{E}\mathtt{v}$

Supplement: When *ChCall* is a B+CSP event with output data, choose $\mathbb{C}\mathtt{hoice}\mathbb{R}\mathtt{tn}\mathbb{E}\mathtt{v}\mathbb{V}\mathtt{alue}$

Supplement: When *ChCall* is a CSP communication channel, choose $\mathbb{P}\mathtt{roc}\mathbb{E}$

Supplement: The rule $\mathbb{I}\mathtt{nteger}$ provides various values for all different repeatable instances of *ProcBPrefix*. It starts with 0, increasing by 1 each time.

□

The integer numbers next to each `case` statement should increase each time when the rule is applied, but it is fairly hard to express this in the BNF definition of B+CSP. With the supplement in this rule, we can explicitly instruct users on how to correctly make use of the translation. Accordingly, the translation rules *Tr* are defined as a conversion from B+CSP specification languages and additional supplements S to Java programs:

$$B + CSP \stackrel{S}{\Longrightarrow}_{Tr} Java/JCSProB$$

The translation rules have two translation modes: without-GUI and with-GUI. In the translation rule definitions, the elements marked with $[NoGUI]$ are used only in without-GUI mode, while those elements marked with $[GUI]$ are used only in with-GUI mode. For example, Rule 3.6.9.6 is designed for translating the channel input of a communication channel:

Rule 3.6.9.5.1. $\mathbb{C}\mathtt{omm}\mathbb{C}\mathtt{hannel}\mathbb{R}\mathtt{ead}\lceil InputCh \rfloor \Rightarrow$
{ $\mathbb{P}\mathtt{roc}\mathbb{V}\mathtt{ar}\mathbb{N}\mathtt{ame}\lceil Var \rfloor$ `'='` $\mathbb{C}\mathtt{h}\mathbb{O}\mathtt{bj}\mathbb{N}\mathtt{ame}\lceil ChName \rfloor$ `'.'read'();'` }$_{[NoGUI]}$
| { $\mathbb{P}\mathtt{roc}\mathbb{V}\mathtt{ar}\mathbb{N}\mathtt{ame}\lceil Var \rfloor$ `'='` `channelRead'('`
$\mathbb{C}\mathtt{h}\mathbb{O}\mathtt{bj}\mathbb{N}\mathtt{ame}\lceil ChName \rfloor$ `',"'` $\mathbb{C}\mathtt{h}\mathbb{O}\mathtt{bj}\mathbb{N}\mathtt{ame}\lceil ChName \rfloor$
`'");'` }$_{[GUI]}$

CSP: $InputCh \vdash ChName\text{'?'}Var$

□

In the without-GUI mode, the first line of this rule marked with *[NoGUI]* is applied. The rest part of this rule, marked with $[GUI]$, is applied in the with-GUI mode. The mode options serves as a static environment setting for the translation. When users start to employ the translation rules for generating Java programs, they must first define the translation mode they want, and then apply it all through a translation run.

The translation rules of without-GUI mode mainly consists three rule sets:

- Rules for generating process classes
- Rules for generating channel classes
- Rules for generating B variable classes, invariants and assertions

The translation rules set with GUI have a set of translation rules for generating configuration files for GUI, as well as three corresponding rules sets of the three ones above.

6.1.1 Translation Rules for Processes

Processes are elementary structures for both *CSP* models and *JCSProB* programs. The translation rules for processes provide the regulations for constructing and composing *JCSProB* processes classes from *CSP* process specifications. In this section, we go over some essential rules for translating process classes.

Rule 3.1 shows the translation rule for producing a process class from a *CSP* process. A named *CSP* process specification *Process* is implemented to a Java class which extends the *BCSProcess* or *BGProcess* class of *JCSProB*.

Rule3.1. $\mathbb{P}\mathrm{rocess}\mathbb{C}\mathrm{lass}\lceil \mathit{Proc} \rfloor \Rightarrow$

$\mathbb{P}\mathrm{ackage}\mathbb{D}\mathrm{ef}$
$\mathbb{C}\mathrm{lass}\mathbb{H}\mathrm{eader}$
`public class` $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{lass}\mathrm{Name}\lceil \mathit{ProcName} \rfloor$ `extends`
$\{$`BCSProcess`$\}_{[NoGUI]} \mid \{$`BGProcess`$\}_{[GUI]}$ '{'
$\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{D}\mathrm{ecl}\lceil \mathit{Chs} \rfloor$
$\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil \mathit{Vars} \rfloor$
$\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{onstructor}$
$\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}\lceil \mathit{ProcBody} \rfloor$
'}'

B+CSP: *Proc* $\vdash$ *ProcHeader* '=' *ProcBody*
B+CSP: *ProcHeader* $\vdash$ *ProcName* {'('VarsList')'}
Supplement: *Chs consists of all the channels/events called by the process, or its subprocess*
Supplement: *Vars consists of all the variables used in the process*

□

A *CSP* process may include a number of variables. In its Java implementation, these process variables are translated into variables of the process class. Rule 3.3 $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}$

is designed for generating declarations for these variables. Also, a process and its sub-processes may call both combined B+CSP events and CSP communication channels. Rule 3.2 ℙrocℂhsDecl (Section 6.1) and Rule 3.2.1 ℙrocℂhDecl is for generating the declaration of channels/evnets.

Rule 3.2. ℙrocℂhsDecl⌈*Chs*⌋ ⇒ ℙrocℂhDecl⌈*Ch*⌋*

B+CSP: *Chs* ⊢ *Ch**

□

Rule 3.2.1. ℙrocℂhDecl⌈*Ch*⌋ ⇒

ℂhannelℙype⌈*Ch*⌋ ℂhOℬjName⌈*Ch*⌋ ';' □

The rule ℂhannelℙype called in *Rule 3.2.1* returns the type information of a channel. The typing here need not to be an implementation channel class. For combined B+CSP event, it returns the abstract class which provides *ready* methods, while for JCSP communication channels, it returns the interfaces of channel input and output. The rule ℂhannelℙype is presented in Section 6.1.3.

A *JCSProB* process class gets references of channel/event objects and input variables from the environment. Therefore, when a process object is constructed, this information needs to be assigned to it. *Rule 3.4* generates the constructor method of a process class. The required information from the environment are assigned to the process in the constructor.

Rule 3.4. ℙrocℂonstructor ⇒

```
ℙrint⌈ProcName⌋ '_'procclass'('
    ℙrocℂhsℙypeList⌈Chs⌋ ','
    {ChannelOutput conf ','}[GUI]
    ℙrocVarsℙypeList⌈Vars⌋')'{'
        {super()';'}[NoGUI] | { super(conf)';' }[GUI]
        ℙrocℂhsAssign⌈Chs⌋
        ℙrocVarsAssign⌈ExtVars⌋
'}'
```

B+CSP: ProcHeader ⊢ *ProcName* {'('VarsList')'}
Supplement: *Chs is all the channels/evnets involved in this process or its sub-processes.*
Supplement: *Vars is all the variables involved in this process.*
Supplement: *ExtVars is all the external variables in VarsList of the ProcHeader definition.*

□

The constructor method obtains the references of channel/event objects, and values of variables from parameters of the method. *Rule 3.4.1* $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{T}\mathrm{ype}\mathbb{L}\mathrm{ist}$ prints out the list of channel/event objects, and *Rule 3.4.2* $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{T}\mathrm{ype}\mathbb{L}\mathrm{ist}$ provides the list for input variables *ExtVars*. Inside the method body, the external object references or values are assigned to their internal counterparts. Here, the rule for translating assignments of channels/events is presented:

Rule 3.4.3. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{A}\mathrm{ssign}\lceil Chs \rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{A}\mathrm{ssign}\lceil Ch \rfloor^*$

Supplement: *Chs is all the channels/evnets involved in this process or its sub-processes.*
Supplement: $Chs \vdash Ch^*$

□

Rule 3.4.3.1. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{A}\mathrm{ssign}\lceil Ch \rfloor \Rightarrow$

`this‘.’`$\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ch \rfloor$ `=` $\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ch \rfloor$ `‘;’` □

Above rules, from *Rule 3.2* to *Rule 3.4*, focus on making all the data and channel/event references ready. The execution sequences of a process, which is expressed in *ProcBody* (refer to Rule 3.1 for the definition), is translated into the *run* method of the process class. *Rule 3.5* $\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}$ is designed for generating the *run* method.

Rule 3.5. $\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}\lceil ProcBody \rfloor \Rightarrow$

`public void run‘(){’`
`{this‘.’creatGUIProc‘();’}`$_{[GUI]}$
`{start‘();’}`$_{[GUI]}$
$\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcBody \rfloor$
`{end‘();’}`$_{[GUI]}$
`‘}’`

□

Rule 3.6 $\mathbb{P}\mathrm{roc}\mathbb{E}$ appeared in the above rule is an important rule for process expressions. It generates all the CSP process expressions supported by the B+CSP specification.

Rule 3.6. $\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcB \rfloor \Rightarrow$
$\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{P}\mathrm{arallel}\lceil ProcBParallel \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{R}\mathrm{epl}\mathbb{P}\mathrm{arallel}\lceil ProcBReplParallel \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{I}\mathrm{nterleave}\lceil ProcBInterleave \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{R}\mathrm{epl}\mathbb{I}\mathrm{nterleave}\lceil ProcBReplInterleave \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{S}\mathrm{equence}\lceil ProcBSequence \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{I}\mathrm{f}\mathbb{T}\mathrm{hen}\lceil ProcBIfThen \rfloor$

$|\ \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{P}\mathrm{refix}\lceil ProcBPrefix\rfloor$
$|\ \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{C}\mathrm{hoice}\lceil ProcBChoice\rfloor$
$|\ \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{R}\mathrm{epl}\mathbb{C}\mathrm{hoice}\lceil ProcBReplChoice\rfloor$
$|\ \mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{C}\mathrm{all}\lceil ProcBCall\rfloor$

B+CSP: $ProcBody \vdash ProcBParallel$
$|\ ProcBReplParallel$
$|\ ProcBInterleave$
$|\ ProcBReplInterleave$
$|\ ProcBSequence$
$|\ ProcBIfThen$
$|\ ProcBPrefix$
$|\ ProcBChoice$
$|\ ProcBReplChoice$
$|\ ProcBCall$

□

When using this rule, we first need to check the BNF definition of *ProcB* to find the pattern of the current *CSP* process expression. For example, if the current *CSP* expression matches *ProcBParallel* in the definition of B+CSP, the translation rule $\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{P}\mathrm{arall}\mathbb{e}\mathbb{l}\lceil ProcBParallel\rfloor$ would be applied here. Many process expression rules are appeared in *Rule 3.6*, but we only continue the discussion of these rules with one particular rule for parallel composition. Other rules can be found in Appendix A.

Rule 3.6.1. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{P}\mathrm{arall}\mathbb{e}\mathbb{l}\lceil ProcBParallel\rfloor \Rightarrow$
$\mathbb{P}\mathrm{ara}\mathbb{C}\mathrm{hs}\mathbb{N}\mathrm{ums}\lceil Chs\rfloor$
{ `new CSParallel'(' `}$_{[NoGUI]}$ | { parallelCtrl'(' }$_{[GUI]}$
{ `new BCSProcess'[]{' `}$_{[NoGUI]}$ | { `new BGProcess'[]{' `}$_{[GUI]}$
$\mathbb{C}\mathrm{all}\mathbb{I}\mathbb{P}\mathrm{roc}\lceil Proc\rfloor^{+',}$
`'}'`
{ `').run'();'' `}$_{[NoGUI]}$ | { `');' `}$_{[GUI]}$

B+CSP: $ProcBParallel \vdash Proc^{+'\|'}$
Supplement: *Chs* includes all the B+CSP events that have multi-way synchronizing
Supplement: *Procs* includes all the process in the parallel composition

□

The translation of parallel composition starts with defining numbers of processes synchronizing on shared events. A shared event needs this number for counting down the synchronization barrier. *Rule 3.6.2* is designed for calling a special method *inc_syn_procs_no* to increase the barrier number.

Rule 3.6.2. $\mathbb{P}\mathrm{ar}\mathbb{C}\mathrm{hs}\mathbb{N}\mathrm{ums}\lceil Chs\rfloor \Rightarrow \mathbb{P}\mathrm{ar}\mathbb{C}\mathrm{h}\mathbb{N}\mathrm{ums}\lceil Ch\rfloor^*$

Supplement: $Chs \vdash Ch^*$

□

Rule 3.6.2.1. $\mathbb{P}\mathrm{ar}\mathbb{C}\mathrm{h}\mathbb{N}\mathrm{ums}\lceil Ch\rfloor \Rightarrow$

$\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ch\rfloor$`‘.’inc‘_’syn‘_’proces‘_’no‘(’` $\mathbb{I}\mathrm{nteger}$ `’);’`

Supplement: Rule $\mathbb{I}\mathrm{nteger}$ here should be the number of processes which synchronize on the event Ch.

□

The other translation rule referred in *Rule 3.6.1* is $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}$, which translates the call on a new process object. As this rule involves many other rules, we do not discuss it here in detail. If interested, please refer to *Rule3.6.4.1* in Appendix A.

Also in *Rule 3.6.1*, the without-GUI and with-GUI modes generate quite different Java code. In the without-GUI mode, a new *CSProcess* object is built upon an array of *BCSProcess* objects, while in the with-GUI mode, the *parallelCtrl* method provided by the *BGProcess* class is called, and an array of *BGProcess* objects is passed to the method as a parameter. However, although the target Java codes are different, the ways in which they apply translation are similar, which means they call same translation rules and use these rules in a similar structure. In this case, the two modes both use rules $\mathbb{P}\mathrm{ar}\mathbb{C}\mathrm{h}\mathbb{N}\mathrm{ums}$ and $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}$, and the ways they apply these rules are identical. Therefore, the differences of the two modes in the translation are just on the syntax level.

6.1.2 Translation Rules for Events

The translation rules for generating event classes obtain most information from the B part of the combined specification. *Rule 4.1* is designed for generating the main structure of an event class.

Rule 4.1. $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\lceil Ev\rfloor \Rightarrow$

$\mathbb{P}\mathrm{ackage}\mathbb{D}\mathrm{ef}$

$\mathbb{C}\mathrm{lass}\mathbb{H}\mathrm{eader}$

`public class` $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil Ev\rfloor$ `extends` $\mathbb{E}\mathrm{vent}\mathbb{T}\mathrm{ype}\lceil Ev\rfloor$ `‘{’`

$\mathbb{M}\mathrm{ach}\mathbb{N}\mathrm{ame}$`‘_’machine var‘;’`

$\mathbb{E}\mathrm{vent}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil BOpVars\rfloor$

$\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{onstructor}\lceil Ev\rfloor$

$\{\mathbb{E}\text{ventInputMethod}\lceil BInVars\rfloor\}$
$\{\mathbb{E}\text{ventOutputMethod}\lceil BOutVars\rfloor\}$
$\mathbb{E}\text{ventRun}\lceil BOpBody\rfloor$
$\{\mathbb{E}\text{ventPrecondition}\lceil BOpBody\rfloor\}$
'}'

B: $Ev \vdash BOutVars \leftarrow EvName'('BInVars')'BOpBody$
Supplement: $BOpVars \vdash BOpInVarsBOpOutVarsy$

□

In *Rule 4.1*, the rule $\mathbb{E}\text{ventVarsDecl}\lceil BOpVars\rfloor$ produces the Java code for declaring event variables. The type information of all the B variables is defined as type invariants inside the **INVARIANTS** clause of a B machine. In *Rule 4.2*, the type of a B variable is obtained from the rule $\mathbb{B}\text{VarType}\lceil BOpVar\rfloor$.

Rule 4.2. $\mathbb{E}\text{ventVarsDecl}\lceil BOpVars\rfloor \Rightarrow$
$\{\mathbb{B}\text{VarType}\lceil BOpVar\rfloor\mathbb{P}\text{rint}\lceil BOpVar\rfloor\}^{+';'}$

B: $BOpVars \vdash BOpVar^{+',}$

□

Also in *Rule 4.1*, the rule $\mathbb{E}\text{ventInputMethod}\lceil BInVars\rfloor$ prints out the Java method *assign_input*, which is used to assign input data to the event variables, and the rule $\mathbb{E}\text{ventOutputMethod}\lceil BOutVars\rfloor$ generates the method *make_output* for outputing data from event variables. The constructor method of an event class is generated by *Rule 4.3* $\mathbb{E}\text{ventConstructor}\lceil Ev\rfloor$. All the three rules are presented in Appendix A.

The *run* method of an event class implements the data substitutions inside the B operation of the combined event. *Rule 4.6* $\mathbb{E}\text{ventRun}\lceil BOpBody\rfloor$ is defined to generate the *run* method.

Rule 4.6. $\mathbb{E}\text{ventRun}\lceil BOpBody\rfloor \Rightarrow$
`protected synchronized void run'(){'`
$\mathbb{B}\text{eforeStateVars}\lceil StateVars\rfloor$
$\mathbb{B}\text{Substitution}\lceil BSub\rfloor$
`var'.'check'();'`
`{ varsPanelsStore'.'getInstance'().'getPanelInstance'('`
`'"'` MachName `'").'refresh'();'` $\}_{[GUI]}$
'}'

B: $BOpBody \vdash BSub$

Supplement: $Vars$ consists of all the state variables

□

The rule $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ars}\lceil StateVars\rfloor$ declares copies of all the state variables of the B machine. These copies represent the system state before performing the data substitutions. In the implementation of the data substitutions, the real state variables can only appear on the left side of the data transitions, whereas the state copies can only appear on the right side. *Rule 4.6.2* $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar}\lceil StateVar\rfloor$ provides different options for different types of state variables, as the Java code for making a copy of an existing object can be different.

Rule 4.6.1. $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ars}\lceil Vars\rfloor \Rightarrow$
$\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar}\lceil Var\rfloor^*$
Supplement: $Vars$ includes all the state variables: $Vars \vdash Var^*$

□

Rule 4.6.2. $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar}\lceil Var\rfloor \Rightarrow$
$\mathbb{B}\mathbb{T}\mathrm{ype}\lceil Var\rfloor\mathbb{P}\mathrm{rint}\lceil Var\rfloor$ ‘=’
$\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar\mathbb{P}\$et}\lceil Var\rfloor$
| $\mathbb{B}\mathrm{efore\$tate\mathbb{V}arArray}\lceil Var\rfloor$
| $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar\mathbb{R}elation}\lceil Var\rfloor$
| $\mathbb{B}\mathrm{efore\$tate\mathbb{V}arAssign\mathbb{O}bj}\lceil Var\rfloor$
‘;’

Supplement: If Var is a B set, $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar\mathbb{P}\$et}$
Supplement: If Var is an array, $\mathbb{B}\mathrm{efore\$tate\mathbb{V}arArray}$
Supplement: If Var is a relation, $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar\mathbb{R}elation}$
Supplement: Otherwise, $\mathbb{B}\mathrm{efore\$tate\mathbb{V}ar\mathbb{O}bj}$

□

The rule $\mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor$ used in *Rule 4.6* generates the Java code for implementing B substitutions. All the supported B substitutions in the translation can be found in this rule. The unsupported substitutions are not included in the translation for different reasons. For example, the **WHILE** loop is not here because it is not supported by ProB, and the **SELECT** is too abstract and non-deterministic to be implemented in concrete programs(Use **IF** or **PRE** instead!).

Rule 5.1. $\mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor \Rightarrow$
$\mathbb{B}\$\mathrm{ubstitution}\mathbb{P}\mathrm{recondition}\lceil BSubPrecdtn\rfloor$
| $\mathbb{B}\$\mathrm{ubstitution}\mathbb{B}\mathrm{egin}\lceil BSubBegin\rfloor$
| $\mathbb{B}\$\mathrm{ubstitution}\mathbb{V}\mathrm{ar}\lceil BSubVar\rfloor$

$$
\begin{array}{l}
\mid \mathbb{B}\mathbb{S}\mathbb{u}\mathbb{b}\mathbb{s}\mathbb{t}\mathbb{i}\mathbb{t}\mathbb{u}\mathbb{t}\mathbb{i}\mathbb{o}\mathbb{n}\mathbb{P}\mathbb{a}\mathbb{r}\mathbb{a}\mathbb{l}\mathbb{l}\mathbb{e}\mathbb{l}\lceil BSubPar \rfloor \\
\mid \mathbb{B}\mathbb{S}\mathbb{u}\mathbb{b}\mathbb{s}\mathbb{t}\mathbb{i}\mathbb{t}\mathbb{u}\mathbb{t}\mathbb{i}\mathbb{o}\mathbb{n}\mathbb{B}\mathbb{e}\mathbb{E}\mathbb{q}\mathbb{u}\mathbb{a}\mathbb{l}\lceil BSubBeq \rfloor \\
\mid \mathbb{B}\mathbb{S}\mathbb{u}\mathbb{b}\mathbb{s}\mathbb{t}\mathbb{i}\mathbb{t}\mathbb{u}\mathbb{t}\mathbb{i}\mathbb{o}\mathbb{n}\mathbb{I}\mathbb{f}\lceil BSubIf \rfloor \\
\mid \mathbb{B}\mathbb{S}\mathbb{u}\mathbb{b}\mathbb{s}\mathbb{t}\mathbb{i}\mathbb{t}\mathbb{u}\mathbb{t}\mathbb{i}\mathbb{o}\mathbb{n}\mathbb{B}\mathbb{e}\mathbb{E}\mathbb{q}\mathbb{u}\mathbb{a}\mathbb{l}\mathbb{F}\mathbb{u}\mathbb{n}\mathbb{c}\lceil BSubBeqFunc \rfloor \\
\mid \mathbb{B}\mathbb{S}\mathbb{u}\mathbb{b}\mathbb{s}\mathbb{t}\mathbb{i}\mathbb{t}\mathbb{u}\mathbb{t}\mathbb{i}\mathbb{o}\mathbb{n}\mathbb{A}\mathbb{n}\mathbb{y}\lceil BSubAny \rfloor
\end{array}
$$

B: $BSub \vdash$

$$
\begin{array}{l}
BSubPrecdtn \\
\mid BSubBegin \\
\mid BSubVar \\
\mid BSubPar \\
\mid BSubBeq \\
\mid BSubIf \\
\mid BSubBeqFunc
\end{array}
$$

□

6.1.3 Translation Rules for Integration

The B and the CSP are connected on combined B+CSP events. The translation rules concerning the combined event usually obtain information from both sides. In *Rule 7.2.1* and *Rule 7.2.2*, the main rule bodies provide all the possible outputs from applying the rules. It is the supplements in these rules that actually instruct users on how to determine the outputs from applying these rules.

