JACK: A Framework for Process Algebra Implementation in Java

Leonardo Freitas, Augusto Sampaio, Ana Cavalcanti
ljsf@cin.ufpe.br, acas@cin.ufpe.br, alcc@cin.ufpe.br
Informatics Center, Federal University of Pernambuco
Av. Luis Freire s/n CDU, 50.740-540, Recife, Brazil, (81)3271-8430,

August 2, 2002

Abstract

The construction of concurrent programs is especially complex due mainly to the inherent non-determinism of their execution, which makes it difficult to repeat test scenarios. Process algebras have been used to design and reason about these programs. This paper presents an approach to developing concurrent programs using a set of process algebra constructs implemented as an object-oriented framework in Java, called JACK. The main objective of the framework is the design and implementation of process algebra constructs that provides as naturally as possible, the algebraic idiom as an extension package to Java. This work emphasises the use of design patterns and pattern languages to properly build frameworks like this, achieving desired software engineering properties and software quality requirements. The user of the JACK framework is able to describe its process specification in Java.

Keywords: Process algebra, Java, Frameworks, Design patterns.

1 Introduction

Process algebras [14, 26, 20] are very attractive formalisms to describe concurrent and dynamic aspects of complex computer systems. However, the execution of process algebra specifications must deal with concurrent programming, a non-trivial task. The construction of concurrent programs is especially complex due mainly to the inherent non-determinism of their execution, which makes it difficult to repeat test scenarios.

A process algebra is a formal language that has notations for describing and reasoning about reactive systems. It introduces the concept of a system of processes, each with its own private set of variables, interacting only by sending messages to each other via *handshaken* communication. Synchronization on atomic events is used as the foundation for process interaction; it is also possible to express event occurrence, choice, parallel composition, abstraction, and recursion. By event we mean an instantaneous communication that can only occur when the participant process and the external environment agree on it. Examples of process algebras are CSP [26] and CCS [20].

Process algebras are useful because they bring the problems of concurrency into sharp focus; they can actually be taken as theoretical models of concurrency. Using them, it is possible to address the problems that arise both at the high level of constructing theories of concurrency and at the lower level of specifying and designing concrete systems, without worrying about other issues.

A process algebra theory consists of a well-defined set of semantic laws that describe the behaviour of processes. These descriptions may use both operational laws [23] and denotational laws [27] to properly define different concepts. Concurrency has proved to be a fascinating subject and there are many subtle distinctions which one can make, both at the level of choice of language constructs (i.e. its operators) and in the subtleties of the theories used to model them (i.e. its semantic laws).

This paper presents an approach for constructing concurrent programs using a set of process algebra constructs implemented as an object-oriented framework. A framework is provided in order to achieve desired software engineering quality requirements and support both *black-box* and *white-box* reuse. This framework is called JACK and stands for *Java Architecture with CSP kernel*. JACK carries "CSP" in its name because the current implementation of the framework focused on CSP operators. Nevertheless, as explained in this paper the framework is quite general, and can be easily adapted for other process algebras.

The main objective of the framework is an implementation in Java [3] of process algebra constructs that provide, as naturally as possible, the algebraic idiom as an extension to this concurrent object-oriented programming language. Java was chosen as a basis for our work basically because it is a sucessfull language that is well-accepted in the industry. We developed a design and implementation for the framework that provides the process abstraction as if it were included in the Java language itself (i.e. embedded in Java as an extension package). The framework solutions are based on micro-architectures of cooperating objects which ought to be applied, combined, and customized to build a process specification using that programming language extension.

As detailed in the next sections, other Java libraries do exists to support some process algebra operators. However, only very few operators are implemented, and the design of these libraries do not seem to address software engineering practices that support reuse, extensibility, compatibility and other quality factors.

In Section 2, we show a brief description of other available libraries. Next, in Section 3, the JACK framework design, architecture and most important decisions are presented. After that, in Section 4, a simple example of JACK usage is given. Finally, in Section 5, we present the conclusions and future work.

2 Other Libraries

There are some other related libraries [13, 22] already available that also aim at supporting process implementation. These libraries are implementations of occam [11] in Java. Basically, they provide the occam constructs as Java classes, in order to allow the user to implement its processes specifications. In spite of this fact, functional and non-functional aspects are mixed in these works, which makes the understanding of the library internals for extensions very difficult, leading to a loss of modularity. In what follows, a brief description of each library is given.

2.1 JCSP

JCSP is a library tailored to build networks of communicating processes. It conforms to the CSP [14, 26] model of communicating systems. In this sense, it can be brought to bear in the support of Java multi-threaded applications.

In JCSP, processes interact solely via CSP synchronising primitives, such as channels; they do not invoke each other's methods. Processes may be defined to run in sequence or in parallel. Processes may also be combined to wait passively on a number of alternative events, with one of them triggered into action only by the external generation of that event. Such collections of events may be serviced fairly (guaranteeing no starvation of one event by the repeated arrival of its siblings), by a user-defined priority, or in an arbitrary manner.

JCSP is an alternative to the built-in monitor model for Java threads [15, Chapter 14 and 17]. It is interesting to note that in JCSP processes are modeled using a simple extension to normal Java threads. Nevertheless, there is a mixture between non-functional controlling code and functional semantic code. The JCSP approach to model processes trusts and uses the thread architecture and thread scheduling policy of Java to model its concurrent behaviour.