Rule 7.2. $\mathbb{C}\mathbb{h}\mathbb{a}\mathbb{n}\mathbb{n}\mathbb{e}\mathbb{l}\mathbb{T}\mathbb{y}\mathbb{p}\mathbb{e}\lceil Ch \rfloor \Rightarrow$

$$\mathbb{E}\mathbb{v}\mathbb{e}\mathbb{n}\mathbb{t}\mathbb{T}\mathbb{y}\mathbb{p}\mathbb{e}\lceil Ev \rfloor \mid \mathbb{C}\mathbb{C}\mathbb{h}\mathbb{T}\mathbb{y}\mathbb{p}\mathbb{e}\lceil CCh \rfloor$$

B+CSP: $Ch \vdash Ev \mid CCh$

□

Rule 7.2.1. $\mathbb{E}\mathbb{v}\mathbb{e}\mathbb{n}\mathbb{t}\mathbb{T}\mathbb{y}\mathbb{p}\mathbb{e}\lceil Ev \rfloor \Rightarrow$

`CChannel` | `InCChannel` | `OutCChannel` | `OutInCChannel`

Supplement: *If the combined event Ev has no data flow between B and CSP, returns* `CChannel`. (**CSP**: *ch* **B**: *op*)

Supplement: *If the combined event Ev only has input data, from the CSP channel to the B operation, returns* `InCChannel`. (**CSP**: $ch`!'InVars$ **B**: $op`('InVars`)'$)

Supplement: *If the combined event Ev only has output data, from the B operation to the CSP channel, returns* `OutCChannel`. (**CSP**: $ch`?'OutVars$ **B**: $OutVars \leftarrow op$)

Supplement: *If combined event Ev has both input data, from CSP to B, and output data, from B to CSP, returns* `OutInCChannel`. (**CSP**: $ch`!'InVars`?OutVars'$ **B**: $OutVars \leftarrow op`('InVars`)'$)

□

Rule 7.2.2. $\mathbb{CCh}\mathbb{T}\mathrm{ype}\lceil CCh \rfloor \Rightarrow$

`ChannelInput` | `ChannelOutput` | $\mathbb{Ch}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil CCh \rfloor$

Supplement: *If the process only read from channel CCh, returns* `ChannelInput`.
Supplement: *If the process only output to channel CCh, returns* `ChannelOutput`.
Supplement: *If the process and its sub-processes both read and write to channel CCh, returns the channel class name using rule* $\mathbb{CCh}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil \mathrm{CCh} \rfloor$.

□

Rule 7.2.2 returns the implementation class for the CSP communication channels; thus it only concern the CSP part. *Rule 7.2.1* is one of the rules which concern both the B and the CSP parts. Choosing an implementation class for the combined B+CSP event is based on the data flow between the B operation and the CSP channel, which is defined in the restricted semantics of B+CSP. The supplements of *Rule 7.2.1* use both the B and the CSP syntax, and combine them together for the translation. They implement a part of the B+CSP semantics in the translation.

6.2 Translation Tool

The automatic translation tool is constructed as an extension of the ProB tool. It is also developed using *SICStus* Prolog, which is the implementation language of ProB. The translation tool has two main functions:

- **Preprocessing**. The translation tool first gets information of a combined B+CSP model, and transforms it into an information structure which is more accessible in the translation.
- **Translation**. The translation tool implements the translation rules, and translates the model information into Java programs.

In ProB, a B+CSP specification is parsed and interpreted into Prolog clauses, which express the combined specification. As the translation tool works in the same environment as ProB, it acquires information on the combined specification from these Prolog clauses, and translates the information into Java programs.

Figure 6.1 illustrates how ProB parses and interprets B and *CSP* specifications, and where the translator lies in ProB. The *B* part and the *CSP* part of the combined

specification are stored in separate files. The ProB tool first carries out syntax analysis on the two specifications. A *CSP* parser parses the CSP specification into a parsed tree structure, while the *JBTools* package [Bru01] is employed to translate B notations into XML format. After that, the *CSP* interpreter, CIA (CSP Interpreter and Animator) [Leu01], converts the parsed trees of CSP processes into a Prolog fact *agent/2*. The first parameter of *agent/2* provides information of the process name. The second parameter represents the process definition. On the other hand, the Pillow package [CH01] is used to convert the XML representation of B specifications into a series of *Prolog* clauses. Pillow is an external Prolog program, which provides certain methods to access XML files by translating XML into Prolog clauses. Finally, the ProB interpreter takes Prolog clauses of the two specifications and produces new Prolog clauses which expresses the semantics of the combined *B+CSP* model.

The translation tool obtains semantic information from Prolog clauses of the combined model. It also builds some new Prolog facts, which are transformed from clauses of the *B+CSP* specification. For example, in ProB, the Prolog rule *b_get_machine_variables/1* in the *bmachine* module of ProB can return names of all the B variables. However, the data type information of these variables are mixed in the invariant, which can be obtained from calling the *b_get_invariant_from_machine/1* rule from the *bmachine* module. Also, the initialization of these variables is in the **INITIALISATION** statement of a

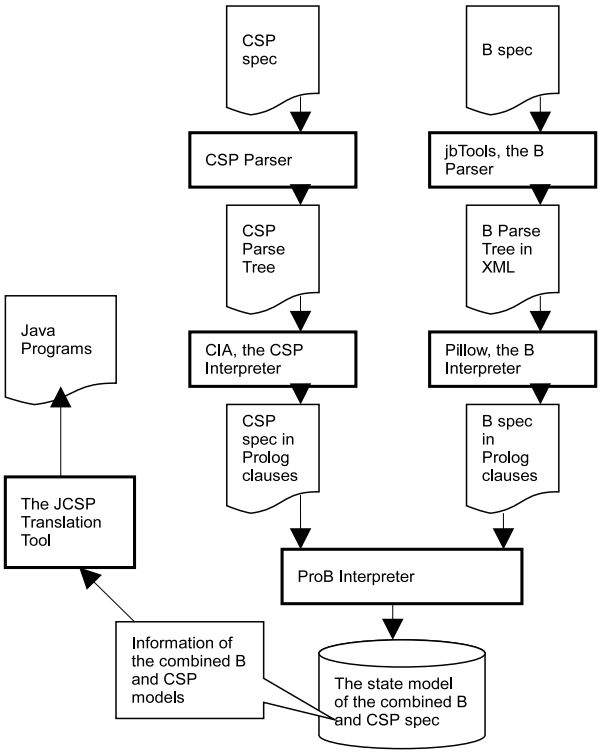

FIGURE 6.1: The parsing and interpretation in PROB

B machine, which can be obtained from calling the *b_get_initialisation_from_machine/1* rule. However, the invariant and initialization information obtained from calling these rules are in complex tree structures which are not very straightforward for the translation. Therefore, we build two new Prolog facts from these rules.

- ***jcsp_b_vars_types/2***. This fact contains two arguments: the variable name, and its typing.
- ***jcsp_b_vars_inits/2***. This fact contains two arguments: the variable name, and its initialization values.

The preprocessed information are more accessible and easier to use in the translation. In some cases, the preprocessing step even implements some fundamental translation rules. In the above example, the data type information stored in *jcsp_b_vars_types/2* is actually the corresponding Java typing of variables, which means the translation rule $\mathbb{T}\mathrm{ype}$ is partially implemented in the preprocessing step.

When the information of combined models is ready, the translation tool starts to generate Java programs according to the translation rules. As we discussed in Section 6.1, the translation rules are expressed in BNF-like production rules. The Prolog language, which is a declarative logic language and is based on mathematic logic and inference, is very appropriate for expressing such a kind of logic rule. Especially, the Prolog rules, which are also inference rules, can implement the translation rules in a straightforward way. In the development of the translation tool, there are correlations defined between translation rules and Prolog rules. For example, in Section 6.1.1, we gave the (Rule 3.5) $\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}$ for translating the *run* method of process classes.

Rule 3.5. $\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}\lceil ProcBody \rfloor \Rightarrow$

```
public void run‘(){’
     {this‘.’creatGUIProc‘();’}[GUI]
     {start‘();’}[GUI]
     ℙroc𝔼⌈ProcBody⌋
     {end‘();’}[GUI]
’}’
```

□

In the translation tool, there is a corresponding Prolog rule for implementing this rule:

```
print_jcsp_process_class_run(Proc,ProcBody) :-
          jcsp_print(1,'public void run(){'),nl,
          (withGUI ->
               (
```

```
                jcsp_print(2,'this.createGUIProc();'),nl,
                jcsp_print(2,'start();'),nl,
                print_jcsp_process_run_proce(Proc,ProcBody,2),
                jcsp_print(2,'end();'),nl
            );
            print_jcsp_process_run_proce(Proc,ProcBody,2)
    ),
    jcsp_print(1,'}'),nl.
```

The correlation of the two rules are quite obvious. The Prolog rule `jcsp_print/2` is used for printing out a string with certain indention. For example, a query of the rule

```
jcsp_print(2,'start();')
```

would print out the string `'start();'` with the indention of two tabs. The special fact `withGUI` contains the information of whether the translation is in the with-gui mode. Also, readers may quickly start to presume that the Prolog rule print_jcsp_process_run_proce/2 is the implementation of *Rule 3.6* $\mathbb{P}\mathtt{roc}\mathbb{E}$. For most translation rules, such corresponding Prolog rules can be directly recognized. However, for some rules which are partially implemented in the preprocessing step of the translation tool, the correlation may be not so easy to identify.

Accordingly, as there is tight correspondence between the translation rules and the Prolog rules of the translator program, the correctness of the translation tool heavily depends on the correctness of the translation rules. For each translation rule, if we can make sure the corresponding Prolog rules produce the same output as the translation rule, then we can say the translation rules correctly implements the translation rule.

6.3 Translation of External Choice

In this section, we use the translation of external choice to demonstrate the translation procedure. The translation rule of external choice, as well as the Prolog rules, are discussed. We also use a toy example of external choice to test the translation, and the generated Java code is presented.

A simple process *Breakfast* has a data *Egg* as its process parameters. The process needs to make choice on two events *fry* and *boil*. The *CSP* specification of this toy example is:

$$Breakfast(Egg) = fry.Egg \rightarrow Breakfast(Egg) \Box boil.Egg \rightarrow Breakfast(Egg)$$

In Section 5.4, we discussed the implementation of external choice for the B+CSP event. The ready state of a combined event is resolved not only based on the processes that

synchronize on the event, but also depends on the precondition of the event. Therefore, the value of input parameters are also required for resolving the choice. *Rule 3.6.10* represents the main translation rule for external choice.

Rule 3.6.10. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{C}\mathrm{hoice}\lceil ProcBChoice \rfloor \Rightarrow$

```
BCSPGuard'[]' in '={' GuardsList⌈ProcBPrefix*⌋ '};'
Vector'<'Vector'<'Object'>>' choiceVec
        = new Vector'<'Vector'<'Object'>>();'
ChoiceValueAssign⌈ProcBPrefix*⌋
{ Alter alt '=' new Alter('in',' choiceVec');'
switch'('alt.select'(){'
      Choices⌈ProcBPrefix*⌋
'}' }[NoGUI]
| { switch'('choiceCtrl'('in',' choiceVec')){'
      Choices⌈ProcBPrefix*⌋
'}' }[GUI]
```

B+CSP: *ProcBChoice* $\vdash$ *ProcBPrefix* {'[]' *ProcBPrefix*}*

□

The translation rule first uses the rule GuardsList to construct an array of guards `in`, which includes the first events of all the possible choice pathes. After that, it introduced a Java *Vector* object `choiceVec` for keeping the values of all the input arguments of these events. In the without-GUI mode, the two objects are used to construct an `Alter` object, whose `select` method actually returns decision of the external choice. In the with-GUI mode, the two objects are passed to the `choiceCtrl` method of the *BGProcess* class for choice decision making. The definitions of the sub-rules, e.g. Choices, can be found in Appendix A.

Based on the translation rules of external choice, a number of Prolog rules are implemented:

```
print_jcsp_process_run_proce(Proc,choice(TransA,TransB),N) :-
      jcsp_print(N,'BCSPGuard[] in = '),
      retractall(jcspcommasymbol),
      print_jcsp_process_run_proce_choice_guard_list(TransA,TransB),
      jcsp_print(N,'Vector<Vector<Object>> choiceVec
              = new Vector<Vector<Object>>();'),nl,
      print_choice_value_assign_trans(TransA,N),
      print_choice_value_assign_trans(TransB,N),
      print(';'),nl,
```

```
        (withGUI ->
              print_jcsp_process_run_proce_choice_with_gui(TransA,TransB,Proc,N);
              print_jcsp_process_run_proce_choice(TransA,TransB,Proc,N)
        ).
print_jcsp_process_run_proce_choice_with_gui(TransA,TransB,Proc,N) :-
        jcsp_print(N,'switch(choiceCtrl(in, choiceVec)){'),nl,
        print_jcsp_process_run_proce_choice_with_gui_trans(TransA,Proc,N+1,0),
        print_jcsp_process_run_proce_choice_with_gui_trans(TransB,Proc,N+1,1),
        jcsp_print(N,'}'),nl.
print_jcsp_process_run_proce_choice(TransA,TransB,Proc,N) :-
        jcsp_print(N,'Alter alt = new Alter(in,choiceVec);'),nl,
        jcsp_print(N,'switch(alt.select()){'),nl,
        print_jcsp_process_run_proce_choice_trans(TransA,Proc,N+1,0),
        print_jcsp_process_run_proce_choice_trans(TransB,Proc,N+1,1),
        jcsp_print(N,'}'),nl.
```

In ProB, the choice pathes are parsed and stored in a tree structure. For example, a external choice on three choice pathes

$$ProcA \Box ProcB \Box ProC$$

would be expressed as:

choice(ProcA,choice(ProcB,ProcC))

Therefore, the *Breakfast* process we specified above would be transformed by the CSP interpreter into the following form:

```
choice(
        prefix([out(term(Egg))],fry,agent_call(a_Breakfast(Egg))),
        prefix([out(term(Egg))],boil,agent_call(a_Breakfast(Egg)))
).
```

The rule head of `print_jcsp_process_run_proce` can capture this tree structure with its second argument `choice(TransA,TransB)`. Also, the following rules, which make use of `TransA` and `TransB`, recursively explore the tree structure to access all the chioce pathes.

- The Prolog rule `print_jcsp_process_run_proce_choice_guard_list` implements the translation rule GuardList. It explores all the choice paths, and prints out an event list including the first event of all the choice paths.

- The rule `print_choice_value_assign_trans`, which implements the translation rules $\mathbb{C}\mathrm{hoice}\mathbb{V}\mathrm{alue}\mathbb{A}\mathrm{ssign}$, explores the tree structure of all the choices, and prints out the Java code for add the values of the first events of all the choice pathes.
- Similarly, the rule `print_jcsp_process_run_proce_choice_with_gui_trans` and `print_jcsp_process_run_proce_choice_trans` also explore the choices tree, and generate all choice pathes in a Java `switch` statement. These two Prolog rules implement *Rule 3.6.10.3* $\mathbb{C}\mathrm{hoices}$, which is presented in Section 6.1

Giving the parsed tree of the *Breakfast* process in without-GUI mode, the above Prolog rules would generate the following Jave code:

```
PCChannel[] in = {fry_ch,boil_ch};
Vector<Vector<Object>> choiceVec = new Vector<Vector<Object>>();
choiceVec.addElement(inputVec(new Object[]{proc_index_a}));
choiceVec.addElement(inputVec(new Object[]{proc_index_a}));
Alter alt = new Alter(in,choiceVec);
switch(alt.select()){
    case 0 :  callNextProc(new Breakfast_procclass(fry_ch,boil_ch,proc_index_a));
              break;
    case 1 :  callNextProc(new Breakfast_procclass(fry_ch,boil_ch,proc_index_a));
              break;
}
```

In the GUI mode, the translator generates a slightly different Java code:

```
PCChannel[] in = {fry_ch,boil_ch};
Vector<Vector<Object>> choiceVec = new Vector<Vector<Object>>();
choiceVec.addElement(inputVec(new Object[]{proc_index_a}));
choiceVec.addElement(inputVec(new Object[]{proc_index_a}));
switch(choiceCtrl(in,choiceVec)){
    case 0 :  choiceRtn(fry_ch);
              nextProcRtn(new Breakfast_procclass(fry_ch,boil_ch,proc_index_a));
              break;
    case 1 :  choiceRtn(boil_ch)
              nextProcRtn(new Breakfast_procclass(fry_ch,boil_ch,proc_index_a));
              break;
}
```

Chapter 7

Experimentations

In this section, experimental evaluation of the whole development strategy from B+CSP to Java is discussed. We first test the expressivity of the restricted B+CSP semantics, and syntax coverage of the translation by applying different syntactic structures to construct various formal models. The generated Java programs are tested, and compared with their formal models.

An important experimentation is to evaluate different kind of properties on the Java programs generated from B+CSP models. The *known properties*, which can be verified in ProB, are also evaluated in the generated Java programs at runtime. Then the runtime checking result is compared with that from the ProB model checking on the B+CSP model. For the properties which are not so easy to express in B+CSP or cannot be verified in ProB, we specify them as runtime assertions, and design runtime assertion checking for them. The example in Section 7.1 shows how the invariant check works for the JCSProB package. In Section 7.2, we construct several models from *Wot-no-chicken* example to demonstrate the user-defined assertion checking.

Another target for the experimentations is to evaluate the scalability issue of this work. One import aspect for scalability here is on the composition/decomposition of the B+CSP model and its Java implementation. Currently, the composition rules for the combined B+CSP model has not been developed. The compositional examples in Section 7.3 practically shows a possible composition style for B+CSP models. There are no formal rules to back this composition attempt.

7.1 Invariant Check: Simple Lift Example

Figure 7.1 presents an abstract lift model. We use this simple example to demonstrate the implementation of invariant check for JCSProB programs.

MACHINE *lift*
VARIABLES *level*
INVARIANT *level : NAT & level ≥ 0 & level ≤ 10*
INITIALIZATION *level := 1*
OPERATIONS
inc = ***BEGIN*** *level := level + 1* ***END****;*
dec = ***BEGIN*** *level := level - 1* ***END***
END
———————
MAIN = inc ⟶ inv_check ⟶ MAIN [] dec ⟶ inv_check ⟶ MAIN ;;

FIGURE 7.1: Combined Specification of lift

Invariants in a B machine demonstrate safety properties of the system model. In the PROB model checking, B invariants are checked on all the states of the system. The violation of the invariants indicates an unsafe state of the system model. The runtime invariant check in the target Java programs can be used to demonstrate the generated Java program preserves the safety properties, which are specified in the invariant of its formal model, in the current trace.

The invariants supported by the translation are mainly from the B0 language conditions. The B0 conditions can be easily translated into Java boolean conditions. The abstract *JcspVar* class, which maintains the system states of B machines, needs to be implemented in a JCSProB program. The *check* method of *JcspVar* need to be implemented for the invariant checking. The Java boolean conditions generated from the B invariants are tested in the *check* method. The *check* method is called each time when an event object performed data changes on state variables. The success of this checking demonstrates the invariants are also preserved in the Java program. When we test a Java program generated from verified B+CSP model, failures of this checking also means the translation does not preserve the semantics of B+CSP.

The unguarded B operations *inc* and *dec* can freely increase or decrease the B variable *floor*. That would easily break the invariant on *floor* (*level ≥ 0 & level ≤ 10*). In the PROB model checking, the violated state can be quickly identified from the state model.

Runtime results of the target Java application demonstrate that the check mechanism can find violation of invariant conditions, and terminate the system accordingly. Therefore, we correct the model to that of Figure 7.2, by adding preconditions.

MACHINE *lift*
VARIABLES *level*
INVARIANT *level : NAT & level ≥ 0 & level ≤ 10*
INITIALISATION *level := 1*
OPERATIONS
inc = ***PRE*** *level < 10* ***THEN*** *level := level + 1* ***END****;*
dec = ***PRE*** *level > 0* ***THEN*** *level := level - 1* ***END***
END

FIGURE 7.2: An example of B machines: lift

The Java programs generated from the modified specification find no violation of invariants.

7.2 Wot-no-chickens: Fairness Assertions

The *Wot,no chickens?* example [Wel98] was originally constructed for emphasizing possible fairness issues in the wait-notify mechanism of Java concurrent programming. There are five philosophers and one chef in this story. The chef repeatedly cooks four chickens each time, puts the chicken in a canteen, and notifies the waiting philosophers. On the other hand, philosophers, but not the greedy one, cyclically continue the following behaviours: think, go to canteen for chickens, get a chicken, and go back to think again. The greedy philosopher does not think, and goes to the canteen directly and finds it devoid of chickens. The Java implementation in [Wel98] employs the Java *wait-notify* mechanism (using the pattern advised by the Java API documentation) to block the philosopher object when there are no chickens left in the canteen. The chef claims the canteen monitor lock (on which the greedy philosopher is waiting), takes some time to set out the freshly cooked chickens and, then, notifies all (any) who are waiting. During this claim period, the diligent philosophers finish their thoughts, try to claim the monitor lock and get in line. If that happens before the greedy philosopher is notified, he finds himself behind all his colleagues again. By the time he claims the monitor (i.e. reaches the canteen), the shelves are bare and back he goes to waiting! The greedy philosopher never gets any chicken.

7.2.1 The Two Models

To test the syntax coverage of the JCSProB package and the translation, several formal models of this example are specified. We use various synchronization strategies and recursion patterns to explore the syntax coverage and the semantics expressivity of the B+CSP specification in the JCSProB package, as well as in the translation. Furthermore, we also want to compare fairness properties of different formal models, in order to evaluate the behaviour of the generated Java programs in practice.

The first combined B+CSP model of this example is presented in Figure 7.3. The CSP part of the specification in Figure 7.4 features some interleaving processes. The atomic access control on the B global variables, and the precondition on the *get_chicken* channel actually require synchronization mechanisms to preserve the consistency of the concurrent Java programs. As all the features concerning the concurrency model are implemented in the JCSProB package, users can work with the high-level concurrency model without noticing the low-level implementation of synchronization.

```
MACHINE chicken
VARIABLES
    canteen,chef,state
INVARIANT
    canteen ∈ NAT & chef ∈ NAT & state ∈ (0..4)→NAT
INITIALISATION
    canteen := 0 || chef := 0 || state := (0..4) * {1}
OPERATIONS
    thinking(pp) =
        SELECT pp:0..4 THEN
        state(pp) := state(pp) - 1
        END;
    getchicken(pp) =
        PRE pp:0..4 & canteen > 0 THEN
        canteen := canteen - 1
        END;
    eat(pp) =
        SELECT pp:0..4 THEN
        state(pp) := state(pp) + 1
        END;
    cook =
        BEGIN
        chef := chef + 4
        END;
    put =
        BEGIN canteen := canteen + 4 || chef := chef - 4 END
END
```

FIGURE 7.3: The B machine of the Wot-no-chicken example

The reader may find that the two machine variables *state* and *chef* are not interesting in this specification. They seems just making some state changes without much meaning. We keep them in the machine in order to use them in later Section 7.3 to demonstrate a decomposition approach. In Appendix B.3, we presents a part of Java source codes for this example.

```
MAIN = Chef ||| XPhil ||| PHILS ;;
PHILS = |||X:0,1,2,3@Phil(X);;
Phil(X) = thinking.X → getchicken.X → eat.X → Phil(X);;
XPhil = getchicken.4 → eat.4 → XPhil;;
Chef = cook → put → Chef ;;
```

FIGURE 7.4: The CSP spec of the Wot-no-chicken example: Model 1

An alternative model is specified in Figure 7.5. As the B machine is the same as the first one in Figure 7.3, only the CSP specification is given here. This model explicitly uses a multi-way synchronization on the *put* channel to force all the philosophers and the chef to synchronize.

$$
\begin{aligned}
&MAIN = Chef\ /|\{put\}|/\ PhilA(4)\ /|\{put\}|/\ PHILS\ ;;\\
&PHILS = [|\{put\}|]X{:}\{0,1,2,3\}@Phil(X)\ ;;\\
&Phil(X) = thinking.X \rightarrow PhilA(X)\ ;;\\
&PhilA(X) = put \rightarrow PhilA(X)\ []\ getchicken.X \rightarrow eat.X \rightarrow\\
&\qquad\quad \textbf{\textit{if}}(X == 4)\\
&\qquad\quad \textbf{\textit{then}}\ PhilA(4)\\
&\qquad\quad \textbf{\textit{else}}\ Phil(X)\\
&\qquad\quad ;;\\
&Chef = cook \rightarrow put \rightarrow Chef\ ;;
\end{aligned}
$$

FIGURE 7.5: Formal specification of Wot-no-chicken example, Model 2

7.2.1.1 Assertion Check and Results

The experimental evaluation test is based on the two models specified above. In the first part of the evaluation, we test the safety and deadlock-freeness properties on the two channels. In Table 7.1, the test results on these properties are demonstrated. The Timing column indicates how many different timing configurations are tested with the model, and the Steps column shows the lengths of the runtime records we collected. As the concurrent Java applications constructed with the JCSProB package need to preserve the same safety and deadlock-freeness properties as their formal models, it partially demonstrates the correctness of the JCSProB package, as well as the translation tool.

Model Name	Property	Processes	Timing	Steps	Result
Model 1	Safety/Invariant	-	15	1000	√
Model 1	Deadlock-freeness	-	15	1000	√
Model 2	Safety/Invariant	-	15	1000	√
Model 2	Deadlock-freeness	-	15	1000	√

TABLE 7.1: The experimental result: Safety and Deadlock-freeness

To test the bounded fairness properties on the target Java programs at runtime, we first need to generate various traces from the concurrent Java programs. In the configuration file of the GUI program, we can define various timing configurations for generating traces for the program. The GUI can force the process to sleep for a fixed time period. In this way, we can explicitly animate formal models with specific timing settings for experimental purposes. Then we employ the fairness assertions check on Java programs embedded with timing settings. The target of this experiment is to practically animate the Java/JCSProB applications, and evaluate their runtime performances with the bounded fairness properties.

In Table 7.2, we show the experimental results of the two models with bounded fairness properties. Frequency 1, 2, and 3 are three different frequency assertion settings (duration and occurrence, Section 5.7.3). For each property, we use five different timing

settings; and for each timing setting, the Java program is tested in five runs. In the result column of the table, *18P7F* means in 25 runs, the check passes 18 times and fails 7 times.

Model Name	Property	Processes	Timing	Steps	Result
Model 1	Frequency 1	All	5	150	4P21F
Model 1'	Frequency 2	Phils+XPhil	5	150	1P24F
Model 1"	Frequency 3	Phils	5	150	23P2F
Model 2	Frequency 1	All	5	150	5P20F
Model 2'	Frequency 2	Phils+XPhil	5	150	0P25F
Model 2"	Frequency 3	Phils	5	150	24P1F

TABLE 7.2: The experimental result: Bounded Fairness Properties

The assertions check also concerns different process groups. In the tests on *Model 1* and *Model 2*, both the philosophers and the chef processes are recorded for assertions check. In model *Model 1'* and *Model 2'*, only the philosopher processes are checked. In *Model 1"* and *Model 2"*, the greedy philosopher is removed and only normal philosopher processes are tested. The testing shows that generated Java programs provide useful simulations for their formal model. It is used to explore and discover the behaviour properties which cannot be verified in ProB model checking.

7.3 Composition of JCSProB Programs

In section, we use two examples to demonstrate a practical composition attempt for the combined B+CSP model, and the JCSProB programs generated from them. In [BL05], only one B and CSP specification pair is allowed, which also applies to the ProB model checker, which means only one such specification pair can be model checked in ProB. However, as the B+CSP semantics also allow CSP communication channels, it is possible and reasonable to connect two specification pair through communication channels. Similar technique [STE05] has been applied in the CSP||B approach for composing different specification pairs.

7.3.1 Composition: Odd-Even example

In [STE05], an Odd-Even example is presented. In this example, two B and CSP specification pairs are connected through their CSP controllers. The *Odd* machine only receives and keeps odd number, whereas the *Even* machine only keeps even number. The controllers of the two machines communicate with each other for the values. For convenience, Figure 7.6 reproduces Figure 2.2 here to specify this example.

The two B+CSP specification pairs are connected through two CSP communication channels, *oddpass* and *evenpass*. The CSP part of the *Even* model first receives a data *Z* from the the *Odd* model through the *oddpass* channel. As the value of the data from the *Odd* is an odd number, the CSP increases the number by 1 and uses the increased number as an input data to call the combined event *evenput*. The B machine *Even* has a state variable *even* for the even number. It provides a corresponding B operation *evenput* which gets the data from its corresponding CSP channel, and also has an *evenget* operation for outputing the data value of *even* to CSP. After the CSP controller receives the data *W* from the combined *evenget* event, it sends the data out through the *evenpass* channel. The function of the *Odd* model is very similar to *Even*, except the B machine *Odd* keeps the state of odd numbers. Figure 7.7 graphically illustrates the communication in this example.

In ProB, we managed to model check the two specification pairs separately. With our JCSProB translator, we can generate two separate Java programs from the models. However, one B+CSP pair in this example only contains one end of a communication channel, while the other end of the channel is in the other specification pair. The model would wait for the channel input or output from the other end. The ProB tool can use its enumeration mechanism to produce the response for the channel call. In JCSProB, we implement special channel-end classes to display channel output to the user, and allow the user to provide data for channel input. Appendix B.4.1 presents the main Java class of the *Even* model. In the class, the *oddpass* channel is declared as the *Ext2OneGUIChannel* class, which is one of the channel-end classes from JCSProB.

The CSP part of the *Even* model requires a channel input from the *oddpass* channel. As the Java program is generated from the *Even* model, it does not have the output-end of this channel which is in the *Odd* model. The translator can detect the absence of the output-end, and employs a message box from the channel-end class to allow the user to

```
MACHINE Even

VARIABLES even

INVARIANT even : NAT & even mod 2 = 0

INITIALISATION even := 0

OPERATIONS
        evenput(nn) =  PRE nn : NAT & nn mod 2 = 0
                       THEN even := nn
                       END;

        nn <-- evenget = BEGIN nn := even END

END
```

```
MACHINE Odd

VARIABLES odd

INVARIANT odd : NAT & odd mod 2 = 1

INITIALISATION odd := 1

OPERATIONS
        oddput(nn) = PRE nn : NAT & nn mod 2 = 1
                     THEN odd := nn
                     END;

        nn <-- oddget = BEGIN nn := odd END

END
```

```
EvenCtrl = oddpass?Z -> evenput!(Z+1) ->
        evenget?W -> evenpass!W -> EvenCtrl ;;
```

```
OddCtrl = oddget?X -> oddpass!X ->
        evenpass?Y -> oddput!(Y+1) -> OddCtrl ;;
```

FIGURE 7.6: The specification of the Odd-Even example

input data for the channel call. Figure 7.8 shows the screenshot of data input for the *oddpass* channel in the *Even* model.

The Java program generated from the *Even* model lacks the output-end of *oddpass*, and the input-end of *evenpass*, whereas the *Odd* model lacks the input-end of *oddpass*, and the output-end of *evenpass*. If we can connect the two programs together using the *oddpass* and *evenpass* channel, it would create a program which implements the whole system model. It is actually very ease to do that. We just need to put the main classes of the two programs together, and replace the four channel-end classes in the two programs with two JCSP communication channels. Appendix B.4.2 shows the main Java class of the combined JCSProB program. In the *Oddeven_run* class, the *evenpass* and *oddpass*

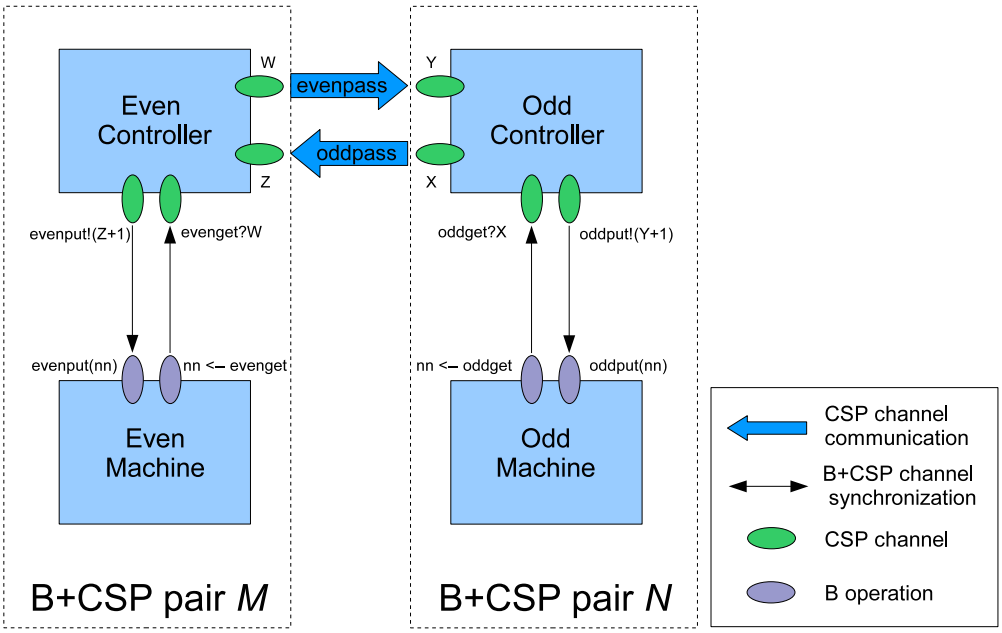

FIGURE 7.7: The communication in the Odd-Even example

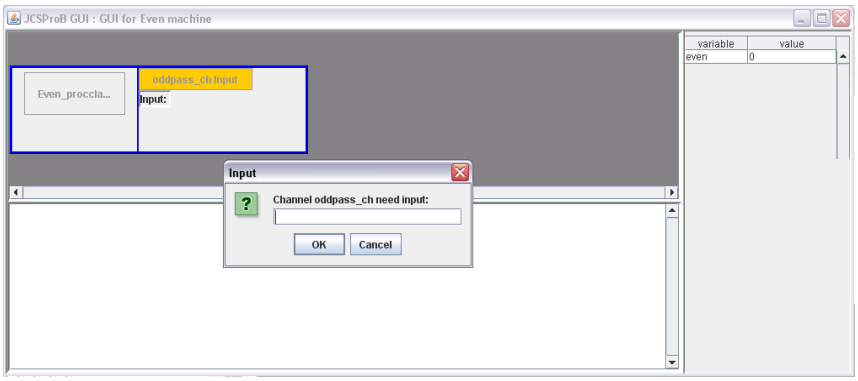

FIGURE 7.8: Data input for communication channel

channels are declared as JCSP communication channels, which are passed to the *Odd* and *Even* models as arguments.

Figure 7.9 shows a screenshot of the combined Java program. Each of the two models runs in different GUI frames, and maintains its own states. They communicate with each other through the two communication channels. Both automatic execution and manual animation of this program practically demonstrated that the composed program correctly implements the $CSP \parallel B$ model specified in Figure 7.6.

7.3.2 Decomposition: Wot-no-chicken

Both *wot-no-chicken* models showed in Section 7.2 contain multiple processes in their CSP specification. In this section, we practically decomposed the B+CSP model in Figure 7.3 and Figure 7.4 into several distributed modules. Each of this decomposed module is a combined B and CSP specification pair. All these modules are connected together through CSP communication channels. In Section 8.2.3, we give a discussion on this decomposition effort.

The combined model defined in Figure 7.3 and Figure 7.4 contain multiple processes, and just one B machine for maintaining the system state. Large scale complex systems are usually distributed system which consists of multiple locations in a network environment. Each of these locations has its own physical memory and thus maintains its own memory state. To model such a system, it is ideal if the system model can have multiple modules. Each of these module should maintain its own state, and can be deployed on an independent location in the distributed system.

It is also not safe to keep all the data variables in a single state machine or memory, because it is not easy to restrict data access control from various processes. In this example, the B variable *state* consists an array of the data for indicating a kind of status of all the philosophers. However, because all the data variables are globally visible to all the processes, it is possible for a process to directly access, or even update the status variables of other processes. Considering both scalability and data safe issues, it should be better to decompose such a model into a distributed model with multiple modules, where each module maintains its own state or physical/virtual memory, and communicates with each other through communication channels.

First Step: Refining the CSP

The CSP specification in Figure 7.4 does not explicitly express the canteen as an entity. In the implementation, we expect a canteen module which maintains the B state variable *canteen* and runs an independent module. Therefore, for the first step, we refines the

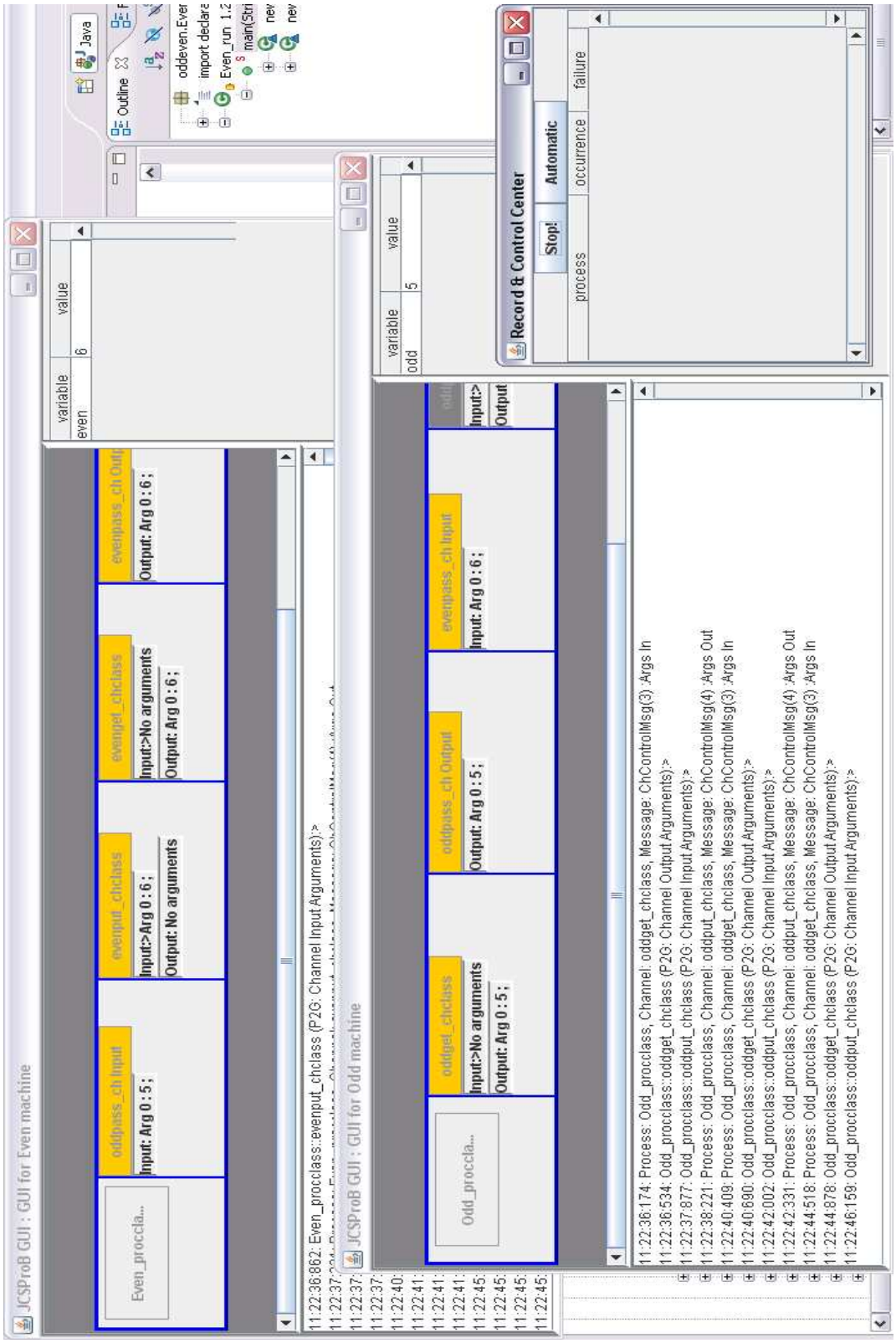

FIGURE 7.9: The GUI program of the combined Odd-Even model

CSP specification by introducing the *Canteen* process. Figure 7.10 presents the new CSP specification with the *Canteen* process.

$$
\begin{aligned}
&MAIN = Chef \ [|\ \{put\}\ |]\ Canteen\ [|\ \{getchicken\}\ |]\ XPHIL\ [|\ \{getchicken\}\ |]\\
&PHILS\ ;;\\
&PHILS = |||X{:}0,1,2,3@Phil(X);;\\
&Phil(X) = thinking.X \rightarrow getchicken.X \rightarrow eat.X \rightarrow Phil(X);;\\
&XPhil = getchicken.4 \rightarrow eat.4 \rightarrow XPhil;;\\
&Chef = cook \rightarrow put \rightarrow Chef\ ;;\\
&Canteen = CProc\ |||\ PProc\ ;;\\
&CProc = put \rightarrow CProc\ ;;\\
&PProc = getchicken.X \rightarrow PProc\ ;;
\end{aligned}
$$

FIGURE 7.10: The Wot-no-chicken example: introducing the *Canteen* process

Second Step: Decomposing Chef and Canteen

To decompose the B machine specified in Figure 7.3, we need to justify which the relation between the B state variables and the CSP processes regarding the events used in each process. In this example, the model is decomposed into three modules: *Chef*, *Canteen* and *Phils*. Figure 7.11 illustrates the connections between three decomposed modules. The specification of this decomposed model can be find in Appendix C.1.

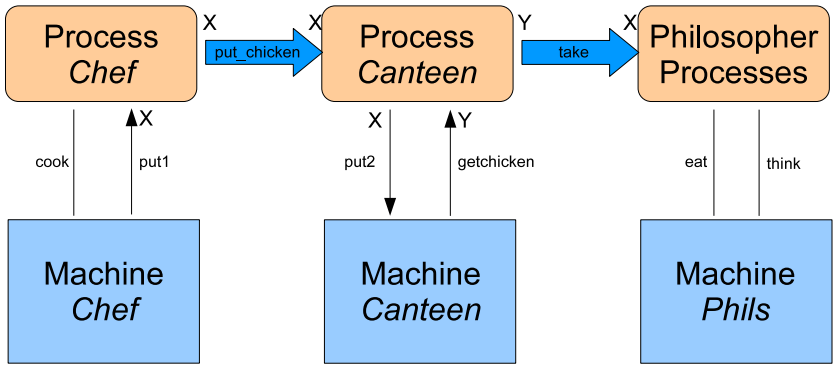

FIGURE 7.11: The decomposed wot-no-chicken model: Step 2

The variable *chef*, which shows the number of chicken in chef's hand, is updated by the *put* and *cook* operations in the B machine. The two corresponding CSP channels only appear in the *Chef* process. So for this variable, we can easily divide it into a new machine for the *Chef* process. The new machine contains the *chef* variable, and the two operations, *put* and *cook*. The situation for the *put* operation is more complex than that of *cook*, as it also updates another B variable *canteen*. That means the combine event *put* also belongs to another decomposed module *Canteen*, which contains the B variable *canteen*. We decomposed the *put* operation into two B operations which belong to two different modules. The *Chef* module has an event *put1*, which updates the B variable *chef*

in the *Chef* machine, whereas the *Canteen* module has an event *put2*, which updates variable *canteen* in the *Canteen* machine. The two events are connected through a communicate channel *put_chicken* between the two modules. The above decomposition step transforms an atomic B operation with two parallel data transition into two B operations on two separate B machines, and the two new B operations communicate through a CSP channel.

The other decomposed B operation is *getchicken*. Although it only update the *canteen* variable from the *Canteen* module, it is a synchronized event. The philosopher processes synchronize with the *Canteen* process on this event. Although the event does not update any state variable on the B machine of the *Phils* module, the philosopher processes need the output data from this event as a data input. Therefore, the *getchchicken* event is decomposed into a combined event *getchicken* in the *Canteen* module, and a communication channel *take* between *Canteen* and *Phils*.

Final Step: Decomposing the Philosophers

In the previous step, all the philosopher processes still remain in the same module. The state variable *state* consists of an array of natural numbers which indicate the status of all the philosophers. The philosopher processes update the state variable using the *think* and *eat* events. When decomposing the B machine, each array element of *state* is treated as an individual state variable for its corresponding philosopher process. As there is no direct communication or synchronization between these processes and the state variables can be clearly decomposed, we do not need to decompose any B operation. Each of the decomposed modules keeps a copy of the two B operations which update its state variables. The *Phils* module is finally be decomposed into four normal philosopher modules *Phil*, and a greedy philosopher module *XPhil*. In Appendix C.2, the specification of the decomposed *Phil* and *XPhil* modules are presented.

In [But93, But96, But06], similar event-based decomposition technique has been applied on classical B and Event-B. We give a discussion of this in Section 8.2.3.

In our experiments, each module of this decomposed model is model checked in the ProB tool, and also translated into Java programs. Finally, we employs the composition technique used in Section 7.3.1 to compose all the generated Java programs together. Then we test the two Java programs, one generated from the abstract model and the other from the decomposed model. The result does not show any difference in the behaviours of the two programs, except the decomposed one maintains its state on separate modules.

Chapter 8

Discussion

In this chapter, we draw some conclusions on the current development of this work, and discuss related techniques. We also propose several possible future research and development areas, which can improve the scalability and usability of this work.

8.1 Conclusions

This work was motivated by the recent trends on integrating state- and event- based formal methods for achieving a more expressive specification language. Our work is to implement such a combined B and CSP specification [BL05] in the Java language. The JCSP package [WM00b] gave us an important inspiration on constructing runnable applications from formal specifications.

We first examined the B+CSP semantics in ProB, and found it is too flexible and non-deterministic for implementation purposes. Thus, we developed a restricted semantics for combining B operations and CSP channels. The restriction is weak enough to allow most B+CSP models in ProB to be expressed, and is also strong enough to give clear synchronization and dataflow patterns between B operations and CSP channels, making it possible to be implemented in a conventional programming language, e.g. Java. The synchronization pattern of the restricted semantics is a multi-way synchronizing rendezvous: multiple CSP processes can synchronize on a combined event; the CSP channel and the B operation of a combined event need to synchronize with each other on the enablement of the event; there are state changes inside the combined event.

We also introduced the JCSProB package, which implements the restricted B+CSP semantics in Java. Although the development of the package is inspired by the development of JCSP and is partially based on JCSP, it is different from JCSP because of the semantics of B+CSP. The target of combined event implementations is to implement the multi-way synchronizing rendezvous pattern. We argue that in order to provide a safe

and effective multi-way synchronization implementation, the guarded external choice decisions of a system should be made in a sequential order. Thus we implemented a multi-way synchronization for the JCSProB event classes using a similar algorithm to [Bag89]. We also presented a new thread/process mechanism, which targets providing a memory-efficient solution for implementing CSP recursion. In the implementation, a process object can create and call a new process object without causing Java stack overflow error. This allows complex mutual recursions to be used in the CSP part of the combined specification. Additionally, we also developed a GUI package for the Java programs constructed with JCSProB. The GUI package and the configuration file allow user interactions with the underlying JCSProB programs. The GUI package also supports runtime checking on invariants, and user-defined assertions.