JCSP also provides a wide range of plug and play components. These components are CSP processes normally used in many different kinds of specifications. This in many cases frees the user from having to start the specification from scratch. It also serves as usage examples and bugless implementation of common specification pieces.

Finally, it is important to note that the JCSP library reflects the occam realisation of CSP, with some extensions to take advantage of the dynamic nature of Java. An occam process declaration corresponds in JCSP to a class implementing the `CSProcess` interface, whose constructor parameters mirror the process parameters, and whose `run()` method mirrors the process body. For a detailed discussion about JCSP see [10].

2.2 CTJ

Concurrent Threads in Java (CTJ) [13], like JCSP, implements occam primitives like guards, alternatives, parallelism, channels, and so on. CTJ follows the same philosophy of JCSP in the treatment of threads and processes; that is, they share the idea of the implementation of active objects [17, Chapter 4]. A CTJ process is an active object with a private thread of control. A process is an active object when its `run()` method has been invoked by some thread of control and has not yet been returned (i.e. successfully terminated). A process is a passive object when its `run()` method is not invoked. Therefore, a parent process should only invoke `run()` on a child process when its child process is in passive state; parent processes are processes that executes immediately before their children processes. Sharing a process by two or more processes is forbidden and, therefore, the `run()` method can never be invoked simultaneously by multiple processes. This simple rule avoids race hazards and strictly separates each thread of control to enable a secure multithreading environment.

CTJ is modeled to capture real-time dependent aspects of systems. The new thread architecture that it implements is important to achieve this goal. It builds up completely new abstractions like threads, processor, context switcher, critical region, scheduler, and so on. This leads its implementation to be quite complex, but sufficiently accurate to

simulate real-time properties. The new thread architecture of CTJ adds many advantages to the library, like fine control over process scheduling policies. For a detailed discussion about CTJ see [10].

The CTJ approach to model processes is very robust. It builds up a completely new thread architecture to deal with concurrency, as opposed to JCSP, which, as previously discussed, is based on the thread architecture of Java. CTJ builds-up a completely new processor and process scheduler abstractions. There are differences between these libraries, nevertheless one can find many convergence points between them.

2.3 Discussion

In terms of functionality, these libraries implement the occam operators which include only a few of the CSP operators (specially considering the new version of CSP as defined by Roscoe [26]). For example, elaborate facilities such as multisynchronization and restrictions on communicated values as available in Roscoe's CSP are not considered in these libraries.

Trying to extend both JCSP and CTJ with these facilities, we found out that they are not designed to be easily extended. This motivated us to build the JACK as a framework with well-defined layers and problem domains. This also includes the separation of the non-functional layer exposed services like concurrency and synchronization schemes, from the functional layer responsibilities, that are the implementation of processes operational semantics; these topics are a non-trivial task. A presentation of the layers is given in Section 3.2. A detailed discussion about these topics can be found in [10, Chapter 5].

Another important aspect to be mentioned about JACK is the fact that it implements most CSP operators and its strong type system to allow the implementation of communication restrictions. In spite of this fact, the framework was built considering the possibility of extension to cover other important issues like data or time aspects. These aspects are important to integrated formalisms like CSP-OZ [6, 4] and Circus [29].

3 JACK Framework Design

A framework is represented by a set of recurring relationships, constraints, or design transformations in different aspects of object systems. Frameworks can be viewed as an implementation of a design pattern [5], or a related set of design patterns.

3.1 Framework Requirements

To properly build a framework, a set of requirements need to be achieved; they are mentioned below [28].

1. **Rigor and Formalism** — The use of a mathematical formalism based on a combination of both theoretical and experimental results.

2. **Separation of Concerns** — Separate handling of different aspects of program construction, in order to better control complexity and also allow separation of responsibilities, thus improving teamwork.

3. **Modularity** — Decomposition of a program into modules; the composition of a program from existing modules; and understanding each part of a program separately.

4. **Abstraction** — Identification of relevant aspects, with details ignored, thus allowing better complexity management and stepwise development.

5. **Anticipation of Change** — The identification of a program's future evolutions must be carried out at the requirements gathering stage. The isolation of parts to be changed may be done using modularity.

6. **Generality** — The construction or use of a tool or method to solve a more general problem may be reused in the context of different programs, like design patterns and pattern languages.

7. **Incrementability** — A program is built in successive increments, allowing in this way, incremental testing and debugging of programs.

We consider that the construction of a framework following these quality requirements is very important to any complex object-oriented system. In addition, dealing with the processes operational semantics [26] implementation using the Java language threading support is a complex problem by itself. Therefore, the so called *pattern initiative* seems to be imperative to build an extensible, reusable, modular, and yet simple process algebra implementation in Java.

Frameworks are normally divided into layers that solve specific problem domains. The definition of framework layers is also an important aspect of the framework construction. Each layer must capture a specific aspect to solve, and provide the solved functionality at well-defined and widespread interface points.