We constructed a set of translation rules, which formally define the correspondence between the B+CSP specification language and Java/JCSProB. We also developed an automated translation tool for implementing the translation rules. The translation tool is developed as an add-on to the ProB tool, which means users can automatically generate runnable Java programs from developed B+CSP specifications in ProB.

We tested and evaluated this work through a number of experiments. We evaluated the expressivity of the restricted B+CSP semantics, explore the coverage of both the semantics and the translation. The examples are also tested with fairness assertions to evaluate their behavioural properties. The scalability of this development strategy was also discussed with practical composition and decomposition efforts.

8.2 Related Works and Discussions

8.2.1 The *Circus* Translation

In Section 2.3.1.5, we introduced the *Circus* [WC01] specification language, and in Section 3.3.4 briefly discussed the automated translation [OC04, FC06] from *Circus* to Java programs using JCSP [WM00b]. It probably is the most closely related work to ours as it also translates integrated state- and event-based formal specifications into Java programs, and makes use of the JCSP package in its translation. Furthermore, it develops a GUI for user interaction with the generated Java programs.

A significant advantage of *Circus* is that it has formally defined refinement and composition rules for the integrated models, whereas such rules are still unavailable for B+CSP. With the refinement technique, an abstract *Circus* model can be developed into concrete models with refinement proofs. Therefore, currently, the *Circus* translation is based on a more mature integrated formal model, and provides a better scalability on development than our approach. More discussion on this issue can be found in Section 8.2.4.

The *Circus* translation makes use of the JCSP package to implement the semantics of the *Circus* specification language. However, as JCSP is mainly based on CSP/*occam*, the translation has many limitations. For example, it only allows use of the JCSP communication channels and barrier for communication. It does not support external choice with multi-way synchronization. Therefore, the *Circus* semantics that can be put into the translation are restricted. The restriction is mainly because of the incompatibility between the different CSP subsets [OC04] supported by *Circus* and *JCSP*. In Section 5.1.2, we described pragmatic difficulties (memory problems) in the direct implementation in JCSP of unbounded mutual tail recursion (a common idiom in CSP). This was solved with a new thread/process mechanism that efficiently hands over the executing thread of one process to another, allowing the memory resources of the first process to be reclaimed. Surprisingly, the *Circus* translation allows an existing process to directly create and call new process objects without modifying the thread/process structure in JCSP.

The GUI programs generated by the *Circus* translator contain simple graphical interfaces, and allow limited interaction with the user. Neither the system behaviours nor the system states are clearly displayed on the GUI. Our GUI package provides a better interaction with the user, and the GUI are configurable and extendable. Also, our GUI package supports runtime checking on invariants and user defined assertions, which presents useful features for evaluating behaviour properties of the generated Java programs.

8.2.2 Event-B and RODIN

The Event-B language [MAV05b] has recently evolved from the classical B-Method [Abr96] as a new generation of state-based formal modelling and development methodology. It has been developed as an essential part of the EU project RODIN. Unlike classical B, Event-B does not have a fixed syntax definition, which provides better expandability to new language constructs. The RODIN tool platform [ABHV06] is an open development environment for Event-B. Basically, it provides modeling, refinement and mathematical proof support for Event-B specifications.

Ongoing development in Event-B includes a RODIN plug-in development which maps an intermediate language, Object- oriented Concurrent-B (OC-B), to both Event-B models and Java programs. The new notation sits at the interface between Event-B and Java. A model specified using OC-B can be used for building a formal Event-B model. It is also can be translated into concurrent Java program. At the time of writing, this work remains unpublished.

It would be very useful to evolve the integrated B+CSP specifications from classical B to Event-B, and move the development environment from ProB to the RODIN platform.

As the semantics of Event-B is based on that of classical B, it would be feasible to develop an integrated Event-B and CSP model. Also, as Event-B and the RODIN platform provide very good expandability, it should not be difficult to introduce CSP to Event-B, or implement the translation in the RODIN platform. The definition of the integrated Event-B and CSP semantics, as well as its refinement and composition rules, would be the most important part of this future development.

8.2.3 Composition and Decomposition of B+CSP models

Being an event-based formal approach, CSP has a convenient way to practise composition and decomposition. The sequential and parallel composition structures can easily model the composition between different subsystems. Processes from different subsystems can use shared events to interact and communicate with each others.

A system model in B is an action system [BKS83b], which consists a number of state variables and uses some guarded atomic operations to change the system state. A B model specifies an isolated reactive system. For large scale, distributed systems, the system model should be able to be decomposed into parallel subsystems. Each of these subsystems maintains its own state, and can be refined or further decomposed into more concrete models.

In both the classical B method and the new Event-B, there are composition and decomposition approaches for constructing large-scale distributed system. In [MAV05a], a state-based composition and decomposition is proposed for the Event-B specification language. This technique is based on shared variables between subsystems. The shared variables are forced to be refined in the same functional manner on different subsystems. This restriction creates dependency among compositional subsystems.

On the other hand, Butler developed an event-based composition and decomposition technique [But93, But96, But97, But06] for both classical B and Event-B. The event-based composition and decomposition approach models the interaction between subsystems using message passing on shared actions. The idea this approach is based on the CSP semantics, and the correspondence between action systems and CSP, which was introduced in [Mor90].

In principle, an abstract B+CSP model should be able to be decomposed into a compositional model with multiple subsystems. To carry out such a system development, it would be better to decompose the system specification in the early design stages of the development process. Then each subsystem can be refined or further decomposed independently. A subsystem maintains its own state variables in the B machine, and the CSP specification defines the behaviour of the subsystem. It uses internal data transitions from the B part to change the state of the system. The CSP part of the combined specification specifies behaviours of the system, and the communication with

other subsystems. Composition and decomposition rules should define the allowed transformation for decomposing an abstract model into parallel subsystems, and the formal proofs for verifying their refinement. One important feature of the composition rule is that a subsystem should be refined or further decomposed independently from the rest the system.

Unfortunately, the composition and decomposition rules of B+CSP have not been formally defined, and ProB only supports model checking on one B and CSP specification pair. The CSP‖B approach developed a composition technique [ST05] for its combined model through message passing. In CSP‖B, a B machine communicates with its CSP controller, and various CSP controllers can communicate with each other through shared CSP channels. So far as we know, there is no direct tool support for the CSP‖B models, so it provides a compositional verification approach, which includes several steps using various tools, to verify the consistency of the compositional models.

Arguably, the restricted semantics of B+CSP discussed in Section 4.3 is very close to the semantics of CSP‖B models. It should be possible to develop a similar composition technique for B+CSP models. In Section 7.3, we tested a CSP‖B example from [STE05] in the ProB environment. We model checked the two compositional components separately, and generated Java programs from the two models. Although we still cannot combine the verification results from ProB into a compositional correctness proof, the effort of model checking a CSP‖B component using ProB can still help the CSP‖B development by simplifying the verification procedures. We believe it is also possible to establish a connection between the CSP‖B and B+CSP approaches.

The practical composition and decomposition efforts in Section 7.3 can also be compared with the event-based composition and decomposition technique used in [But96] and [But06]. Although that approach is based on pure state-based formal specifications, the composition of multiple B/Event-B machines is defined based on the CSP semantics and Morgan's work in [Mor90]. The composition of B operations (or Event-B events) from different machines are composed using parallel and hiding semantics from CSP. It would be ideal if we can utilize the formal basis of this technique to develop a similar composition for B+CSP.

Comparing the experimental attempts in Section 7.3 with existing composition/decomposition approaches shows that it is possible to develop a similar compositional development technique for B+CSP. Also, the JCSProB package and the translation showed good support for this potential composition approach. Developing such formal composition rules and technique would be a promising future work for the B+CSP development.

8.2.4 Refinement Rules for B+CSP

Currently, refinement rules for B+CSP are also unavailable, which means the user does not have systematic techniques for developing B+CSP models.

Circus [WC01], which also integrates state- and event- based formal methods, has developed refinement rules [CSW03] for its combined specification. The refinement rules of *Circus* inherit the *correct-by-construction* style refinement rules from the Z method, and extend them with refinement rules for the CSP part. The *correct-by-construction* style refinement defines a series of rules on the transformations from an abstract model to a more concrete one. Each of these transformation steps guarantees refinement. As the transformation targets, which the user can get, are strictly defined in the refinement rules, it sometimes restricts the freedom of development.

The refinement development in the B method is slightly different. It employs a *posit-and-prove* approach, in which the developer provides both the abstract and refinement models. The refinement rules define what kinds of proof obligations need to be proved to guarantee refinement, and usually this step is done by the proof obligation generation tool. Many proof obligations can be proved automatically using theorem provers. In some cases, *gluing invariants* need to be introduced to help proving some proof obligations, which cannot be automatically proved by the prover.

The user gets more freedom on building the refinement model from the *posit-and-prove* approach. But it is not so easy to define the refinement rules for discovering proof obligations. Especially, for the integrated formal specifications, it is more difficult to define proof obligation rules across various specification languages. That is the main reason why there are still no refinement rules for the B+CSP model.

One possible solution is restricting the allowable refinement transformation for B+CSP using the *correct-by-construction* style refinement rules. There also should be *posit-and-prove* rules which are used to generate proof obligations for handling the possible side conditions brought by the *correct-by-construction* rules. This can reduce the complexity of defining such refinement rules, although it would probably limit the development of the refinement model at the same time. Certainly, more investigation work is still required to justify a proper method for developing such a solution. It would be a very important breakthrough for the B+CSP development to have the refinement rules developed.

8.2.5 Compatibility with JCSP

Although we developed a new JCSProB package, many JCSP classes are also used in the development. The main compatibility issue between JCSP and JCSPorB is on our implementation of external choice in JCSProB. In Section 5.2, we explained that a combined JCSProB event object cannot be put into the same external choice with a

JCSP channel object, because we implement the external choice with a different class *Alter* from the *Alternative* class used in JCSP.

That is because when we started to develop the JCSProB package, the *AltingBarrier* class, which supports guarded external choice with multi-way synchronization, still had not been introduced to the JCSP package. Now as the multi-way synchronization has been fully implemented in JCSP, it is possible to re-implement the synchronization guard of the combined B+CSP event using the *AltingBarrier* class. That would make the event classes using compatible synchronization guards as the JCSP channels, and would remove the current restriction on external choice.

In Section 5.5.4, we discussed the extra cost introduced by the JCSProB thread container in sequential composition. Each child process introduces a new thread container, which can be unnecessary in many cases. A possible solution is proposed. Before running a child process object in a sequential composition, we first test if the process introduce recursion. If yes, the process object should be run inside a new thread container, otherwise, it should be run just inside the thread container of the parent process to avoid extra cost. To test the if a process has recursion, the process or its thread container needs to keep a tree of its descendant process objects. And the tree needs to be built when the process object is created, and before the process starts to run. We will compare the cost introduced by the this solution with the current one to see if it is substantial to implement such a feature.

8.2.6 Formal Correctness Verification for the JCSProB

A correctness verification for the translation is also a potential future development. In [RRS03, OC04], the translations are discussed without considering the correctness proofs. Formal verification which proves the correctness of the translation in terms of semantic models of the specification and Java programs respectively would be the best solution. We propose a more modest approach based on [WM00a] for future work.

In [WM00a], the correctness of the JCSP communication channel classes is verified. Each JCSP channel class (i.e. Java implementation) is formally specified as a CSP model. The desired channel behaviour (which the JCSP class implements) is also specified using CSP. The FDR tool [For03] is employed to verify equivalent between the two CSP models. The proving strategy starts with the simple *One2OneChannel* class without alternation, and gradually builds formal models for more complex JCSP channel classes.

To verify the correctness of the event class in JCSProB, a similar strategy is proposed to prove that the Java implementation refines the B+CSP semantics. First, a B+CSP model for the event class is constructed. Then, the concerned behaviour of the combined B+CSP event is also specified using B+CSP. As the ProB tool supports refinement

checking between B+CSP models, we can check if the event class correctly implements the B+CSP semantics.

Constructing B+CSP models for the JCSProB implementations of the B+CSP events with full functionality would be a difficult task. It would be better to starting with abstract models of event classes with limited functionality, and gradually building concrete models with more complex B+CSP semantics.

Appendix A

Translation Rules

Rule 0. $\mathbb{T}\mathrm{ranslator} \Rightarrow$

$\mathbb{B}\$\mathrm{tate}\mathbb{C}\mathrm{lass}$
$\mathbb{B}\$\mathrm{etC}\mathrm{lasses}\lceil \mathit{BSets} \rfloor$
$\mathbb{P}\mathrm{rocess}\mathbb{C}\mathrm{lasses}\lceil \mathit{Procs} \rfloor$
$\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lasses}\lceil \mathit{Evs} \rfloor$
$\{\mathbb{X}\mathrm{ML}\mathbb{C}\mathrm{onfiguration}\}_{[GUI]}$
$\mathbb{M}\mathrm{ain}\mathbb{C}\mathrm{lass}$

B+CSP: *Clause_sets* $\vdash$ '**SETS**' *BSets*
Supplement: *Procs* include all the CSP processes in the B+CSP model
Supplement: *Evs* include all the B+CSP events in the B+CSP model

□

Rule 1. $\mathbb{B}\$\mathrm{tate}\mathbb{C}\mathrm{lass} \Rightarrow$

$\mathbb{P}\mathrm{ackage}\mathbb{D}\mathrm{ef}$
$\mathbb{C}\mathrm{lass}\mathbb{H}\mathrm{eader}$
`public class` $\mathbb{M}\mathrm{achName}$`'_'machine extends JcspVar{`
$\quad \mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil \mathit{Vars} \rfloor$
$\quad \mathbb{C}\mathrm{onsts}\mathbb{D}\mathrm{ecl}\lceil \mathit{Consts} \rfloor$
$\quad \mathbb{B}\$\mathrm{tate}\mathbb{C}\mathrm{onstructor}$
$\quad \mathbb{B}\$\mathrm{tate}\mathbb{C}\mathrm{heck}$
$\quad \{\mathbb{B}\mathbb{V}\mathrm{ars}\mathbb{V}\mathrm{ec}\}_{[GUI]}$
$\quad$ }

B+CSP: *Clause_variables* $\vdash$ '**VARIABLES**' *Vars*
B+CSP: *Clause_constants* $\vdash$ '**CONSTANTS**' *Consts*

□

Rule 1.1. $\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil Vars\rfloor \Rightarrow \mathbb{V}\mathrm{ar}\mathbb{D}\mathrm{ecl}\lceil Var\rfloor^*$

B+CSP: $Vars \vdash Var^*$

□

Rule 1.1.1. $\mathbb{V}\mathrm{ar}\mathbb{D}\mathrm{ecl}\lceil Var\rfloor \Rightarrow \mathbb{T}\mathrm{ype}\lceil Var\rfloor\ \mathbb{V}\mathrm{ar\_Name}\lceil Var\rfloor$ `';'` □

Rule 1.2. $\mathbb{C}\mathrm{onst}\mathbb{D}\mathrm{ecl}\lceil Consts\rfloor \Rightarrow \mathbb{C}\mathrm{onst}\mathbb{D}\mathrm{ecl}\lceil Const\rfloor^*$

B+CSP: $Consts \vdash Const^*$

□

Rule 1.2.1. $\mathbb{C}\mathrm{onst}\mathbb{D}\mathrm{ecl}\lceil Const\rfloor \Rightarrow \mathbb{T}\mathrm{ype}\lceil Const\rfloor\ \mathbb{V}\mathrm{ar\_Name}\lceil Const\rfloor$ `';'` □

Rule 1.3. $\mathbb{B}\$\mathrm{tate}\mathbb{C}\mathrm{onstructor} \Rightarrow$

```
public MachName'_'machine'('String MachName')'{
    super'("'MachName'");'
    VarsInit⌈Vars⌋
    ConstsInit⌈Consts⌋
    { buildArgVec'();' }[GUI]
```

B+CSP: *Clause_variables* ⊢ '**VARIABLES**' *Vars*
B+CSP: *Clause_constants* ⊢ '**CONSTANTS**' *Consts*

□

Rule 1.3.1. $\mathbb{V}\mathrm{ars}\mathbb{I}\mathrm{nit}\lceil Vars\rfloor \Rightarrow \mathbb{V}\mathrm{ar}\mathbb{I}\mathrm{nit}\lceil Var\rfloor^*$

B+CSP: $Vars \vdash Var^*$

□

Rule 1.3.2. $\mathbb{V}\mathrm{ar}\mathbb{I}\mathrm{nit}\lceil Var\rfloor \Rightarrow$

`this'.'` $\mathbb{V}\mathrm{ar\_Name}\lceil Var\rfloor$ `'='` $\mathbb{B}\mathbb{E}\mathrm{xpr}\lceil Var\_Init\rfloor$ `';'`

Supplement: *Var_Init* is the initialization information for the variable *Var* from the **INITIALISATION** clause of the B part.

□

Rule 1.3.3. $\mathbb{C}\mathrm{onst}\mathbb{I}\mathrm{nit}\lceil Consts\rfloor \Rightarrow \mathbb{C}\mathrm{onst}\mathbb{I}\mathrm{nit}\lceil Const\rfloor^*$

B+CSP: $Consts \vdash Const^*$

□

Rule 1.3.4. $\mathbb{C}\mathrm{onst}\mathbb{I}\mathrm{nit}\lceil Const\rfloor \Rightarrow$

`this‘.’` $\mathbb{V}\mathrm{ar\_Name}\lceil Const\rfloor$ `‘=’` $\mathbb{B}\mathrm{Expr}\lceil Const\_Init\rfloor$ `‘;’`

Supplement: *Const_Init* is the initialization information for the variable *Const* from the **PROPERTIES** or **VALUES** clause of the B part.

□

Rule 1.4. $\mathbb{B}\$\mathrm{tate}\mathbb{C}\mathrm{heck} \Rightarrow$

```
public synchronized void check‘(){’
    try‘{’
        if‘(!(’ BInvs ‘)){’
            System‘.’out‘.’println
            ‘("’Invariant check failed‘!");’
            {RecordsFrame‘.’getInstant‘().’terminate‘();’}[GUI]
            this‘.’terminate‘();’
            ‘}’
    ExceptionHandler
‘}’
```

□

Rule 2. $\mathbb{B}\$\mathrm{et}\mathbb{C}\mathrm{lasses}\lceil BSets\rfloor \Rightarrow \mathbb{B}\$\mathrm{et}\mathbb{C}\mathrm{lass}\lceil BSet\rfloor^*$

B+CSP: $BSets \vdash BSet^{+';'}$

□

Rule 2.1. $\mathbb{B}\$\mathrm{et}\mathbb{C}\mathrm{lass}\lceil BSet\rfloor \Rightarrow$

```
PackageDef
ClassHeader
public enum Print⌈BSetName⌋ ‘{’
    Print⌈Iden⌋+',' 
‘}’
```

B+CSP: $BSet \vdash BSetName' = \{'\ Iden^{+','}\ '\}'$

□

Rule 3. $\mathbb{P}\mathrm{rocess}\mathbb{C}\mathrm{lasses}\lceil Procs\rfloor \Rightarrow \mathbb{P}\mathrm{rocess}\mathbb{C}\mathrm{lass}\lceil Proc\rfloor^*$

B+CSP: $Procs \vdash Proc^*$

□

Rule 3.1. $\mathbb{P}\mathrm{rocess}\mathbb{C}\mathrm{lass}\lceil Process\rfloor \Rightarrow$

$\mathbb{P}\mathrm{ackage}\mathbb{D}\mathrm{ef}$

$\mathbb{C}\mathrm{lass}\mathbb{H}\mathrm{eader}$

`public class` $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil ProcName\rfloor$ `extends`

$\{$`BCSProcess`$\}_{[NoGUI]} \mid \{$`BGProcess`$\}_{[GUI]}$ '{'

$\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{D}\mathrm{ecl}\lceil Chs\rfloor$

$\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil Vars\rfloor$

$\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{onstrucor}$

$\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}\lceil ProcBody\rfloor$

'}'

B+CSP: *Proc* $\vdash$ *ProcHeader* '=' *ProcBody*

B+CSP: *ProcHeader* $\vdash$ *ProcName* {'('VarsList')'}

Supplement: *Chs consists of all the channels/events called by the process, or its sub-process*

Supplement: *Vars consists of all the variables used in the process*

□

Rule 3.2. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{D}\mathrm{ecl}\lceil Chs\rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{D}\mathrm{ecl}\lceil Ch\rfloor^*$

Supplement: $Chs \vdash Ch^*$

□

Rule 3.2.1. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{D}\mathrm{ecl}\lceil Ch\rfloor \Rightarrow$

$\mathbb{C}\mathrm{hannel}\mathbb{T}\mathrm{ype}\lceil Ch\rfloor$ $\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ch\rfloor$ ';' □

Rule 3.3. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil ProcVars\rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{D}\mathrm{ecl}\lceil ProcVar\rfloor^*$

Supplement: $ProcVars \vdash ProcVar^*$

□

Rule 3.3.1. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{D}\mathrm{ecl}\lceil ProcVar\rfloor \Rightarrow$

$\mathbb{T}\mathrm{ype}\lceil ProcVar\rfloor$ $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{O}\mathrm{bj}\lceil ProcVar\rfloor$ ';' □

Rule 3.4. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{onstructor} \Rightarrow$

`public` $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil ProcName\rfloor$ '(' '('

$\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{T}\mathrm{ype}\mathbb{L}\mathrm{ist}\lceil Chs\rfloor$ ','

$\{$`ChannelOutput conf ','`$\}_{[GUI]}$

$\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{T}\mathrm{ype}\mathbb{L}\mathrm{ist}\lceil \mathit{Vars} \rfloor$`')'{`
`{super()';'}`$_{[NoGUI]}$ | `{ super(conf)';' }`$_{[GUI]}$
$\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{A}\mathrm{ssign}\lceil \mathit{Chs} \rfloor$
$\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{A}\mathrm{ssign}\lceil \mathit{ExtVars} \rfloor$
`'}'`

B+CSP: ProcHeader $\vdash$ *ProcName* {'('VarsList')'}
Supplement: *Chs is all the channels/evnets involved in this process or its sub-processes.*
Supplement: *Vars is all the variables involved in this process.*
Supplement: *ExtVars is all the external variables in VarsList of the ProcHeader definition.*

□

Rule 3.4.1. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{T}\mathrm{ype}\mathbb{L}\mathrm{ist}\lceil \mathit{Chs} \rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{T}\mathrm{ype}\lceil \mathit{Ch} \rfloor^{+','}$

Supplement: $\mathit{Chs} \vdash \mathit{Ch}^*$

□

Rule 3.4.1.1. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{T}\mathrm{ype}\lceil \mathit{Ch} \rfloor \Rightarrow \mathbb{C}\mathrm{hannel}\mathbb{T}\mathrm{ype}\lceil \mathit{Ch} \rfloor\ \mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil \mathit{Ch} \rfloor$ □

Rule 3.4.2. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{T}\mathrm{ype}\mathbb{L}\mathrm{ist}\lceil \mathit{ProcVars} \rfloor \Rightarrow$
$\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\lceil \mathit{ProcVar} \rfloor^{+','}$