A primary problem in designing and integrating frameworks is related to the way they are modeled. A class-based approach based in UML [16, 12] is the dominant option, but it fails to adequately describe object collaboration behaviour. When designing a framework a set of guidelines ought to be observed and clarified. Role modeling for framework design [24] addresses three pertinent technical problems of designing, learning, and using object-oriented frameworks: complexity of classes, complexity of object collaboration, and lack of clarity of requirements put upon use-clients of a framework. For instance, the responsibilities an object has, the contexts these responsibilities depend on, how these responsibilities can be combined, and so on, must be considered while modeling a framework.

With these decisions, there are some important questions that need to be answered; some of them are listed below:

- How many and which layers will the framework need to have?
- How many and which responsibilities does each identified layer deal with?
- Which patterns and pattern languages to use in order to achieve the responsibility goals?
- Are these patterns implementations or designs available?
- Which modeling technique to use in order to accurately describe the framework roles?
- Is the framework prepared for gray-box reuse?

These questions are discussed below and addressed in more detail in [10, Chapter 5 and 6].

Answering these questions has guided the development of the JACK framework architecture which includes its layers, design patterns, and design pattern templates[1].

3.2 JACK Architecture

The most complex part of our framework implementation was identified as the definition of a generic and safe locking and threading scheme to execute processes. Both in CTJ and JCSP, this is mixed with process semantics, which makes the understanding of the library internals for extensions very difficult, and leads to modularity loss.

We investigated the framework modeling techniques [28, 24, 19], design patterns and pattern languages [25, 5, 17, 10]. Our aim is a reuseable, simple, expressive, and extensible architecture, that supports incremental development. JACK must also allow the extension of process operators by using traditional reuse mechanisms of object-oriented programming: class inheritance and object composition [5]. It must also support incremental development of operators.

Moreover, synchronization leads to a well-known problem with object-oriented frameworks and synchronization schemes called the inheritance anomaly problem [18]. This occurs most commonly because there is no language support for a kind of *lock inheritance*; this way proper subclass locking becomes obscure or even impossible. That vital non-functional problem is delegated to be treated by a framework called DASCO [28] due to its separation of inheritance for functionality reuse, and synchronization reuse. DASCO stands for *Development of Distributed Application with Separation and Composition of Concerns*. JACK provides a remodeled version of the design and implementation of DASCO in Java using role modeling [24] called JDASCO.

Our decision to use separation of concerns is motivated by the clearly observed properties of modularity and well-defined dependencies between functional (process operational semantics) and non-functional (threading and synchronization schemes) aspects. The separation of concerns approach states that the framework should be broken into well-defined layers that must capture a specific concern of the design and functionality. The JACK framework implementation deals with integration of concurrency, synchronization, recovery, and processes. It provides an implementation of those concerns and also considers concern composition. The DASCO implementation provides each concern without loosing desired object-oriented framework properties like expressiveness, simplicity, reusability, modularity, and incremental development.

DASCO was also designed as a framework, using a detailed and well-defined set of design patterns, composed together using a pattern language. The functional aspects of JACK, the implementation of process algebra operator semantics, is treated by another layer that directly interacts with JDASCO.

3.3 JACK Layers

Each JACK layer represents a piece of functionality that grants service implementation at well-defined points, in order to allow other layers to use this functionality independently. The layering strategy is very important to decouple and distribute the complexity of a large object-oriented system, making each layer to contribute with its partner. JACK has

[1] A design pattern template [24, Chapter 3] is a design pattern instantiated in a generic purpose programming language, in our case Java.