Supplement: $\mathit{ProcVars} \vdash \mathit{ProcVar}^*$

□

Rule 3.4.2.1. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\lceil \mathit{ProcVar} \rfloor \Rightarrow \mathbb{T}\mathrm{ype}\lceil \mathit{ProcVar} \rfloor\ \mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{O}\mathrm{bj}\lceil \mathit{ProcVar} \rfloor$
□

Rule 3.4.3. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hs}\mathbb{A}\mathrm{ssign}\lceil \mathit{Chs} \rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{A}\mathrm{ssign}\lceil \mathit{Ch} \rfloor^*$

Supplement: *Chs is all the channels/evnets involved in this process or its sub-processes.*
Supplement: $\mathit{Chs} \vdash \mathit{Ch}^*$

□

Rule 3.4.3.1. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{h}\mathbb{A}\mathrm{ssign}\lceil \mathit{Ch} \rfloor \Rightarrow$
`this'.'` $\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil \mathit{Ch} \rfloor$ `'='` $\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil \mathit{Ch} \rfloor$ `';'` □

Rule 3.4.4. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ars}\mathbb{A}\mathrm{ssign}\lceil \mathit{ProcVars} \rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{A}\mathrm{ssign}\lceil \mathit{ProcVar} \rfloor^*$

Supplement: $\mathit{ProcVars} \vdash \mathit{ProcVar}^*$

□

Rule 3.4.4.1. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{A}\mathrm{ssign}\lceil ProcVar \rfloor \Rightarrow$

`this'.'`$\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{O}\mathrm{bj}\lceil ProcVar \rfloor$ `'='` $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{ar}\mathbb{O}\mathrm{bj}\lceil ProcVar \rfloor$`';'` □

Rule 3.5. $\mathbb{P}\mathrm{roc}\mathbb{R}\mathrm{un}\lceil ProcBody \rfloor \Rightarrow$

```
public void run'(){'
```
$\{$`this'.'createGUIProc'();'`$\}_{[GUI]}$
$\{$`start'();'`$\}_{[GUI]}$
$\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcBody \rfloor$
$\{$ `end'();'`$\}_{[GUI]}$
`'}'` □

Rule 3.6. $\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcB \rfloor \Rightarrow$

$\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Parallel}\lceil ProcBParallel \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{ReplParallel}\lceil ProcBReplParallel \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Interleave}\lceil ProcBInterleave \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{ReplInterleave}\lceil ProcBReplInterleave \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Sequence}\lceil ProcBSequence \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{IfThen}\lceil ProcBIfThen \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Prefix}\lceil ProcBPrefix \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Choice}\lceil ProcBChoice \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{ReplChoice}\lceil ProcBReplChoice \rfloor$
$\mid \mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Call}\lceil ProcBCall \rfloor$

B+CSP: $ProcBody \vdash ProcBParallel$
$\mid ProcBReplParallel$
$\mid ProcBInterleave$
$\mid ProcBReplInterleave$
$\mid ProcBSequence$
$\mid ProcBIfThen$
$\mid ProcBPrefix$
$\mid ProcBChoice$
$\mid ProcBReplChoice$
$\mid ProcBCall$

□

Rule 3.6.1. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Parallel}\lceil ProcBParallel \rfloor \Rightarrow$

$\mathbb{P}\mathrm{ara}\mathbb{C}\mathrm{hs}\mathbb{N}\mathrm{ums}\lceil Chs \rfloor$
$\{$ `new CSParallel'('` $\}_{[NoGUI]} \mid \{$ parallelCtrl`'('` $\}_{[GUI]}$

$\{$ `new BCSProcess‘[]{’` $\}_{[NoGUI]} \mid \{$ `new BGProcess‘[]{’` $\}_{[GUI]}$
$\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\lceil Proc\rfloor^{+','}$
`‘}’`
$\{$ `‘)’.run‘();’` $\}_{[NoGUI]} \mid \{$ `‘);’` $\}_{[GUI]}$

B+CSP: $ProcBParallel \vdash Proc^{+'\|'}$

Supplement: *Chs* includes all the B+CSP events that have multi-way synchronizing.

Supplement: *Procs* includes all the process in the parallel composition.

□

Rule 3.6.2. $\mathbb{P}\mathrm{ara}\mathbb{C}\mathrm{hs}\mathbb{N}\mathrm{ums}\lceil Chs\rfloor \Rightarrow \mathbb{P}\mathrm{ara}\mathbb{C}\mathrm{h}\mathbb{N}\mathrm{ums}\lceil Ch\rfloor^*$

Supplement: $Chs \vdash Ch^*$

□

Rule 3.6.2.1. $\mathbb{P}\mathrm{ara}\mathbb{C}\mathrm{h}\mathbb{N}\mathrm{ums}\lceil Ch\rfloor \Rightarrow$
$\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ch\rfloor$ `‘.’inc‘_’syn‘_’proces‘_’no‘(’` $\mathbb{I}\mathrm{nteger}$ `‘);’`

Supplement: Rule $\mathbb{I}\mathrm{nteger}$ here should be the number of processes which synchronize on the event Ch.

□

Rule 3.6.3. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{R}\mathrm{epl}\mathbb{P}\mathrm{arallel}\lceil ProcBReplParallel\rfloor \Rightarrow$
$\mathbb{P}\mathrm{ara}\mathbb{C}\mathrm{hs}\mathbb{N}\mathrm{ums}\lceil Chs\rfloor$
$\{$ `new CSParallel‘(’` $\}_{[NoGUI]} \mid \{$ parallelCtrl‘(’ $\}_{[GUI]}$
$\{$ `new BCSProcess‘[]{’` $\}_{[NoGUI]} \mid \{$ `new BGProcess‘[]{’` $\}_{[GUI]}$
$\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\lceil Proc\rfloor^{+',}$
`‘}’`
$\{$ `‘)’.run‘();’` $\}_{[NoGUI]} \mid \{$ `‘);’` $\}_{[GUI]}$

B+CSP: $ProcBReplParallel \vdash [|\ \{Chs\}\ |]x \in N@Proc$

□

Rule 3.6.4. $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\lceil Proc\rfloor \Rightarrow \mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\mathbb{N}\mathrm{ew}\lceil Proc\rfloor \mid \mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{xisting}\lceil Proc\rfloor$

Supplement: The rule $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{xisting}$ is applied when a named process is called.

□

Rule 3.6.4.1. $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\mathbb{N}\mathrm{ew}\lceil Proc\rfloor \Rightarrow$
$\{$`new BCSProcess‘(){’`$\}_{[NoGUI]} \mid \{$ `new BGProcess‘(){’` $\}_{[GUI]}$

```
        public void run‘(){’
            {this‘.’createGUIProc‘();’}[GUI]
            {start‘();’}[GUI]
            ℙrocℍ⌈ProcBody⌋
            { end‘();’}[GUI]
        ‘}’
    ‘}’
```

B+CSP: $Proc \vdash ProcName' =' ProcBody$

□

Rule 3.6.4.2. $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{xist}\lceil Proc\rfloor \Rightarrow$

```
    new ℙrocℂlassℕame⌈ProcName⌋‘(’
        ℙrocℂhsList⌈Chs⌋
        { configure‘,’ }[GUI]
        ℙrocVarsList⌈Vars⌋
    ‘)’
```

B+CSP: $Proc \vdash ProcName' =' ProcBody$

Supplement: Chs includes all the channels/events called by the process

Supplement: $ProcVars$ includes all the process variables in the process

□

Rule 3.6.4.3. $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{hsList}\lceil Chs\rfloor \Rightarrow \mathbb{C}\mathrm{hO}\mathbb{b}\mathrm{jName}\lceil Ch\rfloor^{+','}$

Supplement: $Chs \vdash Ch^*$

□

Rule 3.6.4.4. $\mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{arsList}\lceil Vars\rfloor \Rightarrow \mathbb{P}\mathrm{roc}\mathbb{V}\mathrm{arO}\mathbb{b}\mathrm{j}\lceil Var\rfloor^{+','}$

Supplement: $Vars \vdash Var^*$

□

Rule 3.6.5. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Interleave}\lceil ProcBInterleave\rfloor \Rightarrow$

```
    { new CSParallel‘(’ }[NoGUI] | { parallelCtrl‘(’ }[GUI]
        { new BCSProcess‘[]{’ }[NoGUI] | { new BGProcess‘[]{’ }[GUI]
            ℂall℗roc⌈Proc⌋+',',
        ‘}’
```

{ ‘)’.run‘();’ }$_{[NoGUI]}$ | { ‘);’ }$_{[GUI]}$

B+CSP: $ProcBInterleave \vdash Proc^{+'|||'}$

□

Rule 3.6.6. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Repl}\mathbb{I}\mathrm{nterleave}\lceil ProcBReplInterleave \rfloor \Rightarrow$

{ new CSParallel‘(’ }$_{[NoGUI]}$ | { parallelCtrl‘(’ }$_{[GUI]}$
{ new BCSProcess‘[]{’ }$_{[NoGUI]}$ | { new BGProcess‘[]{’ }$_{[GUI]}$
$\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\lceil Proc \rfloor^{+',}$
‘}’
{ ‘).’run‘();’ }$_{[NoGUI]}$ | { ‘);’ }$_{[GUI]}$

B+CSP: $ProcBInterleave \vdash ||| \; x \in N@Proc$

□

Rule 3.6.7. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Sequence}\lceil ProcBSequence \rfloor \Rightarrow$

{ new CSPSequence‘(’ }$_{[NoGUI]}$ | { sequenceCtrl‘(’ }$_{[GUI]}$
{ new BCSProcess‘[]{’ }$_{[NoGUI]}$ | { new BGProcess‘[]{’ }$_{[GUI]}$
$\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\lceil Proc \rfloor^{+',}$
‘}’
{ ‘).’run‘();’ }$_{[NoGUI]}$ | { ‘);’ }$_{[GUI]}$

B+CSP: $ProcBSequence \vdash Proc^{+';'}$

□

Rule 3.6.8. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{If}\mathbb{T}\mathrm{hen}\lceil ProcBIfThen \rfloor \Rightarrow$

if‘(’ $\mathbb{P}\mathrm{roc}\mathbb{C}\mathrm{ond}\lceil CSPCdtn \rfloor$ ‘({’
$\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcB \rfloor$
‘}’
{ else‘{’
$\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcB \rfloor$
‘}’ }

B+CSP: $ProcBIfThen \vdash 'if' CSPCdtn 'then' ProcE \{'else' ProcE\}$

□

Rule 3.6.9. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{Prefix}\lceil ProcBPrefix \rfloor \Rightarrow$

$\mathbb{C}\mathrm{hannel}\mathbb{C}\mathrm{alls}\lceil ChCall \rfloor$
$\mathbb{P}\mathrm{roc}\mathbb{E}\lceil ProcB \rfloor$

B+CSP: $ProcBPrefix \vdash ChCall \rightarrow ProcB$

□

Rule 3.6.9.1. $\mathbb{C}\mathrm{hannel}\mathbb{C}\mathrm{alls}\lceil ChCall \rfloor \Rightarrow$

$\mathbb{C}\mathrm{ombined}\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{all}\lceil ChCall \rfloor \mid \mathbb{C}\mathrm{omm}\mathbb{C}\mathrm{hannel}\mathbb{C}\mathrm{all}\lceil ChCall \rfloor$ □

Rule 3.6.9.2. $\mathbb{C}\mathrm{ombined}\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{all}\lceil ProcB \rfloor \Rightarrow$

$\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{all}\lceil ChCall \rfloor \mid \mathbb{E}\mathrm{vent}\mathbb{R}\mathrm{eturn}\lceil ChCall \rfloor$ □

Rule 3.6.9.3. $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{all}\lceil Ev \rfloor \Rightarrow$

$\{\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor$ `‘.’ready‘(’`

`{ inputVec‘(’newObject‘[]{’`

$\mathbb{C}\mathbb{S}\mathbb{P}\mathbb{E}\mathrm{xpr}\mathbb{L}\mathrm{ist}\lceil CSPExpr^* \rfloor$

`‘}’ }`

`‘);’ }`$_{[NoGUI]}$

`| { channelCall‘(’`

$\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor$ `‘.’ready‘(’`

`{ ‘,’inputVec‘(’newObject‘[]{’`

$\mathbb{C}\mathbb{S}\mathbb{P}\mathbb{E}\mathrm{xps}\mathbb{L}\mathrm{ist}\lceil CSPExpr^* \rfloor$

`‘}’ }`

`‘);’ }`$_{[GUI]}$

CSP: $Ev \vdash EvName$'!'$CSPExpr^*$

B: $Ev \vdash EvName$'('Exprs')'

□

Rule 3.6.9.4. $\mathbb{E}\mathrm{vent}\mathbb{R}\mathrm{eturn}\lceil Ev \rfloor \Rightarrow$

`{ ‘{’ Vector‘<’Object‘>’ rtnVec ‘=’` $\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor$ `‘.’ready‘(’`

`{ inputVec‘(’newObject‘[]{’`

$\mathbb{C}\mathbb{S}\mathbb{P}\mathbb{E}\mathrm{xpr}\mathbb{L}\mathrm{ist}\lceil InExpr^* \rfloor$

`‘}’ }`

`‘);’`

$\mathbb{E}\mathrm{vent}\mathbb{R}\mathrm{eturn}\mathbb{V}\mathrm{alues}\lceil OutVars \rfloor$

`}`$_{[NoGUI]}$

`| { ‘{’ Vector‘<’Object‘>’ rtnVec ‘=’ channelCall‘(’`

$\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor$ `‘.’ready‘(’`

`{ ‘,’inputVec‘(’newObject‘[]{’`

$\mathbb{C}\mathbb{S}\mathbb{P}\mathbb{E}\mathrm{xpr}\mathbb{L}\mathrm{ist}\lceil InExpr^* \rfloor$

`‘}’ }`

‘);’
EventReturnValues⌈*OutVars*⌋
}$_{[GUI]}$

CSP: *Ev* ⊢ *EvName*’!’*InExpr**’?’OutVars
B: *Ev* ⊢ OutVars ← *EvName*’(’InVars’)’

□

Rule 3.6.9.4.1. EventReturnValues⌈*Vars*⌋ ⇒ EventReturnValue⌈*Var*⌋*

Supplement: *Vars* ⊢ *Var**

□

Rule 3.6.9.4.2. EventReturnValue⌈*Var*⌋ ⇒
ProcVarObj⌈*Var*⌋ ‘=’ `rtnVec.elementAt‘(’` Integer ‘);’

Supplement: The *Integer* values are recursively generated from **Rule 3.6.9.4.1**

□

Rule 3.6.9.5. CommChannelCall⌈*Ch*⌋ ⇒
CommChannelRead⌈*InputCh*⌋ | CommChannelWrite⌈*OutputCh*⌋

CSP: *Ch* ⊢ *InputCh* | *OutputCh*

□

Rule 3.6.9.5.1. CommChannelRead⌈*InputCh*⌋ ⇒
{ ProcVarObj⌈*Var*⌋ ‘=’ ChObjName⌈*ChName*⌋ ‘.’`read‘();’` }$_{[NoGUI]}$
| { ProcVarObj⌈*Var*⌋ ‘=’ `channelRead‘(’`
ChObjName⌈*ChName*⌋ ‘,"’ ChObjName⌈*ChName*⌋
‘");’ }$_{[GUI]}$

CSP: *InputCh* ⊢ *ChName*’?’*Var*

□

Rule 3.6.9.5.2. CommChannelWrite⌈*OutputCh*⌋ ⇒
{ ChObjName⌈*ChName*⌋ ‘.’`write‘(’` ProcVarObj⌈*Var*⌋ ‘);’ }$_{[NoGUI]}$
| { { `channelWrite‘(’`
ChObjName⌈*ChName*⌋ ‘,"’ ChObjName⌈*ChName*⌋ ‘",’
ProcVarObj⌈*Var*⌋
‘");’ }$_{[GUI]}$

CSP: $InputCh \vdash ChName\text{'?'}Var$

□

Rule 3.6.10. $\mathbb{P}\mathrm{rocE}\mathbb{C}\mathrm{hoice}\lceil ProcBChoice\rfloor \Rightarrow$
$\mathbb{P}\mathrm{rocE}\mathbb{C}\mathrm{hoice}\mathbb{J}\mathbb{C}\$\mathbb{P}\mathrm{roB}\lceil ProcBChoice\rfloor$
$|\ \mathbb{P}\mathrm{rocE}\mathbb{C}\mathrm{hoice}\mathbb{J}\mathbb{C}\$\mathbb{P}\lceil ProcBChoice\rfloor$ □

Rule 3.6.10.1. $\mathbb{P}\mathrm{rocE}\mathbb{C}\mathrm{hoice}\mathbb{J}\mathbb{C}\$\mathbb{P}\mathrm{roB}\lceil ProcBChoice\rfloor \Rightarrow$
`BCSPGuard‘[]’ in ‘=’ ‘{’` $\mathbb{P}\mathrm{rocC}\mathbb{C}\mathrm{hs}\mathbb{L}\mathrm{ist}\lceil ProcBPrefix^*\rfloor$ `‘};’`
`Vector‘<’Vector‘<’Object‘>>’ choiceVec`
`‘=’ new Vector‘<’Vector‘<’Object‘>>();’`
$\mathbb{C}\mathrm{hoice}\mathbb{V}\mathrm{alue}\mathbb{A}\mathrm{ssign}\lceil ProBPrefix^*\rfloor$
`{ Alter alt ‘= ’new Alter‘(’in‘,’ choiceVec ‘);’`
`switch‘(’alt‘.’select‘(){’`
$\mathbb{C}\mathrm{hoices}\lceil ProBPrefix^*\rfloor$
`‘}’ }`$_{[NoGUI]}$
`| { switch‘(’choiceCtrl‘(’in‘,’choiceVec‘)){’`
$\mathbb{C}\mathrm{hoice}\lceil ProcBPrefix^*\rfloor$
`‘}’ }`$_{[GUI]}$

CSP: $ProcBChoice \vdash ProcBPrefix^{+'\Box'}$

□

Rule 3.6.10.1.1. $\mathbb{G}\mathrm{uards}\mathbb{L}\mathrm{ist}\lceil ProcBPrefix^*\rfloor \Rightarrow \mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev\rfloor^{+`,'}$

CSP: $ProcBPrefix \vdash Ev \rightarrow ProcB$

□

Rule 3.6.10.1.2. $\mathbb{C}\mathrm{hoice}\mathbb{V}\mathrm{alue}\mathbb{A}\mathrm{ssign}\lceil ProcBPrefix\rfloor \Rightarrow$
`{ choiceVec‘.’addElement‘(’inputVec‘(’new Object‘[]{’`
$\mathbb{C}\$\mathbb{P}\mathbb{E}\mathrm{xpr}\mathbb{L}\mathrm{ist}\lceil CSPExpr^*\rfloor$
`‘}))’`

CSP: $ProcBPrefix \vdash Ev \rightarrow ProcB$
CSP: $Ev \vdash EvName'!'CSPExpr^* \mid EvName\text{'!'}CSPExpr^{*}\text{'?'}OutVars$

□

Rule 3.6.10.2. $\mathbb{P}\mathrm{rocE}\mathbb{C}\mathrm{hoice}\mathbb{J}\mathbb{C}\$\mathbb{P}\lceil ProcBChoice\rfloor \Rightarrow$
`Guard‘[]’ in ‘=’ ‘{’` $\mathbb{P}\mathrm{rocC}\mathbb{C}\mathrm{hs}\mathbb{L}\mathrm{ist}\lceil ProcBPrefix^*\rfloor$ `‘};’`

```
Alternative alt ‘= ’new Alternative‘(’in‘);’ {
switch‘(’alt‘.’select‘()){’
      ℂℎ𝕠𝕚𝕔𝕖𝕤⌈ProBPrefix*⌋
‘}’ }[NoGUI]
| { switch‘(’choiceCtrl‘(’in‘,’null‘)){’
      ℂℎ𝕠𝕚𝕔𝕖⌈ProcBPrefix*⌋
‘}’ }[GUI]
```

CSP: $ProcBChoice \vdash ProcBPrefix^{+'\Box'}$

□

Rule 3.6.10.3. $\mathbb{Choice}\lceil ProcBPrefix^* \rfloor \Rightarrow$

```
case 𝕀𝕟𝕥𝕖𝕘𝕖𝕣 ‘:’
      ℂℎ𝕠𝕚𝕔𝕖ℝ𝕥𝕟𝔼𝕧⌈ProcBPrefix⌋
      | ℂℎ𝕠𝕚𝕔𝕖ℝ𝕥𝕟𝔼𝕧𝕍𝕒𝕝𝕦𝕖⌈ProcBPrefix⌋
      | ℙ𝕣𝕠𝕔𝔼⌈ProcBPrefix⌋
      break‘;’
‘}’
```

CSP: $ProcBPrefix \vdash ChCall \rightarrow ProcB$
Supplement: When *ChCall* is a B+CSP event without output data, choose $\mathbb{ChoiceRtnEv}$
Supplement: When *ChCall* is a B+CSP event with output data, choose $\mathbb{ChoiceRtnEvValue}$
Supplement: When *ChCall* is a CSP communication channel, choose $\mathbb{ProcE}$
Supplement: The rule $\mathbb{Integer}$ provides various values for all different repeatable instances of *ProcBPrefix*. It starts with 0, increasing by 1 each time.