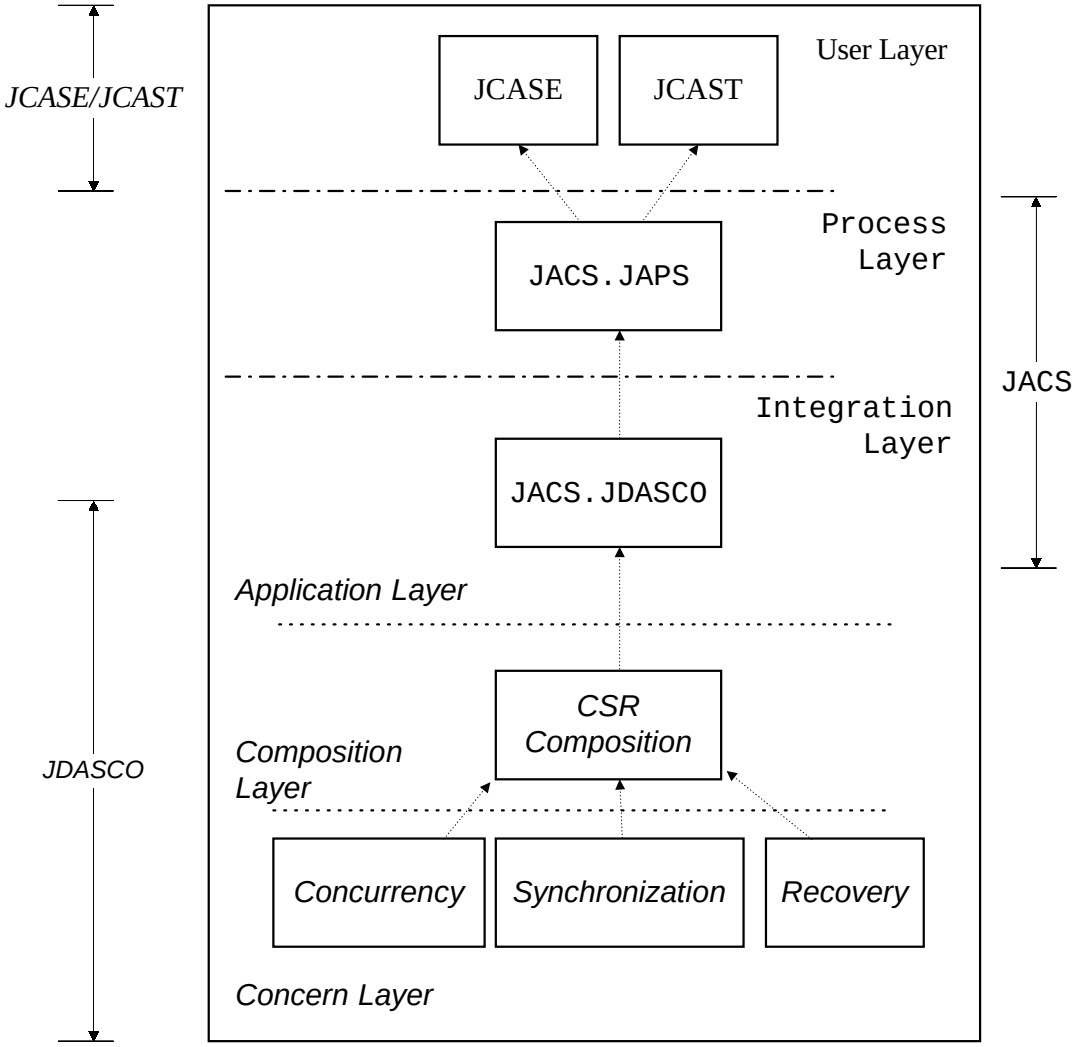

Figure 1: JACK layers detailed

three main layers that have themselves some sub-layers. Figure 1 shows a detailed view of how the layers are organized.

3.3.1 Execution Layer: JDASCO — Java Development of Distributed Application with Separation of Concerns

The bottommost layer, JDASCO, is responsible for dealing with low-level functionality, and for providing non-functional features like threads, monitor locks, transactions, etc. It is a Java extended version of DASCO. This layer provides and combines concurrency with thread management; synchronization with monitor and locks coordination. It also provides recovery through commitment, abortion, and preparation operations, and a walkable execution history.

JDASCO is divided in three sub-layers, as shown in Figure 1. The bottom layer is called the concern layer; it provides the concerns under consideration: concurrency, synchronization, and recovery. The middle layer is called the composition layer; it provides the combination of those concerns: concurrency with synchronization, synchronization

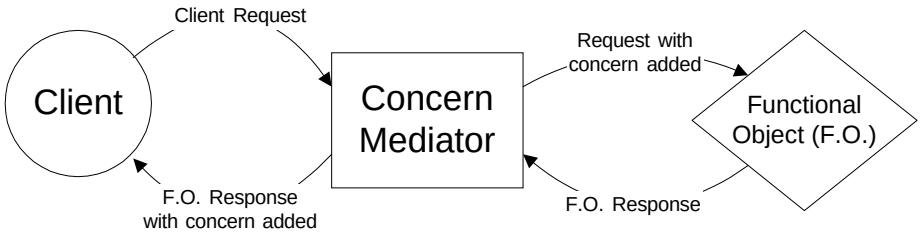

Figure 2: JDASCO Main Participants

with recovery, concurrency with recovery, and so forth. Finally, there is a top application layer example, showing how to use the concern combination. The JDASCO client ought to make use of its composition services filling in all user dependent roles, and following its strict usage guidelines; these are fully detailed in [28].

The service offered and the composition alternatives define a number of policies with configuration possibilities. They must be strictly defined by the application layer which, in the case of JACK, is one of the semantic sub-layers (`JACS.JDASCO`). The policy selection of each concern and composition must follow some restrictions and usage guidelines defined in [28] and in [10, Chapter 6]. The choice of this kind of service arrangement using separation of concerns was based on a thorough research in the concurrency development field [10, Chapter 3].

The services provided are based on three design patterns. They are called concurrent object, synchronized object, and recovery object, and are related to concurrency, synchronization, and recovery respectively. These patterns have a uniform set of participants: a JDASCO client, a concern mediator, and a functional object.

The scenario related to the functionality of these patterns was drawn as a client requesting some service of a functional object, as shown in Figure 2. That request is intercepted by the concern mediator responsible for introducing the desired concern service (i.e. concurrency, synchronization, or recovery) or compositions of them. By composing the pattern entities in this way, neither the functional object nor the client need to deal with each concern functionality directly, but just with its own related behaviour and responsibilities.

3.3.2 Semantic Layer: JACS — Java Architecture with CSP Semantics

The semantic layer acts as the user of JDASCO. It provides user processes and process algebra constructs semantics. It is the heart of the JACK framework, since it provides most of the functionality observed by the final user (i.e. a JACK client). Its main objective is to provide a high-level process representation. It makes use of the services of the execution layer.

The implementation of the semantics of the process constructs must follow some operational semantics description of the underlying considered process algebra. For instance, for CSP, it can use the semantic descriptions found in [26, 14]. In this case, the process semantics implementation considers the multisynchronization and restrictions on communicated values mentioned above.