□

Rule 3.6.10.3.1. $\mathbb{ChoiceRtnEv}\lceil ProcBPrefix \rfloor \Rightarrow \mathbb{ProcE}\lceil ProcB \rfloor$
CSP: $ProcBPrefix \vdash Ev \rightarrow ProcB$

□

Rule 3.6.10.3.2. $\mathbb{ChoiceRtnEvValue}\lceil ProcBPrefix \rfloor \Rightarrow$

```
{ ‘{’
      Vector‘<’Object‘>’ rtnVec
            ‘=’ ℂℎ𝕆𝕓𝕛ℕ𝕒𝕞𝕖⌈ChName⌋ ‘.’output‘_’return‘();’
      𝔼𝕧𝕖𝕟𝕥ℝ𝕖𝕥𝕦𝕣𝕟𝕍𝕒𝕝𝕦𝕖𝕤⌈OutVars⌋
‘}’ }[NoGUI]
| { ‘{’
      Vector‘<’Object‘>’ rtnVec
            ‘=’ choiceRtn‘(’ ℂℎ𝕆𝕓𝕛ℕ𝕒𝕞𝕖⌈ChName⌋ ‘);’
```

$\mathbb{E}\mathrm{vent}\mathbb{R}\mathrm{eturn}\mathbb{V}\mathrm{alues}\lceil OutVars\rfloor$

‘}’ $\}_{[GUI]}$

CSP: $ProcBPrefix \vdash Ev\text{'?'}OutVars \mid Ev\text{'!'}CSPExpr^{*}\text{'?'}OutVars \rightarrow ProcB$ □

Rule 3.6.11. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{R}\mathrm{epl}\mathbb{C}\mathrm{hoice}\lceil ProcBReplChoice\rfloor \Rightarrow$

$\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{C}\mathrm{hoice}\lceil ProcBChoice\rfloor$ □

Rule 3.6.12. $\mathbb{P}\mathrm{roc}\mathbb{E}\mathbb{C}\mathrm{all}\lceil ProcBCall\rfloor \Rightarrow$

`nextProcCtrl‘(’` $\mathbb{C}\mathrm{all}\mathbb{P}\mathrm{roc}\mathbb{E}\mathrm{xist}(\mathbb{P}\mathrm{roB}\mathbb{C}\mathrm{all})$ `‘);’` □

Rule 4. $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lasses}\lceil Evs\rfloor \Rightarrow \mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\lceil Ev\rfloor^{*}$

Supplement: $Evs \vdash Ev^{*}$

□

Rule 4.1. $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\lceil Ev\rfloor \Rightarrow$

$\mathbb{P}\mathrm{ackage}\mathbb{D}\mathrm{ef}$

$\mathbb{C}\mathrm{lass}\mathbb{H}\mathrm{eader}$

`public class` $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil Ev\rfloor$ `extends` $\mathbb{E}\mathrm{vent}\mathbb{T}\mathrm{ype}\lceil Ev\rfloor$ ‘{’

$\mathbb{M}\mathrm{ach}\mathbb{N}\mathrm{ame}$`‘_’machine var‘;’`

$\mathbb{E}\mathrm{vent}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil BOpVars\rfloor$

$\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{onstructor}\lceil Ev\rfloor$

$\{\mathbb{E}\mathrm{vent}\mathbb{I}\mathrm{nput}\mathbb{M}\mathrm{ethod}\lceil BInVars\rfloor\}$

$\{\mathbb{E}\mathrm{vent}\mathbb{O}\mathrm{utput}\mathbb{M}\mathrm{ethod}\lceil BOutVars\rfloor\}$

$\mathbb{E}\mathrm{vent}\mathbb{R}\mathrm{un}\lceil BOpBody\rfloor$

$\{\mathbb{E}\mathrm{vent}\mathbb{P}\mathrm{recondition}\lceil BOpBody\rfloor\}$

‘}’

B: $Ev \vdash BOutVars \leftarrow EvName\text{'}(\text{'}BInVars\text{'})\text{'}BOpBody$

Supplement: $BOpVars \vdash BOpInVarsBOpOutVarsy$

□

Rule 4.2. $\mathbb{E}\mathrm{vent}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil BOpVars\rfloor \Rightarrow$

$\{\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\lceil BOpVar\rfloor\mathbb{P}\mathrm{rint}\lceil BOpVar\rfloor\}^{+\text{'};\text{'}}$

B: $BOpVars \vdash BOpVar^{+\text{'},\text{'}}$

□

Rule 4.3. $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{onstructor}\lceil Ev\rfloor \Rightarrow$

`public` $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil Ev\rfloor$ ‘(’

$\mathbb{M}\mathrm{ach}\mathbb{N}\mathrm{ame}$ `‘_’machine var‘,’Block block‘){’`

```
        this‘.’var ‘=’ var‘;’
        this‘.’block ‘=’ block‘;’
        this‘.’setChName‘(’this‘.’getClass‘().’getSimpleName‘());’
    ‘}’
```
□

Rule 4.4. $\mathbb{E}\mathrm{ventInputMethod}\lceil BInVars\rfloor \Rightarrow$

```
protected synchronized void assign‘_’input‘(’
  Vector‘<’Object‘>’ inputVec‘){’
```
$\mathbb{A}\mathrm{ssignInVar}\lceil BInVar\rfloor^*$
```
‘}’
```

B: $BInVars \vdash BInVar^{+','}$

□

Rule 4.4.1. $\mathbb{A}\mathrm{ssignInVar}\lceil Var\rfloor \Rightarrow$

$\mathbb{A}\mathrm{ssignInVarInt}\lceil Var\rfloor$
| $\mathbb{A}\mathrm{ssignInVar\$et}\lceil Var\rfloor$
| $\mathbb{A}\mathrm{ssignInVarP\$et}\lceil Var\rfloor$
| $\mathbb{A}\mathrm{ssignInVar\mathbb{O}bj}\lceil Var\rfloor$

Supplement: When Var is an Integer, choose $\mathbb{A}\mathrm{ssignInVarInt}$
Supplement: When Var is an element of a B set, choose $\mathbb{A}\mathrm{ssignInVar\$et}$
Supplement: When Var is a set, choose $\mathbb{A}\mathrm{ssignInVarP\$et}$
Supplement: Otherwise, choose $\mathbb{A}\mathrm{ssignIn\mathbb{O}bj}$

□

Rule 4.4.2. $\mathbb{A}\mathrm{ssignInVarInt}\lceil Var\rfloor \Rightarrow$

$\mathbb{P}\mathrm{rint}\lceil Var\rfloor$
```
    ‘=’ ‘(’Integer‘)(’inputVec‘.’elementAt‘(’ Integer ‘);’
```
Supplement: $Var \in Ingeter$

□

Rule 4.4.3. $\mathbb{A}\mathrm{ssignInVar\$et}\lceil Var\rfloor \Rightarrow$

```
if‘(’inputVec‘.’elementAt‘(’ Integer ‘)’ instanceof String‘){’
    Print⌈Var⌋ ‘=’ returnSet‘(’
        BType⌈Var⌋ ‘.’class‘,’
        ‘(’String‘)’inputVec‘.’elementAt‘(’Integer ‘)’
    ‘);’
}else{
    Print⌈Var⌋ ‘=’ ‘(’ BType⌈Var⌋ ‘)(’
```

```
                inputVec‘.’elementAt‘(’ Integer ‘)’
            ‘);’
        ‘}’
```

Supplement: $Var \in Set$

□

Rule 4.4.4. $\mathbb{A}\mathrm{ssignIn}\mathbb{V}\mathrm{ar}\mathbb{P}\mathbb{S}\mathrm{et}\lceil Var\rfloor \Rightarrow$

```
if‘(’inputVec‘.’elementAt‘(’ Integer ‘)’ instanceof String‘){’
    Print⌈Var⌋ ‘=’ returnSets‘(’
        BType⌈Var⌋ ‘.’class‘,’
        ‘(’String‘)’inputVec‘.’elementAt‘(’Integer ‘)’
    ‘);’
}else{
    Print⌈Var⌋ ‘=’ ‘(’EnumSet‘<’ BType⌈Var⌋ ‘>)(’
        inputVec‘.’elementAt‘(’ Integer ‘)’
    ‘);’
‘}’
```

Supplement: $Var \in' POW'BSet$

□

Rule 4.4.5. $\mathbb{A}\mathrm{ssignIn}\mathbb{V}\mathrm{ar}\mathbb{O}\mathrm{bj}\lceil Var\rfloor \Rightarrow$

```
Print⌈Var⌋ ‘=’ inputVec‘.’elementAt‘(’ Integer ‘);’
```

□

Rule 4.5. $\mathbb{E}\mathrm{vent}\mathbb{O}\mathrm{utputMethod}\lceil BOutVars\rfloor \Rightarrow$

```
EventOutputMethodWithoutInput⌈BOutVars⌋
| EventOutputMethodWithInput⌈BOutVars⌋
```

B: $BInVars \vdash BInVar^{+','}$

□

Rule 4.5.1. $\mathbb{E}\mathrm{vent}\mathbb{O}\mathrm{utputMethod}\mathbb{W}\mathrm{ithout}\mathbb{I}\mathrm{nput}\lceil BOutVars\rfloor \Rightarrow$

```
protected synchronized void make‘_’output‘(){’
    MakeOutVar⌈BOutVar⌋*
‘}’
```

B: $BOutVars \vdash BOutVar^{+','}$

□

Rule 4.5.2. $\mathbb{E}\mathrm{vent}\mathbb{O}\mathrm{utputMethod}\mathbb{W}\mathrm{ith}\mathbb{I}\mathrm{nput}\lceil BOutVars\rfloor \Rightarrow$

```
protected synchronized void make‘_’output‘(’int indexInt‘){’
    MakeOutVarWithIn⌈BOutVar⌋*
‘}’
```

B: $BOutVars \vdash BOutVar^{+','}$

□

Rule 4.5.3. $\mathbb{M}\mathrm{ake}\mathbb{O}\mathrm{ut}\mathbb{V}\mathrm{ar}\lceil Var\rfloor \Rightarrow$

```
out‘_’element‘(’ Integer ‘,’ Print⌈Var⌋ ‘);’
```

□

Rule 4.5.4. $\mathbb{M}\mathrm{ake}\mathbb{O}\mathrm{ut}\mathbb{V}\mathrm{ar}\mathbb{W}\mathrm{ith}\mathbb{I}\mathrm{n}\lceil Var\rfloor \Rightarrow$

```
out‘_’element‘(’indexInt‘,’ Integer ‘,’ Print⌈Var⌋ ‘);’
```

□

Rule 4.6. $\mathbb{E}\mathrm{vent}\mathbb{R}\mathrm{un}\lceil BOpBody\rfloor \Rightarrow$

```
protected synchronized void run‘(){’
    BeforeStateVars⌈StateVars⌋
    BSubstitution⌈BSub⌋
    var‘.’check‘();’
    { varsPanelsStore‘.’getInstance‘().’getPanelInstance‘(’
       ‘"’ MachName ‘").’refresh‘();’ }[GUI]
‘}’
```

B: $BOpBody \vdash BSub$

Supplement: $Vars$ consists of all the state variables

□

Rule 4.6.1. $\mathbb{B}\mathrm{efore}\mathbb{S}\mathrm{tate}\mathbb{V}\mathrm{ars}\lceil Vars\rfloor \Rightarrow$

$\mathbb{B}\mathrm{efore}\mathbb{S}\mathrm{tate}\mathbb{V}\mathrm{ar}\lceil Var\rfloor^*$

Supplement: $Vars \vdash Var^*$

□

Rule 4.6.2. $\mathbb{B}\mathrm{efore}\mathbb{S}\mathrm{tate}\mathbb{V}\mathrm{ar}\lceil Var\rfloor \Rightarrow$

```
BType⌈Var⌋Print⌈Var⌋ ‘=’
BeforeStateVarP$et⌈Var⌋
| BeforeStateVarArray⌈Var⌋
| BeforeStateVarRelation⌈Var⌋
| BeforeStateVarAssignObj⌈Var⌋
‘;’
```

Supplement: If Var is a B set, $\mathbb{B}\mathrm{efore}\mathbb{S}\mathrm{tate}\mathbb{V}\mathrm{ar}\mathbb{P}\mathrm{Set}$

Supplement: If Var is an array, $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{arArray}$
Supplement: If Var is a relation, $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{arRelation}$
Supplement: Otherwise, $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{ar}\mathbb{O}\text{bj}$

□

Rule 4.6.2.1. $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{ar}\mathbb{P}\$\text{et}\lceil Var\rfloor \Rightarrow$

```
EnumSet‘.’copyOf‘(’var‘.’ ℙrint⌈Var⌋ ‘)’
```

B: $Var \in P(Set)$

□

Rule 4.6.2.2. $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{ar}\mathbb{A}\text{rray}\lceil Var\rfloor \Rightarrow$

```
new 𝕋ype⌈Var⌋‘[’var‘.’ℙrint⌈Var⌋‘.’length‘];’
System‘.’arraycopy‘(’
   var‘.’ℙrint⌈Var⌋‘,’0‘,’ℙrint⌈Var⌋‘,’0‘,’var‘.’
   ℙrint⌈Var⌋‘.’length
‘);’
```

Supplement: Var is an array

□

Rule 4.6.2.3. $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{ar}\mathbb{R}\text{elation}\lceil\rfloor \Rightarrow$

```
var‘.’ ℙrint⌈Var⌋ ‘.’clone‘();’
```

Supplement: Var is a relation type

□

Rule 4.6.2.4. $\mathbb{B}\text{efore}\$\text{tate}\mathbb{V}\text{ar}\mathbb{A}\text{ssign}\mathbb{O}\text{bj}\lceil\rfloor \Rightarrow$

```
var‘.’ ℙrint⌈Var⌋ ‘;’
```

□

Rule 4.7. $\mathbb{E}\text{vent}\mathbb{P}\text{recondition}\lceil BSubPreCdtn\rfloor \Rightarrow$

```
protected synchronized boolean conditionCheck‘(){’
     return 𝔹ℂondition⌈BCdtn⌋ ‘;’
‘}’
```

B: $BSubPreCdtn \vdash' PRE'BCdtn'THEN'BSub'END'$

□

Rule 5.1. $\mathbb{B}\$\text{ubstitution}\lceil BSub\rfloor \Rightarrow$

$\mathbb{B}\$\text{ubstitution}\mathbb{P}\text{recondition}\lceil BSubPrecdtn\rfloor$
$|\ \mathbb{B}\$\text{ubstitution}\mathbb{B}\text{egin}\lceil BSubBegin\rfloor$

$$
\begin{array}{l}
\mid \mathbb{B}\$\mathrm{ubstitution}\mathbb{V}\mathrm{ar}\lceil BSubVar\rfloor \\
\mid \mathbb{B}\$\mathrm{ubstitution}\mathbb{P}\mathrm{arall}\mathbb{e}\mathrm{l}\lceil BSubPar\rfloor \\
\mid \mathbb{B}\$\mathrm{ubstitution}\mathbb{B}\mathrm{e}\mathbb{E}\mathrm{qual}\lceil BSubBeq\rfloor \\
\mid \mathbb{B}\$\mathrm{ubstitution}\mathbb{I}\mathrm{f}\lceil BSubIf\rfloor \\
\mid \mathbb{B}\$\mathrm{ubstitution}\mathbb{B}\mathrm{e}\mathbb{E}\mathrm{qual}\mathbb{F}\mathrm{unc}\lceil BSubBeqFunc\rfloor \\
\mid \mathbb{B}\$\mathrm{ubstitution}\mathbb{A}\mathrm{ny}\lceil BSubAny\rfloor
\end{array}
$$

B: $BSub \vdash$

$$
\begin{array}{l}
BSubPrecdtn \\
\mid BSubBegin \\
\mid BSubVar \\
\mid BSubPar \\
\mid BSubBeq \\
\mid BSubIf \\
\mid BSubBeqFunc
\end{array}
$$

□

Rule 5.1.1. $\mathbb{B}\$\mathrm{ubstitution}\mathbb{P}\mathrm{recondition}\lceil BSubPrecdtn\rfloor \Rightarrow$
$\mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor$

B: $BSubPreCdtn \vdash' PRE' BCdtn' THEN' BSub' END'$

□

Rule 5.1.2. $\mathbb{B}\$\mathrm{ubstitution}\mathbb{B}\mathrm{egin}\lceil BSubBegin\rfloor \Rightarrow \mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor$

B: $BSubPreCdtn \vdash' BEGIN' BSub' END'$

□

Rule 5.1.3. $\mathbb{B}\$\mathrm{ubstitution}\mathbb{V}\mathrm{ar}\lceil BSubVar\rfloor \Rightarrow$
$\mathbb{E}\mathrm{vent}\mathbb{V}\mathrm{ars}\mathbb{D}\mathrm{ecl}\lceil Vars\rfloor$
$\mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor$

B: $BSubVar \vdash' VAR' Vars' IN' BSub' END'$

□

Rule 5.1.4. $\mathbb{B}\$\mathrm{ubstitution}\mathbb{P}\mathrm{arall}\mathbb{e}\mathrm{l}\lceil BSubPar\rfloor \Rightarrow$
$\mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor$
$\mathbb{B}\$\mathrm{ubstitution}\lceil BSub\rfloor$

B: $BSubPrePar \vdash BSub \parallel BSub$

□

Rule 5.1.5. $\mathbb{B}\$ubstitution\mathbb{B}eEqual\lceil BSubBeq \rfloor \Rightarrow$

$\mathbb{B}\mathbb{B}eEqualArray\lceil BSubBeq \rfloor$
$\mid \mathbb{B}\mathbb{B}eEqual\$etUnion\lceil BSubBeq \rfloor$
$\mid \mathbb{B}\mathbb{B}eEqual\$etMinus\lceil BSubBeq \rfloor$
$\mid \mathbb{B}\mathbb{B}eEqual\mathbb{O}ther\lceil BSubBeq \rfloor$

B: $BSubBeq \vdash BVar' =' BExpression$
Supplement: If $BVar$ is an array, $\mathbb{B}efore\$tate\mathbb{V}arArray$
Supplement: If $BExpression$ is a B Set union, $\mathbb{B}\mathbb{B}eEqual\$etUnion$
Supplement: If $BExpression$ is a B Set minus, $\mathbb{B}\mathbb{B}Equal\$etMinus$
Supplement: Otherwise, $\mathbb{B}\mathbb{B}eEqual\mathbb{O}ther$

□

Rule 5.1.6. $\mathbb{B}\$ubstitutionIf\lceil BSubIf \rfloor \Rightarrow$

`if‘(’` $\mathbb{B}\mathbb{C}ondition\lceil BCdtn \rfloor$ `‘){’`
$\mathbb{B}\$ubstitution\lceil BSub \rfloor$
`‘}’`
$\mathbb{B}\$ubstitution\mathbb{E}lseIf\lceil BCdtn, BSub \rfloor^*$
$\mathbb{B}\$ubstitution\mathbb{E}lse\lceil BSub \rfloor$

B: $BSubIf \vdash$

'IF' BCdtn 'THEN' BSub
{ 'ELSIF' BCdtn 'THEN' BSub }*
{ 'ELSE' BSub }
'END'

□

Rule 5.1.7. $\mathbb{B}\$ubstitution\mathbb{B}eEqual\mathbb{F}unc\lceil BSubBeqFunc \rfloor \Rightarrow$

$\mathbb{B}\$ubstitution\mathbb{B}eEqualArray\lceil BSubBeqFunc \rfloor \mid \mathbb{B}\$ubstitution\mathbb{B}eEqual\mathbb{F}unctio$

B: $BSubBeq \vdash BVar' =' BExpression$
Supplement: If $BVar$ is an array, $\mathbb{B}\$ubstitution\mathbb{B}eEqualArray$
Supplement: If $BVar$ is a function, $\mathbb{B}\$ubstitution\mathbb{B}eEqual\mathbb{F}unction$

□

Rule 5.1.8. $\mathbb{B}\$ubstitution\mathbb{A}ny\lceil BSubBeqAny \rfloor \Rightarrow$

$\mathbb{E}vent\mathbb{V}ars\mathbb{D}ecl\lceil Vars \rfloor$
$\mathbb{B}\$ubstitution\lceil BSub \rfloor$

B: $BSubVar \vdash' ANY' Vars' WHERE' Predicates' IN' BSub' END'$

□

Rule 5.2. $\mathbb{B}\mathbb{E}\mathrm{xpression}\lceil BExpr \rfloor$ □

Rule 5.3. $\mathbb{B}\mathbb{C}\mathrm{ondition}\lceil BCdtn \rfloor \Rightarrow$ □

Rule 5.4. $\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\lceil Var \rfloor \Rightarrow$
$\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{I}\mathrm{nteger}\lceil Var \rfloor$
$\mid \mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{S}\mathrm{et}\lceil Var \rfloor$
$\mid \mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{P}\mathbb{S}\mathrm{et}\lceil Var \rfloor$

□

Rule 5.4.1. $\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{I}\mathrm{nteger}\lceil Var \rfloor \Rightarrow$ `Integer`

B: $Var \in INT \mid NAT$ □

Rule 5.4.2. $\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{S}\mathrm{et}\lceil Var \rfloor \Rightarrow \mathbb{P}\mathrm{rint}\lceil BSetName \rfloor$

B: $Var \in BSet$

□

Rule 5.4.3. $\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{P}\mathbb{S}\mathrm{et}\lceil Var \rfloor \Rightarrow$
`EnumSet'<'` $\mathbb{P}\mathrm{rint}\lceil BSetName \rfloor$ `'>'`

B: $Var \in' POW'BSet$

□

Rule 5.4.4. $\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{R}\mathrm{elation}\lceil Var \rfloor \Rightarrow$
`Relation'<'` $\mathbb{P}\mathrm{rint}\lceil BType \rfloor$ `','` $\mathbb{P}\mathrm{rint}\lceil BType \rfloor$ `'>'`

B: $Var \in BType' + - >' BType$

□

Rule 5.4.5. $\mathbb{B}\mathbb{V}\mathrm{ar}\mathbb{T}\mathrm{ype}\mathbb{A}\mathrm{rray}\lceil Var \rfloor \Rightarrow$
$\mathbb{P}\mathrm{rint}\lceil BType \rfloor$ `'[]'`

B: $Var \in NAT' - - >' BType$

□

Rule 5.5. $\mathbb{B}\mathbb{E}\mathrm{xpr}\mathbb{T}\mathrm{ype}\lceil BExpr \rfloor$ □

Rule 6.1. $\mathbb{CSPExpression}\lceil CSPExpr \rfloor$ □

Rule 6.2. $\mathbb{CSPCondition}\lceil CSPCdtn \rfloor$ □

Rule 6.3. $\mathbb{CSPExprList}\lceil CSPExpr^* \rfloor$ □

Rule 7.2. $\mathbb{ChannelType}\lceil Ch \rfloor \Rightarrow$

$$\mathbb{EventType}\lceil Ev \rfloor \mid \mathbb{CChType}\lceil CCh \rfloor$$

B+CSP: $Ch \vdash Ev \mid CCh$

□

Rule 7.2.1. $\mathbb{EventType}\lceil Ev \rfloor \Rightarrow$

`CChannel` | `InCChannel` | `OutCChannel` | `OutInCChannel`

Supplement: *If the combined event Ev has no data flow between B and CSP, returns* `CChannel`. (**CSP**: *ch* **B**: *op*)

Supplement: *If the combined event Ev only has input data, from the CSP channel to the B operation, returns* `InCChannel`. (**CSP**: $ch`!'InVars$ **B**: $op`('InVars`)'$)

Supplement: *If the combined event Ev only has output data, from the B operation to the CSP channel, returns* `OutCChannel`. (**CSP**: $ch`?'OutVars$ **B**: $OutVars` <-- 'op$)

Supplement: *If combined event Ev has both input data, from CSP to B, and output data, from B to CSP, returns* `OutInCChannel`. (**CSP**: $ch'!'InVars'?'OutVars$ **B**: $OutVars' <-- 'op'('InVars')'$)

□

Rule 7.2.2. $\mathbb{CChType}\lceil CCh \rfloor \Rightarrow$

`ChannelInput` | `ChannelOutput` | $\mathbb{ChClassName}\lceil CCh \rfloor$

Supplement: *If the process only read from channel CCh, returns* `ChannelInput`.

Supplement: *If the process only output to channel CCh, returns* `ChannelOutput`.

Supplement: *If the process and its sub-processes both read and write to channel CCh, returns the channel class name using rule* $\mathbb{CChClassName}\lceil \mathrm{CCh} \rfloor$.

□

Rule 7.3. $\mathbb{ChClassName}\lceil Ch \rfloor \Rightarrow$

$$\mathbb{EventClassName}\lceil Ev \rfloor$$
$$\mid \mathbb{CChClassName}\lceil CCh \rfloor$$

B+CSP: $Ch \vdash Ev \mid CCh$

□

Rule 7.3.1. $\mathbb{E}\mathrm{vent}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor \Rightarrow \mathbb{P}\mathrm{rint}\lceil EvName \rfloor$ `'_chclass'`

B: $BEv \vdash OutVars' <\!\!- \!\!- 'EvName'('InVars')'$

□

Rule 7.3.2. $\mathbb{C}\mathbb{C}\mathrm{h}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil CCh \rfloor \Rightarrow$

`One2OneChannel` | `One2AnyChannel` | `Any2OneChannel`

| `Any2AnyChannel` | $\mathbb{C}\mathrm{h}\mathbb{E}\mathrm{nd}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil CCh \rfloor$

Supplement: *One output end, one input end:* `One2OneChannel`.
Supplement: *One output end, multiple input ends:* `One2AnyChannel`.
Supplement: *Multiple output ends, one input end:* `Any2OneChannel`.
Supplement: *Multiple output ends, multiple input ends:* `Any2AnyChannel`.