Auxiliary constructors like types, channels, communications, alphabets, etc, are also provided. With these constructors, building a process that represents a specification

becomes easy, and results in readable and safe contained code. Currently, the framework provides most common operators like prefixes, choices, recursion, sequential composition, and parallel compositions. In [10, Chapter 4] these operators and some examples and situations illustrating useful usage guidelines of JACK are briefly described.

Attention should be given to the possibility of user defined process descriptions. JACK gives the user the ability to compose and define their own processes in whatever way they want through specific `Behaviour` interface implementations. For instance, the user process can make use of any complex Java data structure.

The JACS semantic layer has two sub-layers, as shown in Figure 1. The one called JACS.JAPS (process sub-layer) provides the semantic machinery to describe specifications (i.e. operators and auxiliary constructs) and processes. JAPS stands for *Java Applications for Processes Surveyor*. The other layer, called JACS.JDASCO, links the semantic elements with JDASCO, acting as the JDASCO application layer; it is called the integration sub-layer, and it plays all the roles expected by the JDASCO framework as described in [10].

Another important aspect of the design of JACK is the use of a symbolic approach to deal with processes execution. With this choice, it is possible for the framework to deal with infinite data types. Dealing with this kind of types is important in the object-oriented world, since class object instances, like `Object`, are considered an infinite data type. The available designs and implementations provide a normalization followed by an expansion of the process network. Nevertheless, for infinite data types, expansion cannot be considered as a possible solution. The JACK framework design provides a strong and robust type system that is able to deal with this symbolic approach for process execution, providing infinite data types to define process specifications.

3.3.3 User Layer

The user layer is open and should be used by the JACK client to implement its specification or desired functionality. In other words, it is just a design layer that represents the entry point of the JACK framework; it does not provide any functionality related to processes.

For instance, JACK provides a user layer example called JAST; it stands for *JACK Abstract Syntax Trees*. The AST representation layer acts as a user of the JACS semantic layer. It provides AST representation of the process algebra operators. It could be used by a parser to build an AST that represents a process specification in JACK.

Since JACK is a process-oriented framework, this means that it provides process functionality to the final user that uses it for any purpose. The stated purpose in this view of the framework is to describe some process specification, possibly in pure formalism like CSP, or in combined ones, like Circus. A more detailed description of each layer can be found in [10, Chapter 5 and 6].

3.4 A Simplified Class Diagram for JACK

A JACK process is composed by two main aspects:

1. The process interface (`CSPProcess`) — responsible for dealing with process infrastructure and generic information.

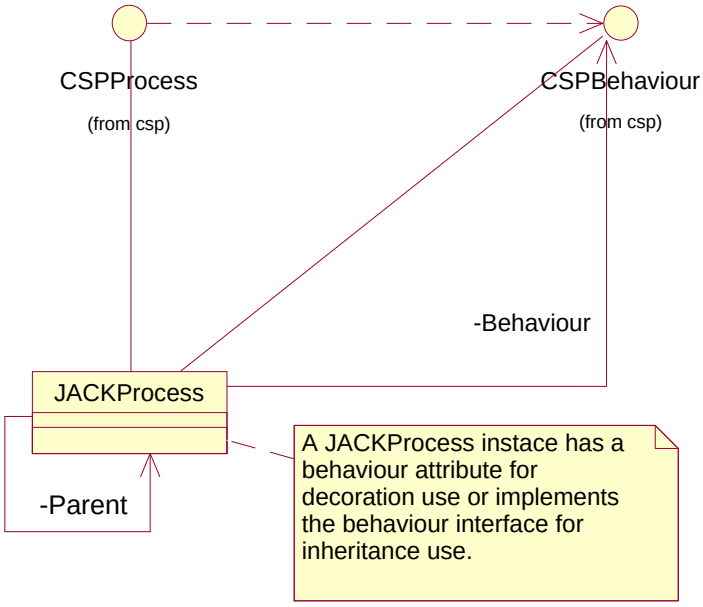

Figure 3: JACK Process x Behaviour Relationship

2. The process behaviour (`CSPBehaviour`) — responsible for dealing with process behavioural and specific information.

To completely define a process, it must have a companion behaviour: a specific definition to be plugged inside a generic execution environment. In this way, a process algebra operator can be viewed as a JACK process environment with a behaviour already defined and provided by the framework.

JACK users that want to describe their own processes must provide a behaviour interface implementation to achieve their needs. The user behaviour interface must provide a function f describing the process behaviour and a domain D defining the set of events which the process is initially prepared to engage [14, Chapter 1].

JACK already provides a process interface implementation for the generic part of a process specification, called `JACKProcess`. The user must just provide its specific desired behaviour; this can be achieved either implementing an interface and submitting that behaviour to the already implemented process interface, or just deriving from `JACKProcess`. Thus, the process interface deals with structural (generic) properties, and the user behaviour interface deals with behavioural (specific) properties. In Figure 3 this relationship between a process and a behaviour interfaces and corresponding implementation is provided; this modeling scheme is very close to the Java `Thread` and `Runnable` design [3]. For details on topics related to this modeling scheme, see [9].

4 JACK Example

This section presents a simple example of the use of the JACK operators. The example provides the implementation of a stop-and-wait protocol specification in CSP. The stop-and-wait protocol implements an one-place buffer. It consists of two processes, the sender

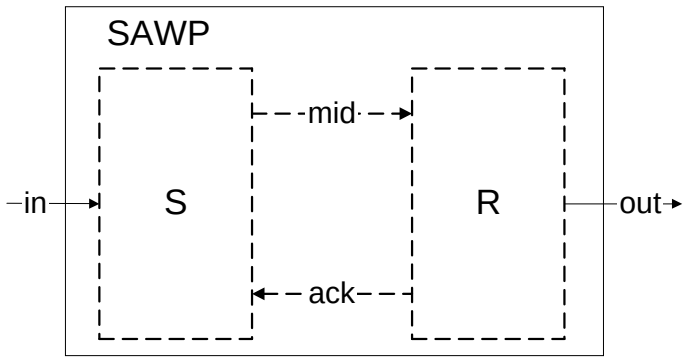

Figure 4: Stop-and-Wait Protocol

channel ack%Event Type signal.
channel in, out, mid: int

% Build a set containing x' that comes from $\{0, 1\}$

$$T = \{x^{'} | x^{'} \leftarrow \{0 \ldots 1\}\}$$

$$S = in?x : T \rightarrow mid!x \rightarrow ack \rightarrow S$$
$$R = mid?y \rightarrow out!y \rightarrow ack \rightarrow R$$
$$SAWP = S[|\{|mid, ack|\}|]R \setminus (\{|mid|\} \cup \{ack\})$$

Figure 5: Stop-and-Wait Protocol — CSP Specification

process S and the receiver process R: a message is input by S, passed to R by S, and finally output by R. The protocol is illustrated by Figure 4.

Having accepted a message through in, the sender process S passes the message to R along the mid channel, and then wait for an acknowledge before accepting the next message. The receiver process R accepts messages along mid, and sends an acknowledgement once a message has been output through out. The channel mid and the acknowledgement signal event ack are private connections and should have no participants other than S and R: they are internal events.

The two processes are designed to be combined in parallel. The processes S and R must wait for synchronization on their in and out channels from the external environment. This means that they ought to synchronize on these channels with other processes that perform the physical input and output operations. The specification of the protocol in CSP code is given in Figure 5. The set comprehension represented by T, denotes the restriction on the values that can be communicated through channel c. The "$\rightarrow$" notation, indicates the prefix communication, either for input ($c?x \rightarrow \ldots$) or output ($c!v \rightarrow \ldots$). Two CSP binary operators are also used. The first one represents the parallel composition of processes S and R synchronizing on communications through channels mid and ack, and is denoted by $S[|\{|mid, ack|\}|]R$. The next one is the hiding operator that abstracts these internal communications from the external environment point of view; it is denoted by the "$\setminus$" (hiding operator).

Parts of the Java code that represents the specification of Figure 5 are given in Figure 6; the complete example can be found in [10]. For conciseness, we omit the import section and the package declaration.

In the main method, we first create a StopAndWaitProtocol behaviour, attach it to a process instance, and start it using the default JACK supervisor environment. Since a JACK process is an active object, after the start() call, the main thread finishes its execution and only the threads related to JACK processes remain active.

The StopAndWaitProtocol constructor first creates the used channels with the appropriate type. After that, it starts building each subprocess of the protocol (i.e. S, R, and $SAWP$). For instance, to build the sender process S, we create a name Sr to use in recursive calls to S recursive calls are given by the special process Sr. The prefix processes expect a channel and a process to follow. In our example, the WritePrefix WPS is on channel *mid*, and the process thaat follows is an EventPrefix, where no communication is involved, just synchronization; the event is *ack* and the process that follows is the recursive call Sr. The ReadPrefix RPS is on the channel *in*. For technical reasons, we pass two copies of WPS as well; this is because WPS is the process parameter of RPS and we use a constructor that embedes extra checks on the types of *in* and *mid*. The construction of the recieve process R is similar to that of S. The definition of the main process ($SAWP$) is straightforward; one just needs to define the synchronization and hiding alphabets, and then create the main process passing the previously instantiated processes.

The `CSPBehaviour` interface that this class implement represents the specific behaviour of a user process specification. It simply expects two methods to be defined: one that establishes how the process must interact with its supervisor execution environment, and another one that defines which communications are immediately available to be performed. Due to space restrictions, we will not provide a deeper discussion about this important aspect of the framework here. It is completely described and defined in [10].

5 Conclusion

We have presented an approach to implement process algebras with separation and composition of concerns, integrating concurrent and object-oriented programming [28]; framework role modeling [24]; design pattern and pattern languages [5]. In contrast with other similar libraries [22, 13] available, JACK is modeled using object-oriented framework techniques, which leads to a more robust and adaptative white-box framework [24] that is appropriate for evolution.

JACK has generic features which allow its use in many different process algebra contexts. For instance, to implement other process algebra operators one just needs to follow the stated [28, 10] JDASCO roles and guidelines. JDASCO is divided by concerns: concurrency, synchronization, recovery, and composition of them. Each concern defines two level implementation policies to be selected and properly configured. It also specifies some management classes to be implemented in order to satisfy all the framework obligations, what we called here JDASCO roles. These details for JDASCO are described in [28, 10] and the details to the integration with CSP are described in [8].