□

Rule 7.3.3. $\mathbb{C}\mathrm{h}\mathbb{E}\mathrm{nd}\mathbb{C}\mathrm{lass}\mathbb{N}\mathrm{ame}\lceil CCh \rfloor \Rightarrow$

{ `Ext2OneChannel` | One2ExtChannel

| Any2OneChannel | Any2ExtChannel }$_{[NoGUI]}$

{ `Ext2OneGUIChannel` | One2ExtGUIChannel

| Ext2AnyGUIChannel | Any2ExtGUIChannel }$_{[GUI]}$ □

Rule 7.4. $\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ch \rfloor \Rightarrow$
$\mathbb{E}\mathrm{vent}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor$
| $\mathbb{C}\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil CCh \rfloor$

B+CSP: $Ch \vdash Ev \mid CCh$

□

Rule 7.4.1. $\mathbb{E}\mathrm{vent}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil Ev \rfloor \Rightarrow \mathbb{P}\mathrm{rint}\lceil EvName \rfloor$ `'_ch'`

B: $BEv \vdash OutVars' <\!\!- \!\!- 'EvName'('InVars')'$

□

Rule 7.4.2. $\mathbb{C}\mathbb{C}\mathrm{h}\mathbb{O}\mathrm{bj}\mathbb{N}\mathrm{ame}\lceil CCh \rfloor \Rightarrow \mathbb{P}\mathrm{rint}\lceil CChName \rfloor$

CSP: $BCCh \vdash CChName\{'?'InVars\}\{'!'OutVars\}$ □

Appendix B

Java Classes

B.1 Runtime Assertion Checking

B.2 Dining Philosophers

B.2.1 PHIL_procclass.java

```
package dps;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;
import java.util.*;

public class PHIL_procclass extends BGProcess{
    private InCChannel sits_ch;
    private InCChannel picks_ch;
    private InCChannel eats_ch;
    private InCChannel putsdown_ch;
    private InCChannel getsup_ch;

    Integer proc_index_a;

    public PHIL_procclass(InCChannel sits_ch,InCChannel picks_ch,
            InCChannel eats_ch,InCChannel putsdown_ch,
            InCChannel getsup_ch,Integer proc_index_a){
        super();
        this.sits_ch = sits_ch;
        this.picks_ch = picks_ch;
        this.eats_ch = eats_ch;
        this.putsdown_ch = putsdown_ch;
```

```
        this.getsup_ch = getsup_ch;
        this.proc_index_a = proc_index_a;
    }


    public void run(){
        this.createGUIProc();
        start();
        channelCall(sits_ch,inputVector(new Object[]{proc_index_a}));
        Vector<Vector<Object>> choiceVec = new Vector<Vector<Object>>();
        choiceVec.addElement(inputVector(
            new Object[]{proc_index_a,proc_index_a}));
        choiceVec.addElement(inputVector(
            new Object[]{proc_index_a,(proc_index_a + 1) % 5}));
        BCSPGuard[] in = {picks_ch,picks_ch};
        switch(choiceCtrl(in, choiceVec)){
            case 0 :
                choiceRtn(picks_ch);
                channelCall(picks_ch,inputVector(
                    new Object[]{proc_index_a,(proc_index_a + 1) % 5}));
                channelCall(eats_ch,inputVector(
                    new Object[]{proc_index_a}));
                channelCall(putsdown_ch,inputVector(
                    new Object[]{proc_index_a,(proc_index_a + 1) % 5}));
                channelCall(putsdown_ch,inputVector(
                    new Object[]{proc_index_a,proc_index_a}));
                channelCall(getsup_ch,inputVector(
                    new Object[]{proc_index_a}));
                nextProcCtrl(new PHIL_procclass(
                    sits_ch,picks_ch,eats_ch,putsdown_ch,getsup_ch,proc_index_a));
                break;
            case 1 :
                choiceRtn(picks_ch);
                channelCall(picks_ch,inputVector(
                    new Object[]{proc_index_a,proc_index_a}));
                channelCall(eats_ch,inputVector(
                    new Object[]{proc_index_a}));
                channelCall(putsdown_ch,inputVector(
                    new Object[]{proc_index_a,proc_index_a}));
                channelCall(putsdown_ch,inputVector(
                    new Object[]{proc_index_a,(proc_index_a + 1) % 5}));
                channelCall(getsup_ch,inputVector(
                    new Object[]{proc_index_a}));
                nextProcCtrl(new PHIL_procclass(
                    sits_ch,picks_ch,eats_ch,putsdown_ch,getsup_ch,proc_index_a));
                break;
        }
```

```
        end();
    }
}
```

B.2.2 FORK_procclass.java

```
package dps;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;
import java.util.*;

public class FORK_procclass extends BGProcess{
    private InCChannel picks_ch;
    private InCChannel putsdown_ch;

    Integer proc_index_a;

    public FORK_procclass(InCChannel picks_ch,
            InCChannel putsdown_ch,Integer proc_index_a){
        super();
        this.picks_ch = picks_ch;
        this.putsdown_ch = putsdown_ch;
        this.proc_index_a = proc_index_a;
    }


    public void run(){
        this.createGUIProc();
        start();
        Vector<Vector<Object>> choiceVec = new Vector<Vector<Object>>();
        choiceVec.addElement(inputVector(
            new Object[]{proc_index_a,proc_index_a}));
        choiceVec.addElement(inputVector(
            new Object[]{(proc_index_a + 4) % 5,proc_index_a}));
        BCSPGuard[] in = {picks_ch,picks_ch};
        switch(choiceCtrl(in, choiceVec)){
            case 0 :
                choiceRtn(picks_ch);
                channelCall(putsdown_ch,
                    inputVector(new Object[]{proc_index_a,proc_index_a}));
                nextProcCtrl(
                    new FORK_procclass(picks_ch,putsdown_ch,proc_index_a));
                break;
```

```
            case 1 :
                choiceRtn(picks_ch);
                channelCall(putsdown_ch,
                    inputVector(new Object[]{(proc_index_a + 4) % 5,proc_index_a}));
                nextProcCtrl(
                    new FORK_procclass(picks_ch,putsdown_ch,proc_index_a));
                break;
        }
        end();
    }
}
```

B.2.3 picks_chclass.java

```
package dps;

import org.dsse.jcsprob.gui.*;
import org.dsse.jcsprob.lang.*;
import java.util.*;

public class picks_chclass extends InCChannel{
    diningphils_machine var;
    Integer pp;
    Integer ff;
    public picks_chclass(diningphils_machine var, BLock block){
        this.var = var;
        this.block = block;
        this.setChName(this.getClass().getSimpleName());
    }
    protected synchronized void assign_input(Vector<Object> inputVec){
        pp = (Integer)(inputVec.elementAt(0));
        ff = (Integer)(inputVec.elementAt(1));
    }
    protected synchronized void run(){
        try{
            var.pstate[pp] = 2;
            var.fstate[ff] = 1;

            varsPanelsStore.getInstance().getPanelInstance("diningphils").refresh();
        }catch(Exception e){
            System.out.println("Exception in
                Channel picks_chclass :> "+e.getMessage());
            System.exit(0);
        }
    }
```

```
    protected synchronized boolean conditionCheck(){
        return pp>=0 && pp<=4 && ff>=0 && ff<=4;
    }
}
```

B.2.4 eats_chclass.java

```
package dps;

import org.dsse.jcsprob.gui.*;
import org.dsse.jcsprob.lang.*;
import java.util.*;

public class eats_chclass extends InCChannel{
    diningphils_machine var;
    Integer pp;
    public eats_chclass(diningphils_machine var, BLock block){
        this.var = var;
        this.block = block;
        this.setChName(this.getClass().getSimpleName());
    }
    protected synchronized void assign_input(Vector<Object> inputVec){
        pp = (Integer)(inputVec.elementAt(0));
    }
    protected synchronized void run(){
        try{
            var.pstate[pp] = 3;
            var.count[pp] = var.count[pp] + 1;

            varsPanelsStore.getInstance().getPanelInstance("diningphils").refresh();
        }catch(Exception e){
            System.out.println("Exception in
                Channel eats_chclass :> "+e.getMessage());
            System.exit(0);
        }
    }
    protected synchronized boolean conditionCheck(){
        return pp>=0 && pp<=4;
    }
}
```

B.3 Wot, no chicken?

B.3.1 chicken_run.java

```
package wnck1;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;

public class chicken_run{
    public static void main(String[] args){
        try{
            GUIModelConfigStore.getInstance().addModelInstance("chicken");
            chicken_machine var = new chicken_machine("chicken");
            BLock block = new BLock();
            varsPanelsStore tmp = varsPanelsStore.getInstance();
            tmp.addPanelInstance(var);
            thinking_chclass thinking_ch = new thinking_chclass(var,block);
            getchicken_chclass getchicken_ch = new getchicken_chclass(var,block);
            eat_chclass eat_ch = new eat_chclass(var,block);
            cook_chclass cook_ch = new cook_chclass(var,block);
            put_chclass put_ch = new put_chclass(var,block);
            final One2OneChannel configure = new One2OneChannel();
            final One2OneChannel comm = new One2OneChannel();
            tmp.setCommCh(comm);

            new CSParallel(
                new BCSProcess[]{
                    new BCSProcess(){
                        public void run(){
                            GUIFrame frame = new GUIFrame(configure,"chicken");
                            frame.pack();
                            frame.setVisible(true);
                            frame.run();
                        }
                    },
                    new BCSProcess(){
                        public void run(){
                            RecordsFrame rcdFrame = RecordsFrame.getInstance();
                            rcdFrame.init(comm);
                            rcdFrame.pack();
                            rcdFrame.setVisible(true);
                            rcdFrame.run();
                        }
                    },
```

```
                    new chicken_procclass(
                        cook_ch,put_ch,getchicken_ch,eat_ch,thinking_ch,configure)
                }
            ).run();
        }catch(Exception e){
            System.out.println("Error: chicken_run main() :> "+e.getMessage());
            System.exit(0);
        }
    }
}
```

B.3.2 chicken_procclass.java

```
package wnck1;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;
import java.util.*;

public class chicken_procclass extends BGProcess{
    private CChannel cook_ch;
    private CChannel put_ch;
    private InCChannel getchicken_ch;
    private InCChannel eat_ch;
    private InCChannel thinking_ch;


    public chicken_procclass(CChannel cook_ch,CChannel put_ch,
            InCChannel getchicken_ch,InCChannel eat_ch,
            InCChannel thinking_ch,ChannelOutput conf){
        super(conf);
        this.cook_ch = cook_ch;
        this.put_ch = put_ch;
        this.getchicken_ch = getchicken_ch;
        this.eat_ch = eat_ch;
        this.thinking_ch = thinking_ch;
    }


    public void run(){
        this.createGUIProc();
        start();
        parallelCtrl(
            new BGProcess[]{
                new Chef_procclass(cook_ch,put_ch),
```

```
                new XPhil_procclass(getchicken_ch,eat_ch),
                new PHILS_procclass(thinking_ch,getchicken_ch,eat_ch)
            }
        );
        end();
    }
}
```

B.3.3 Phil_procclass.java

```
package wnck1;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;
import java.util.*;

public class Phil_procclass extends BGProcess{
    private InCChannel thinking_ch;
    private InCChannel getchicken_ch;
    private InCChannel eat_ch;

    Object proc_index_a;

    public Phil_procclass(InCChannel thinking_ch,InCChannel getchicken_ch,
            InCChannel eat_ch,Object proc_index_a){
        super();
        this.thinking_ch = thinking_ch;
        this.getchicken_ch = getchicken_ch;
        this.eat_ch = eat_ch;
        this.proc_index_a = proc_index_a;
        argsVec.add(proc_index_a);
    }

    public void run(){
        this.createGUIProc();
        start();
        channelCall(thinking_ch,
            inputVector(new Object[]{(Integer)proc_index_a}));
        channelCall(getchicken_ch,
            inputVector(new Object[]{(Integer)proc_index_a}));
        channelCall(eat_ch,
            inputVector(new Object[]{(Integer)proc_index_a}));
        nextProcCtrl(new Phil_procclass(
            thinking_ch,getchicken_ch,eat_ch,proc_index_a));
```

```
        end();
    }
}
```

B.3.4 getchicken_chclass.java

```
package wnck1;

import org.dsse.jcsprob.gui.*;
import org.dsse.jcsprob.btypes.*;
import org.dsse.jcsprob.lang.*;
import java.util.*;

public class getchicken_chclass extends InCChannel{
    chicken_machine var;
    Integer pp;
    public getchicken_chclass(chicken_machine var, BLock block){
        this.var = var;
        this.block = block;
        this.setChName(this.getClass().getSimpleName());
    }
    protected synchronized void assign_input(Vector<Object> inputVec){
        pp = (Integer)(inputVec.elementAt(0));
    }
    protected synchronized void run(){
        try{
            Integer[] state = new Integer[var.state.length];
            System.arraycopy(var.state,0,state,0,var.state.length);
            Integer chef = var.chef;
            Integer canteen = var.canteen;
            var.canteen = canteen - 1;

            var.check();
            varsPanelsStore.getInstance().getPanelInstance("chicken").refresh();
        }catch(Exception e){
            System.out.println("Exception in Channel
                getchicken_chclass :> "+e.getMessage());
            System.exit(0);
        }
    }
    protected synchronized boolean conditionCheck(){
        return pp>=0 && pp<=4 && var.canteen > 0;
    }
}
```

B.4 The Odd-Even Example

B.4.1 Even_run.java

```
package oddeven.Even;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;

public class Even_run{
    public static void main(String[] args){
        try{
            GUIModelConfigStore.getInstance().addModelInstance("Even");
            Even_machine var = new Even_machine("Even");
            BLock block = new BLock();
            varsPanelsStore tmp = varsPanelsStore.getInstance();
            tmp.addPanelInstance(var);
            evenput_chclass evenput_ch = new evenput_chclass(var,block);
            evenget_chclass evenget_ch = new evenget_chclass(var,block);
            /* The following channel communicates with the environment */
            final Ext2OneGUIChannel oddpass_ch
                = new Ext2OneGUIChannel("oddpass_ch");
            /* The following channel communicates with the environment */
            final One2ExtGUIChannel evenpass_ch
                = new One2ExtGUIChannel("evenpass_ch");
            final One2OneChannel configure = new One2OneChannel();
            final One2OneChannel comm = new One2OneChannel();
            tmp.setCommCh(comm);

            new CSParallel(
                new BCSProcess[]{
                    new BCSProcess(){
                        public void run(){
                            GUIFrame frame = new GUIFrame(configure,"Even");
                            frame.pack();
                            frame.setVisible(true);
                            frame.run();
                        }
                    },
                    new BCSProcess(){
                        public void run(){
                            RecordsFrame rcdFrame = RecordsFrame.getInstance();
                            rcdFrame.init(comm);
                            rcdFrame.pack();
                            rcdFrame.setVisible(true);
```

```
                        rcdFrame.run();
                    }
                },
                new Even_procclass(
                    oddpass_ch,evenput_ch,evenget_ch,evenpass_ch,configure)
            }
        ).run();
    }catch(Exception e){
        System.out.println("Error: Even_run main() :> "+e.getMessage());
        System.exit(0);
    }
  }
}
```

B.4.2 Oddeven_run.java

```
package oddeven;

import org.dsse.jcsprob.lang.*;
import org.dsse.jcsprob.gui.*;
import jcsp.lang.*;
import oddeven.Even.*;
import oddeven.Odd.*;

public class Oddeven_run{
    public static void main(String[] args){
        try{
            GUIModelConfigStore.getInstance().addModelInstance("Odd");
            GUIModelConfigStore.getInstance().addModelInstance("Even");

            Even_machine var1 = new Even_machine("Even");
            Odd_machine var2 = new Odd_machine("Odd");
            BLock block1 = new BLock();
            BLock block2 = new BLock();
            varsPanelsStore tmp = varsPanelsStore.getInstance();
            tmp.addPanelInstance(var1);
            tmp.addPanelInstance(var2);
            evenput_chclass evenput_ch = new evenput_chclass(var1,block1);
            evenget_chclass evenget_ch = new evenget_chclass(var1,block1);
            oddput_chclass oddput_ch = new oddput_chclass(var2,block2);
            oddget_chclass oddget_ch = new oddget_chclass(var2,block2);
            final One2OneChannel oddpass_ch = new One2OneChannel();
            final One2OneChannel evenpass_ch = new One2OneChannel();

            final One2OneChannel configure1 = new One2OneChannel();
```

```
            final One2OneChannel configure2 = new One2OneChannel();
            final Any2OneChannel comm = new Any2OneChannel();
            tmp.setCommCh(comm);

            new CSParallel(
                new BCSProcess[]{
                    new BCSProcess(){
                        public void run(){
                            GUIFrame frame = new GUIFrame(configure1,"Even");
                            frame.pack();
                            frame.setVisible(true);
                            frame.run();
                        }
                    },
                    new Even_procclass(
                        oddpass_ch,evenput_ch,evenget_ch,evenpass_ch,configure1),
                    new BCSProcess(){
                        public void run(){
                            GUIFrame frame = new GUIFrame(configure2,"Odd");
                            frame.pack();
                            frame.setVisible(true);
                            frame.run();
                        }
                    },
                    new BCSProcess(){
                        public void run(){
                            RecordsFrame rcdFrame = RecordsFrame.getInstance();
                            rcdFrame.init(comm);
                            rcdFrame.pack();
                            rcdFrame.setVisible(true);
                            rcdFrame.run();
                        }
                    },
                    new Odd_procclass(
                        oddget_ch,oddpass_ch,evenpass_ch,oddput_ch,configure2)
                }
            ).run();
        }catch(Exception e){
            System.out.println("Error: Oddeven_run main() :> "+e.getMessage());
            System.exit(0);
        }
    }
}
```

Appendix C

Specifications

C.1 The Decomposed Wot-no-chicken model: Step 2

C.1.1 The CSP Specification

$MAIN = Chef\ [|\ \{put\}\ |]\ Canteen\ [|\ \{take\}\ |]\ PHILS\ ;;$
$PHILS = GPhils\ |||\ XPhil\ ;;$
$GPhils = |||X{:}0,1,2,3@Phil(X);;$
$Phil(X) = thinking.X \rightarrow take?Y \rightarrow eat.X.Y \rightarrow Phil(X);;$
$XPhil = take?Y \rightarrow eat.4.Y \rightarrow XPhil;;$
$Chef = cook \rightarrow put \rightarrow Chef\ ;;$
$Canteen = CProc\ |||\ PProc\ ;;$
$CProc = put \rightarrow CProc\ ;;$
$PProc = getchicken.X \rightarrow PProc\ ;;$

C.1.2 B Machine: Chef

MACHINE *Chef*
VARIABLES
    chef
INVARIANT
    $chef \in NAT$
INITIALISATION
    $chef := 0$
OPERATIONS
    cook =
        BEGIN
        $chef := chef + 4$
        END;
    $nn \leftarrow put1$ =
        BEGIN $chef := chef - 4 \parallel nn := 4$ ***END***

END

C.1.3 B Machine: Canteen

MACHINE *Canteen*
VARIABLES
canteen
INVARIANT
canteen ∈ *NAT*
INITIALISATION
canteen := 0
OPERATIONS
nn ← *getchicken(pp)* =
PRE *canteen > 0* ***THEN***
canteen := canteen - 1 ‖ *nn := 1*
END*;*
put2(pp) =
PRE *pp* ∈ *NAT* ∧ *pp > 0* ***THEN***
canteen := canteen + pp ***END***
END

C.1.4 B Machine: Phils

MACHINE *Phils*
VARIABLES
state
INVARIANT
state ∈ *(0..4)*→*NAT*
INITIALISATION
*state := (0..4) * {1}*
OPERATIONS
thinking(pp) =
PRE *pp*∈*0..4* ***THEN***
state(pp) := state(pp) - 1
END*;*
eat(pp,nn) =
PRE *pp*∈*0..4* ∧ *nn*∈*NAT* ∧ *nn*>*0* ***THEN***
state(pp) := state(pp) + nn
END
END

C.2 The Decomposed Wot-no-chicken model: Step 3

C.2.1 The CSP Specification

$MAIN = Chef\ [|\ \{put\}\ |]\ Canteen\ [|\ \{take\}\ |]\ PHILS\ ;;$
$PHILS = GPhils\ |||\ XPhil\ ;;$
$GPhils = |||X{:}0,1,2,3@Phil(X);;$
$Phil(X) = thinking \rightarrow take?Y \rightarrow eat.Y \rightarrow Phil(X);;$
$XPhil = take?Y \rightarrow eat.Y \rightarrow XPhil;;$
$Chef = cook \rightarrow put \rightarrow Chef\ ;;$
$Canteen = CProc\ |||\ PProc\ ;;$
$CProc = put \rightarrow CProc\ ;;$
$PProc = getchicken.X \rightarrow PProc\ ;;$

C.2.2 B Machine: Phil

MACHINE *Phil*
VARIABLES
state
INVARIANT
$state \in NAT$
INITIALISATION
state := 1
OPERATIONS
thinking =
BEGIN
state := state - 1
END*;*
eat(nn) =
PRE $nn \in NAT \wedge nn > 0$ ***THEN***
state := state + nn
END
END

C.2.3 B Machine: XPhil

MACHINE *XPhil*
VARIABLES
state
INVARIANT
$state \in NAT$
INITIALISATION
state := 1
OPERATIONS
eat(nn) =

PRE $nn \in NAT \wedge nn > 0$ ***THEN***

$state := state + nn$

END

END

Bibliography

[ABHV06] *Jean-Raymond Abrial, Michael Butler, Stefan Hallerstede, and Laurent Voisin. An open extensible tool environment for event-b. In* ICFEM*, pages 588–605, 2006.*

[Abr96] *J.-R. Abrial.* The B-book: assigning programs to meanings*. Cambridge University Press, New York, NY, USA, 1996.*

[AM98] *Jean-Raymond Abrial and Louis Mussat. Introducing dynamic constraints in b. In* B'98*, volume LNCS 1393, pages 83–128, 1998.*

[Ate01] *Atelier b. http://www.atelierb.societe.com/index_uk.html, 2001.*

[Bac80] *Ralph-Johan Back.* Correctness Preserving Program Refinements: Proof Theory and Applications*, volume 131 of* Mathematical Center Tracts*. Mathematical Centre, Amsterdam, The Netherlands, 1980.*

[Bag87] *Rajive Bagrodia. A distributed algorithm to implement n-party rendezvous. In* Proc. of the seventh conference on Foundations of software technology and theoretical computer science*, pages 138–152, London, UK, 1987. Springer-Verlag.*

[Bag89] *Rajive Bagrodia. Process synchronization: Design and performance evaluation of distributed algorithms.* IEEE Trans. Software Eng.*, 15(9):1053–1065, 1989.*

[BCo01] *B-toolkit. http://www.b-core.com/ONLINEDOC/BToolkit.html, 2001.*

[BFMW01] *Detlef Bartetzko, Clemens Fischer, Michael Möller, and Heike Wehrheim. Jass - java with assertions.* Electr. Notes Theor. Comput. Sci.*, 55(2):15, 2001.*

[BHR84] *S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes.* J. ACM*, 31(3):560–599, 1984.*

[BKS83a] *R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In* PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM Press.*

[BKS83b] *R. J. R. Back and R. Kurki-Suonio. Decentralization of process nets with centralized control. In* PODC '83: Proceedings of the second annual ACM symposium on Principles of distributed computing*, pages 131–142, New York, NY, USA, 1983. ACM Press.*

[BL05] *Michael J. Butler and Michael Leuschel. Combining CSP and B for specification and property verification. In* FM '05: Proceedings of the International Symposium of Formal Methods Europe*, volume LNCS 3582, pages 221–236, 2005.*

[BM02] Mark Brörken and Michael Möller. Jassda trance assertions: Runtime checking the dynamic of java programs. In International Conference on Testing of Communicating Systems*, 2002.*

[BMRA98] Juan Bicarregui, Brian Matthews, Brian Ritchie, and Sten Agerholm. Investigating the integration of two formal methods. Formal Asp. Comput.*, 10(5-6):532–549, 1998.*

[BR85] S. D. Brookes and A. W. Roscoe. An improved failures model for csp. In the Pittsburgh Seminar on Concurrency*, volume LNCS 197. Springer, 1985.*

[Bru01] Tatibouet Bruno. The JBTools Package*, 2001. Available at http://lifc.univ-fcomte.fr/ tatibouet/JBTOOLS/index_en.html.*

[BS05] Peter A. Beerel and Arash Saifhashemi. High Level Modeling of Channel-Based Asynchronous Circuits Using Verilog. In Communicating Process Architectures 2005*, sep 2005.*

[But93] Michael J. Butler. Refinement and decomposition of value-passing action systems. In CONCUR '93: Proceedings of the 4th International Conference on Concurrency Theory*, pages 217–232, London, UK, 1993. Springer-Verlag.*

[But96] Michael J. Butler. Stepwise refinement of communicating systems. Sci. Comput. Program.*, 27(2):139–173, 1996.*

[But97] Michael J. Butler. An approach to the design of distributed systems with b amn. In ZUM '97: Proceedings of the 10th International Conference of Z Users on The Z Formal Specification Notation*, pages 223–241, London, UK, 1997. Springer-Verlag.*

[But99] Michael J. Butler. csp2B: A practical approach to combining CSP and B. In World Congress on Formal Methods*, pages 490–508, 1999.*

[But06] Michael J. Butler. RODIN Deliverable D19: Intermediate report on methodology*, chapter 3.6, pages 47–56. RODIN project, 2006.*

[CGJ+01] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. Progress on the state explosion problem in model checking. In Informatics - 10 Years Back. 10 Years Ahead.*, pages 176–194. Springer-Verlag, 2001.*

[CH01] D. Cabeza and M. Hermenegildo. The pillow web programming library. Technical report, The CLIP Group, School of Computer Science, Technical University of Madrid, 2001. Available at http://www.clip.dia.fi.upm.es/.

[Che00] Jessica Chen. On verifying distributed multithreaded java programs. In HICSS '00: Proceedings of the 33rd Hawaii International Conference on System Sciences-Volume 8*, page 8010, Washington, DC, USA, 2000. IEEE Computer Society.*

[CKK+00] Jordi Cortadella, Michael Kishinevsky, Alex Kondratyev, Luciano Lavagno, and Alexandre Yakovlev. Hardware and petri nets: Application to asynchronous circuit design. In ICATPN*, pages 1–15, 2000.*

[Cle01] ClearSy. B Language Reference Manual*, 1.8.5 edition, 2001.*

[Cle02] ClearSy. Atelier B Translator User Manual*, 4.6 edition, 2002.*

[CS02] Ana Cavalcanti and Augusto Sampaio. From csp-oz to java with processes. In IPDPS '02: Proceedings of the 16th International Parallel and Distributed Processing Symposium*, page 161, Washington, DC, USA, 2002. IEEE Computer Society.*

[CSW03] ALC Cavalcanti, A. Sampaio, and JCP Woodcock. A Refinement Strategy for Circus*.* Formal Aspects of Computing*, 15(2 - 3):146 – 181, 2003.*

[Dij65] Edsger W. Dijkstra. Programming considered as a human activity. In the 1965 IFIP Congress*, pages 213–217. North-Holland Publishing Co., 1965.*

[Dij68] Edsger W. Dijkstra. A constructive approach to the problem of program correctness. BIT*, 8(3):174–186, 1968.*

[Dij97] Edsger Wybe Dijkstra. A Discipline of Programming*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.*

[DRS95] R. Duke, G. Rose, and G. Smith. Object-Z: A specification language advocated for the description of standards. Computer Standards & Interfaces*, 17(5–6):511–533, September 1995.*

[DS98] C. Demartini and R. Sisto. Static analysis of java multithreaded and distributed applications. In PDSE '98: Proceedings of the International Symposium on Software Engineering for Parallel and Distributed Systems*, page 215, Washington, DC, USA, 1998. IEEE Computer Society.*

[ELL94] René Elmstrøm, Peter Gorm Larsen, and Poul Bøgh Lassen. The ifad vdm-sl toolbox: a practical approach to formal specifications. SIGPLAN Not.*, 29(9):77–80, 1994.*

[EMCP99] Orna Grumberg Edmund M. Clarke and Doron A. Peled. Model Checking*. The MIT Press, 1999.*

[FC06] A. F. Freitas and A. L. C. Cavalcanti. Automatic Translation from Circus *to Java. In J. Misra, T. Nipkow, and E. Sekerinski, editors,* FM 2006: Formal Methods*, volume 4085 of* Lecture Notes in Computer Science*, pages 115–130. Springer-Verlag, 2006.*

[Fis97a] Clemens Fischer. CSP-OZ: A combination of Object-Z and CSP. Technical report, Fachbereich Informatik, University of Oldenburg, 1997.

[Fis97b] Clemens Fischer. CSP-Z: A combination of Z and CSP. Technical report, University of Oldenburg, 1997.

[Fis98] Clemens Fischer. How to combine Z with a process algebra. In J. Bowen, A. Fett, and M. Hinchey, editors, ZUM'98: The Z Formal Specification Notation*, volume 1493 of* LNCS*, pages 5–23. Springer, 1998.*

[Fis99] Clemens Fischer. Software development with Object-Z, CSP, and Java: A pragmatic link from formal specifications to programs. In Proceedings of the Workshop on Object-Oriented Technology*, pages 108–109, London, UK, 1999. Springer-Verlag.*

[Fis00] Clemens Fischer. Combination and Implementation of Processes and Data: From CSP-OZ to Java. *PhD thesis, Fachbereich Informatik, Universitat Oldenburg, 2000.*

[FLL+02] Cormac Flanagan, K. Rustan M. Leino, Mark Lillibridge, Greg Nelson, James B. Saxe, and Raymie Stata. Extended static checking for java. In PLDI '02: Proceedings of the ACM SIGPLAN 2002 Conference on Programming language design and implementation*, pages 234–245, New York, NY, USA, 2002. ACM.*

[Flo67] Robert W. Floyd. Assigning meanings to programs. In J. T. Schwartz, editor, Mathematical Aspects of Computer Science*, volume 19 of* Proceedings of Symposia in Applied Mathematics*, pages 19–32, Providence, Rhode Island, 1967. American Mathematical Society.*

[For03] Formal System Ltd. Failures-Divergence Refinement - FDR2 User Manual*, 2003.*

[FW99] Clemens Fischer and Heike Wehrheim. Model-checking CSP-OZ specifications with FDR. In 1st International Conference on Integrated Formal Methods*, pages 315–34. Springer-Verlag, 1999.*

[Gal96] A. Galloway. Integrated Formal Methods*. PhD thesis, University of Teesside, 1996.*

[GJSB00] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification, Second Edition*. Addison-Wesley, Boston, Mass., 2000.*

[GJSB05] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. The Java Language Specification Third Edition*. Addison-Wesley, Boston, Mass., 2005.*

[Goe03] Brain Goetz. Introduction to java threads. Sep 2003. https://www6.software.ibm.com/developerworks/education/j-threads/j-threads-a4.pdf.

[Goe04] Brain Goetz. Concurrency in JDK 5.0. 2004. http://www.ibm.com/developerworks/edu/j-dw-java-concur-i.html.

[HBS73] Carl Hewitt, Peter Bishop, and Richard Steiger. A universal modular actor formalism for artificial intelligence. In IJCAI*, pages 235–245, 1973.*

[HHS86] Jifeng He, C. A. R. Hoare, and Jeff W. Sanders. Data refinement refined. In ESOP '86: Proceedings of the European Symposium on Programming*, pages 187–196, London, UK, 1986. Springer-Verlag.*

[HK91] Iain Houston and Steve King. Cics project report experiences and results from the use of z in ibm. In VDM'91 Formal Software Development Methods*, volume Volume 551/1991, pages 588–596. Springer, 1991.*

[HL06] Max Haustein and Klaus-Peter Löhr. Jac: declarative java concurrency: Research articles. Concurrency and Computation : Practice and Experience*, 18(5):519–546, 2006.*

[Hoa69] C. A. R. Hoare. An axiomatic basis for computer programming. Commun. ACM*, 12(10):576–580, 1969.*

[Hoa78] *C. A. R. Hoare. Communicating sequential processes.* Commun. ACM, *21(8):666–677, 1978.*

[Hoa80] *C. A. R. Hoare. A model for communicating sequential processes. In* On the Construction of Programs*, pages 229–254. Cambridge University Press, 1980.*

[Hoa85] *C.A.R Hoare.* Communicating Sequential Processes. *Prentice Hall International, 1985.*

[Hol03] *Gerard J. Holzmann.* The SPIN Model Checker : Primer and Reference Manual. *Addison-Wesley Professional, September 2003.*

[HOS+93] *M. R. Hansen, E.-R. Olderog, M. Schenke, M. Fränzle, B. von Karger, M. Müller-Olm, and H. Rischel. A Duration Calculus semantics for real-time reactive systems. Technical Report [OLD MRH 1/1], Universitat Oldenburg, Germany, 1993.*

[HS00] *Klaus Havelund and Jens Skakkebaek. Practical application of model checking in software verification. In* the 6th Workshop on the SPIN Verification System*, 2000.*

[Jav] *http://java.sun.com/j2se/1.5.0/docs/api/java/lang/object.html.*

[Jon90] *Cliff B. Jones.* Systematic Software Development Using VDM. *Prentice-Hall Inc., 1990.*

[Law04] *Jonathan Lawrence. Practical application of csp and fdr to software design. In* Communicating Sequential Processes: The First 25 Years.*, volume 3525/2005 of* LNCS*, pages 151–174. Springer, 2004.*

[LB03] *Michael Leuschel and Michael J. Butler. Prob: A model checker for B. In* FME 2003: International Symposium of Formal Methods Europe*, pages 855–874, 2003.*

[LC03] *Gary T. Leavens and Yoonsik Cheon. Design by contract with JML, 2003.*

[Lea99] *Douglas Lea.* Concurrent Programming in Java. Second Edition: Design Principles and Patterns. *Addison-Wesley Longman Publishing Co., Inc., 1999.*

[Leu01] *Michael Leuschel. Design and implementation of the high-level specification language csp(lp) in prolog. In* PADL01*, LNCS 1990, pages 14–28. Springer-Verlag, 2001.*

[Lim95] *SGS-Thomson Microelectronics Limited.* occam 2.1 reference manual*, 1995.*

[LMC01] *Michael Leuschel, Thierry Massart, and Andrew Currie. How to make fdr spin ltl model checking of csp by refinement. In* FME '01: Proceedings of the International Symposium of Formal Methods Europe on Formal Methods for Increasing Software Productivity*, pages 99–118, London, UK, 2001. Springer-Verlag.*

[LPC+05] *Gary T. Leavens, Erik Poll, Curtis Clifton, Yoonsik Cheon, Clyde Ruby, David Cok, Peter Müller, and Joseph Kiniry.* JML Reference Manual. *Iowa State University, Department of Computer Science, Iowa State University, Iowa, USA, 1.156 edition, 8 2005.*

[LS03] Brad Long and Paul Strooper. A classification of concurrency failures in java components. In IPDPS '03: Proceedings of the 17th International Symposium on Parallel and Distributed Processing*, page 287.1, Washington, DC, USA, 2003. IEEE Computer Society.*

[MAV05a] C. Métayer, J-R. Abrial, and L. Voisin. RODIN Deliverable D7: Event-B language*. RODIN project, 2005.*

[MAV05b] C. Mtayer, J.-R. Abrial, and L. Voisin. Event-B Language*. Rigorous Open Development Environment for Complex Systems, rodin deliverable 3.2 edition, May 2005.*

[McE06] Alasdair A. McEwan. Concurrent Program Development*. PhD thesis, The University of Oxford, 2006.*

[MD99] Brendan P. Mahony and Jin Song Dong. Overview of the semantics of tcoz. In IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 66–85, London, UK, 1999. Springer-Verlag.*

[Mey92] Bertrand Meyer. Applying "design by contract". Computer*, 25(10):40–51, 1992.*

[Mil80] Robin Milner. A Calculus of Communicating Systems*. Springer-Verlag,, 1980.*

[Mil89] Robin Milner. Communication and concurrency*. Prentice-Hall, Inc., 1989.*

[Mil99] Robin Milner. Communicating and mobile systems: the pi-calculus*. Cambridge University Press, New York, NY, USA, 1999.*

[MK99] Jeff Magee and Jeff Kramer. Concurrency: State Models & Java Programs*. John Wiley & Sons, 1999.*

[Möl02] Michael Möller. Specifying and checking java using CSP. Technical report, Computing Science Department, University of Nijmegen, 2002.

[Mor88] Carroll Morgan. The specification statement. ACM Trans. Program. Lang. Syst.*, 10(3):403–419, 1988.*

[Mor90] Carroll Morgan. Of wp and CSP. In Beauty is our business: a birthday salute to Edsger W. Dijkstra*, pages 319–326. Springer-Verlag New York, Inc., 1990.*

[MPA05] Jeremy Manson, William Pugh, and Sarita V. Adve. The java memory model. In POPL '05: Proceedings of the 32nd ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 378–391, New York, NY, USA, 2005. ACM Press.*

[MS98] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z. In FASE'98: Fundamental Approaches to Software Engineering*, volume 1382 of* LNCS*, pages 205–220, 1998.*

[MS01] Alexandre Mota and Augusto Sampaio. Model-checking CSP-Z: strategy, tool support and industrial application. Sci. Comput. Program.*, 40(1):59–96, 2001.*

[MS02a] Giuseppe Milicia and Vladimiro Sassone. Jeeg: A programming language for concurrent objects synchronization. In JavaGrande/ISSCOPE 2002*, Seattle, November 2002.*

[MS02b] P. Mota, A.and Borba and A. Sampaio. Mechanical abstraction of CSP-Z processes. In FME 2002: Formal Methods-Getting IT Right. International Symposium of Formal Methods Europe*. Springer-Verlag, 2002.*

[MST92] David May, Roger Shepherd, and Peter Thompson. The t9000 transputer. In ICCD*, pages 209–212, 1992.*

[MU05] Petra Malik and Mark Utting. CZT: A framework for Z tools. In ZB 2005: Formal Specification and Development in Z and B*, LNCS, pages 65–84. Springer, 2005.*

[MW00] H. Muller and K. Walrath. Threads and swing, 2000. http://java.sun.com/products/jfc/tsc/articles/threads/threads1.html.

[Nel89] Greg Nelson. A generalization of dijkstra's calculus. ACM Trans. Program. Lang. Syst.*, 11(4):517–561, 1989.*

[OC04] M.V.M. Oliveira and A.L.C. Cavalcanti. From Circus to JCSP. In J. Davies et al., editor, Sixth International Conference on Formal Engineering Methods*, volume 3308 of* Lecture Notes in Computer Science*, pages 320 – 340. Springer-Verlag, November 2004.*

[Pel04] Jan Peleska. Applied formal methods - from csp to executable hybrid specifications. In Communicating Sequential Processes: The first 25 Years*, volume 3525/2005 of* LNCS*, pages 293–320. Springer, 2004.*

[Pet81] James Lyle Peterson. Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 1981.*

[PGB+05] Tim Peierls, Brian Goetz, Joshua Bloch, Joseph Bowbeer, Doug Lea, and David Holmes. Java Concurrency in Practice*. Addison-Wesley Professional, 2005.*

[PST96] Ben Potter, Jane Sinclair, and David Till. An Introduction to Formal Specification and Z*. Prentice Hall, 1996.*

[Pug00] William Pugh. The Java memory model is fatally flawed. Concurrency: Practice and Experience*, 12(6):445–455, 2000.*

[RDF+05] Edwin Rodríguez, Matthew B. Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, and Robby. Extending jml for modular specification and verification of multi-threaded programs. In ECOOP*, pages 551–576, 2005.*

[RGS94a] A. W. Roscoe, M.H. Goldsmith, and B.G.O. Scott. Denotational semantics for occam2, part 1. Transputer Communications*, 1:65–91, 1994.*

[RGS94b] A. W. Roscoe, M.H. Goldsmith, and B.G.O. Scott. Denotational semantics for occam2, part 2. Transputer Communications*, 2:25–67, 1994.*

[RH88] A. W. Roscoe and C. A. R. Hoare. The laws of occam programming. Theor. Comput. Sci.*, 60:177–229, 1988.*

[Ros98] A W Roscoe. The Theory and Practice of Concurrency*. Pearson Education, 1998.*

[RRS03] V. Raju, L. Rong, and G. S. Stiles. Automatic Conversion of CSP to CTJ, JCSP, and CCSP. In Jan F. Broenink and Gerald H. Hilderink, editors, Communicating Process Architectures 2003*, pages 63–81, 2003.*

[RS05] Rafael Ramirez and Andrew E. Santosa. Formal verification of concurrent and distributed constraint-based java programs. In ICECCS*, pages 76–84, 2005.*

[SB06] Colin Snook and Michael Butler. Uml-b: Formal modeling and design aided by uml. ACM Trans. Softw. Eng. Methodol.*, 15(1):92–122, 2006.*

[Sch00] Steven Schneider. Concurrent and Real-Time System: The CSP Approach*. John Wiley and Sons LTD, 2000.*

[Sch01] Steven Schneider. The B-Method: An Introduction*. Palgrave Macmillan, 2001.*

[SH00] Graeme Smith and Ian J. Hayes. Structuring real-time object-z specifications. In IFM '00: Proceedings of the Second International Conference on Integrated Formal Methods*, pages 97–115, London, UK, 2000. Springer-Verlag.*

[ST02] Steve Schneider and Helen Treharne. Communicating B machines. In ZB '02: Proceedings of the 2nd International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 416–435, London, UK, 2002. Springer-Verlag.*

[ST03] Steven Schneider and Helen Treharne. CSP theorems for communicating B machines. Technical report, Department of Computer Science, University of London - Royal Holloway, 2003.

[ST05] Steve Schneider and Helen Treharne. Csp theorems for communicating b machines. Formal Asp. Comput.*, 17(4):390–422, 2005.*

[STE05] Steve A. Schneider, Helen Treharne, and Neil Evans. Chunks: Component verification in csp——b. In IFM*, pages 89–108, 2005.*

[TB06] Edd Turner and Michael Butler. Symmetry reduction in the prob model checker. In FM 2006 Doctoral Symposium*, McMaster University, Canada, 2006.*

[TS99a] Helen Treharne and Steve Schneider. Capturing timing requirements formally in AMN. Technical Report CSD-TR-99-06, University of Surrey, Egham, Surrey TW20 0EX, England, 1999.

[TS99b] Helen Treharne and Steve Schneider. Using a process algebra to control B operations. In IFM '99: Proceedings of the 1st International Conference on Integrated Formal Methods*, pages 437–456, London, UK, 1999. Springer-Verlag.*

[TS00] Helen Treharne and Steve Schneider. How to drive a B machine. In ZB '00: Proceedings of the First International Conference of B and Z Users on Formal Specification and Development in Z and B*, pages 188–208, London, UK, 2000. Springer-Verlag.*

[TSB03] Helen Treharne, Steve Schneider, and Marchia Bramble. Composing specifications using communication. In ZB*, pages 58–78, 2003.*

[VHBP99] Willem Visser, Klaus Havelund, Guillaume Brat, and SeungJoon Park. Model checking programs. In Int. Conf. on Automated Software Engineering*, 1999.*

[WB04] Peter H. Welch and Fred R. M. Barnes. Communicating mobile processes. In 25 Years Communicating Sequential Processes*, pages 175–210, 2004.*

[WBM+07] Peter Welch, Neil Brown, James Moores, Kevin Chalmers, and Bernhard Sputh. Integrating and extending jcsp. In CPA '07: Communicating Process Architectures 2007*, 2007.*

[WBP06] P.H. Welch, F.R.M. Barnes, and F.A.C. Polack. Communicating complex systems. In Michael G Hinchey, editor, the 11th IEEE International Conference on Engineering of Complex Computer Systems (ICECCS-2006)*, pages 107–117, Stanford,California, August 2006. IEEE.*

[WC01] JCP Woodcock and ALC Cavalcanti. A concurrent language for refinement. In A. Butterfield and C. Pahl, editors, IWFM'01: The 5th Irish Workshop in Formal Methods*, BCS Electronic Workshops in Computing, Dublin, Ireland, July 2001.*

[WC02] J. C. P. Woodcock and A. L. C. Cavalcanti. The semantics of circus*. In D. Bert, J. P. Bowen, M. C. Henson, and K. Robinson, editors,* ZB 2002: Formal Specification and Development in Z and B*, volume 2272 of* Lecture Notes in Computer Science*, pages 184–203. Springer-Verlag, 2002.*

[Weh99] Heike Wehrheim. Data abstraction for CSP-OZ. In J. Davies J. Wing, J. Woodcock, editor, FM'99: World Congress on Formal Methods*, volume 1709 of* LNCS*, pages 1028–1047. Springer-Verlag, 1999.*

[Wel98] Peter H. Welch. Java Threads in the Light of occam/CSP. In P.H.Welch and A.W.P.Bakkers, editors, Architectures, Languages and Patterns for Parallel and Distributed Applications*, volume 52 of* Concurrent Systems Engineering Series*, pages 259–284, Amsterdam, April 1998. WoTUG, IOS Press.*

[Wir71] Niklaus Wirth. Program development by stepwise refinement. Commun. ACM*, 14(4):221–227, 1971.*

[WM00a] Peter H. Welch and Jeremy M.R. Martin. Formal analysis of concurrent java system. In CPA '00: Communicating Process Architectures 2000*, volume 58 of* Concurrent Systems Engineering*, pages 275–301. IOS Press (Amsterdam), 2000.*

[WM00b] Petre H. Welch and Jeremy M.R. Martin. A CSP Model for Java Multithreading. In P.Nixon and I.Ritchie, editors, Software Engineering for Parallel and Distributed Systems*, pages 114–122. ICSE 2000, IEEE Computer Society Press, June 2000.*