JACK is also a realistic case study on the use of the DASCO framework described in [28]; JACK validates the results mentioned there and realizes some of the mentioned future work, like remodeling DASCO to use framework role modeling. The use of DASCO

```
public class StopAndWaitProtocol implements CSPBehaviour {
    public static void main(String[] args) {
        CSPBehaviour swpBehaviour = new StopAndWaitProtocol();
        CSPProcess SWP = new JACKProcess(swpBehaviour);

        SWP.start(DEFAULT_SUPERVISOR_ENVIRONMENT);
    }

    private CSPOperator SAWP;// Main Hiding Process

    public StopAndWaitProtocol() {
        // Create the channels and channel type
        CSPType channelType = new IntType(0, 1);// Int type {0 ... 1}
        CSPChannel in = new Channel("in", channelType);
        CSPChannel out = new Channel("out", channelType);//Int range typed channel
        CSPChannel mid = new Channel("mid", channelType);
        CSPChannel ack = new Channel("ack", EVENT_TYPE);//Event typed channel

        // Definition of Process S = in?x : T → mid!x → ack → S
        Recursion Sr = new Recursion("S");

        // Creates the write prefix process mid!x → ack → S
        WritePrefix WPS = new WritePrefix(mid, new EventPrefix(ack, Sr));

        // Creates the read prefix process in?x → mid!x → ack → S
        ReadPrefix RPS = new ReadPrefix(in, WPS, WPS);

        // Creates the recursive process S.
        Mu S = new Mu(RPS, Sr);

        // Definition of Process R = mid?y → out!y → ack → R
        Recursion Rr = new Recursion("R");

        WPR = new WritePrefix(out, new EventPrefix(ack, Rr));//out!x → ack → R
        RPR = new ReadPrefix(mid, WPR, WPR);//mid?x → out!x → ack → R
        R = new Mu(RPR, Rr);// Creates the recursive process R.

        // Definition of Process SAWP = S[|{|mid, ack|}|]R \ {|mid, ack|})

        //Builds the synchronization alphabet: {|mid, ack|}
        CSPAlphabet SAWPSynchAlpha = new Alphabet();
        SAWPSynchAlpha.add(new CSPChannel[] { fIn, fOut, fMid, fAck });

        //Builds the hiding alphabet: {|mid, ack|}
        CSPAlphabet fSAWPHiddenhAlpha = new Alphabet();
        fSAWPHiddenhAlpha.add(new CSPChannel[] { fMid, fAck });

        // Creates the parallelism between S and R synchronizing and hiding on the alphabets
        SAWP = new Hiding(new GeneralizedParallel(S, fSAWPSynchAlpha, R), fSAWPHiddenhAlpha);

    ...

}
```

Figure 6: Stop-and-Wait Protocol using JACK

has shown to be a very attractive solution for complex concurrent systems. It is neither easy nor complete, but it has proven to be adequate for our purposes.

JACK also provides an implementation of a symbolic approach to execute processes without expansion, which leads the specification to be able to deal with infinite data types. This is a contribution when contrasted with tools like FDR, which cannot deal with infinite data types because it uses an expansion strategy to analyse processes. For a physical implementation of processes, it is important to accept infinite data types like `Object` and numeric types like `integer`.

Due to the layering organization of the framework, important software engineering aspects like incremental development, composition of concerns, modularity, expressiveness, abstraction, anticipation of changes, and so on, are achieved.

The implementation is built using separation of concerns in a way that is highly beneficial to class-based design of frameworks. This work emphasizes the use of design patterns and pattern languages to properly build frameworks, achieve desired software engineering properties and software quality requirements. The user of the JACK framework is able to describe its process specification in Java. At the moment we implement the CSP operators, but as pointed out before, it can be extended/adapted to other process algebras or combined algebras like Circus. The user can also represent infinite data types without starvation of the process network execution, since the framework provides a symbolic approach for dealing with this sort of types. This is in contrast with the expansion approach used by tools like FDR [7].

The extension developer and the user of the JACK framework is benefited by expressive documentation of the whole framework, modeling decisions, and implementation details [10, 9]. Moreover, the use of design patterns makes it easier to extend JACK, since it shares the same idiom with other framework developers, an inherited benefit of the use of design patterns.

The JACK framework combines the strengths of role modeling [24] and design patterns [5] with those of class-based and layered modeling, while leaving out their weaknesses. This increases the framework robustness. Another major aspect is the possibility to directly use the framework as the heart of a set of important tools to be used in the formal methods field, like formal translation tools and model checkers.

Some architectural improvements can be done for JACK. For instance, JACK defines some new concern policies for JDASCO not available in the original work of DASCO. These new policies extends JDASCO to be more modular. Some of the extended policies were not used due to time constraints. Nevertheless, it can be an interesting work to better explore policies like the history sensitive synchronization and recovery policies. Improvements in the layer integration procedure, in the type system, and in the framework documentation are also under development.

An interesting set of tools can be built using JACK, in order to provide tool support for analysis and execution of specifications.

There is a work [4] that provides formal translation rules from CSP-OZ [6] to CTJ. Another possible work is to do the same formal translation to JACK.

In [1], a tool that translates CSP-OZ specifications to FDR following the guidelines in [21] is presented. An extension of this tool to translate CSP-OZ specifications directly to JACK could be considered.

There is a working plan under consideration to check the feasibility of using JACK to be the base to build a model checker for process algebras. A related work called

JMocha [2] that implements general primitives for model checkers implementation is also under consideration. This close relation with tools is an important contribution of the JACK framework to the formal methods field, since it is impossible to have a productive application of a formal approach without tool support.

6 Acknowledgments

This work is partially supported by **CNPq** and **CAPES**, the Brazilian Research Agencies, and the **University of Kent at Canterbury**. We would like to acknowledge Paulo Borba and Nabor Chagas for design discussion about JACK architecture. Alexandre Mota and Adalberto Cajueiro also made relevant suggestions.

References

[1] Alexandre Motta Adalberto Farias and Augusto Sampaio. De CSPz para CSPm - Uma Ferramenta transformacional Java (in Portuguese). In *Proceedings of V Formal Methods Workshop of XVII SBES in Rio de Janeiro*, October 2001.

[2] Rajev Alur. Mocha: Modularity in Model Checking., April 2002. http://www-cad.eecs.berkeley.edu/~mocha.

[3] Ken Arnold and James Gosling. *The Java Programming Language*. The Java Series. Addison Wesley, 2 edition, 1997.

[4] Ana Cavalcanti and Augusto Sampaio. From CSP-OZ to Java with Processes. In *Proceddings of the Workshop on Formal Methods for Parallel Programming: Theory and Applications*. IEEE CS Press, April 2002.

[5] Ralph Jonhson Erich Gamma, Richard Helm and John Vlissides. *Design Pattern Elements of Reusable Object-Oriented Software*. Adison Wesley, 1 edition, 1995.

[6] Clemens Fischer. *Combination and Implementation of Process and Data: from CSP-OZ to Java*. PhD thesis, University of Oldenburg, January 2000.

[7] Formal Methods (Europe) Ltd. *FDR User's Manual version 2.28*, 1997.

[8] Leonardo Freitas. CSP Implementation in JDASCO: Process Supervision, Backtracking and Multisynchronization, March 2002. Technical report available at http://www.jackcsp.hpg.com.br (site under construction).

[9] Leonardo Freitas. JACK - A Process Algebra Implementation for Java Home Page, March 2002. Technical report available at http://www.jackcsp.hpg.com.br (site under construction).

[10] Leonardo Freitas. JACK: A Process Algebra Implementation in Java. Master's thesis, UFPE, April 2002.

[11] Michael Goldsmith. *Programming in OCCAM 2*. Prentice Hall, 1 edition, 1988.

[12] Ivar Jacobson Grady Booch, James Rumbaugh. *The Unified Modeling Language User Guide*. Object Technology Series. Adisson Wesley, 1998.

[13] G.H. Hilderink and E.A.R. Hendriks. Concurrent Threads in Java - CTJ v0.9, r17, September 2000. http://www.rt.el.utwente.nl/javapp.

[14] C.A.R. Hoare. *Communicating Sequential Process*. Prentice Hall, 1985.

[15] Guy Steele James Gosling, Bill Joy and Gilad Bracha. *The Java Language Specification: Online Draft Version*. Java Series. Adisson Wesley, 2 edition, 2000.

[16] Grady Booch James Rumbaugh, Ivar Jacobson. *The Unified Modeling Language Reference Manual*. Object Technology Series. Addison Wesley, 1998.

[17] Doug Lea. *Concurrent Programming in Java*. The Java Series. Addison Wesley, 2 edition, February 2000.

[18] Satoshi Matsuoka and Akinori Yonezawa. Analysis of inheritance anomaly in object-oriented concurrent programming languages. In *Research Directions in Concurrent Object-Oriented Programming*, pages 107–150. MIT Press, 1993.

[19] Bertrand Meyer. *Object-Oriented Software Construction*. Series in Computer Science. Prentice Hall, 2 edition, 1997.

[20] R. Milner. A Calculus for Communicating Systems. *LNCS 92*, 1980.

[21] Alexandre Mota and Augusto Sampaio. Model checking csp-z. *Science of Computer Programming, Elsivier*, 40:59–96, 2001. www.elsevier.nl/locate/scico.

[22] P.D.Austin and P.H.Welch. Java Communicating Sequential Process - JCSP, August 2000. http://www.cs.ukc.ac.uk/projects/ofa/jcsp/.

[23] G.D. Plotkin. A structural approach to operational semantics. *Lecture Notes DAIMI FN-19, Aarhus University, Denmark*, 1991.

[24] Dirk Riehle. *Framework Design: A Role Modeling Approach*. PhD thesis, Swiss Federal Institute of Technology Zurich, Universität Hamburg, 2000.

[25] Linda Rising. *The Pattern Almanac 2000*. Software Pattern Series. Addison Wesley, May 2000.

[26] A.W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1 edition, 1997.

[27] David A. Schmidt. *Denotational Semantics: A Methodology for Language Development*. Allyn and Bacon, Inc., 1 edition, 1986.

[28] António Manuel Ferreira Rito Silva. *Concurrent Object-Oriented Programming: Separation and Composition of Concerns using Design Patterns, Pattern Languages, and Object-Oriented Frameworks*. PhD thesis, Instituto Superior Técnico, Universidade Técnica de Lisboa, March 1999.

[29] Jim Woodcock and Ana Cavalcanti. A Concurrent Language for Refinement. *5th Irish Workshop on Formal Methods*, 2001.