PSP_{VDC}: Una propuesta que incorpora el Diseño por Contrato Verificado al Personal Software Process

Silvana Moreno
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
smoreno@fing.edu.uy

Álvaro Tasistro
Escuela de Ingeniería
Universidad ORT Uruguay
Montevideo, Uruguay
tasistro@ort.edu.uy

Diego Vallespir
Facultad de Ingeniería
Universidad de la República
Montevideo, Uruguay
dvallesp@fing.edu.uy

***Resumen*—El desarrollo de software se ha vuelto una actividad muy importante en el mundo actual. Existen números procesos de desarrollo de software que buscan aumentar la calidad de los productos y disminuir los tiempos de salida al mercado. Sin embargo, el software contiene defectos y éstos causan fallas potencialmente graves durante su ejecución. Este trabajo propone un nuevo proceso de desarrollo de software denominado PSP_{VDC} que combina el enfoque del Personal Software Process (PSP) y del Diseño por Contrato Verificado (VDbC) con el objetivo de mejorar la calidad de los productos con respecto al PSP. Además se presenta una revisión sistemática de la literatura que busca conocer las adaptaciones al PSP que hayan sido documentadas, en particular aquellas que incorporan métodos formales.**

***Términos Clave*—métodos formales, diseño por contrato verificado, personal software process, revisión sistemática.**

I. INTRODUCCION

La construcción de software confiable es uno de los desafíos de la Ingeniería de Software. El tamaño, la complejidad de las aplicaciones de software, las tasas apresuradas de entregas de proyectos, las características de los equipos de desarrollo, entre otros, hacen que los productos de software contengan defectos. Estos defectos pueden ocasionar fallas en el software mientras éste se está ejecutando [1]. La presencia de defectos provoca un alto costo de corrección de las aplicaciones, así como una pérdida de confianza en el mismo. La búsqueda de desarrollar software *cero defectos* ha dado lugar a un gran número de procesos y metodologías de desarrollo.

El Personal Software Process (PSP) es uno de estos procesos. El PSP aplica disciplina de proceso y gestión cuantitativa al trabajo individual del ingeniero de software. Promueve la utilización de prácticas específicas durante todas las etapas del desarrollo con el objetivo de mejorar la calidad del producto y aumentar la productividad del individuo [2], [3].

Por otro lado, los métodos formales también buscan construir software cero defectos. Estos métodos son un conjunto de técnicas y herramientas para especificar, desarrollar y verificar sistemas software mediante el uso de lenguaje matemático formalizado. Consisten en demostrar matemáticamente que los programas producidos cumplen sus especificaciones.

El diseño por contrato (DbC) es una técnica para el diseño de los componentes de un sistema de software, mediante su especificación en un lenguaje formal, generalmente en la forma de pre- y post-condiciones escritas en Lógica de Primer Orden. Cuando las técnicas y herramientas incorporadas permiten demostrar que se cumplen las especificaciones establecidas, estamos en presencia de un método formal, generalmente llamado Verified Design by Contract (VDbC).

En este artículo se presenta una adaptación del PSP que incorpora VDbC con el objetivo de mejorar, en comparación con el uso del PSP sin VDbC, la calidad de los productos desarrollados. Se denomina a esta propuesta PSP_{VDC}.

Esperamos de esta forma aportar en la búsqueda de procesos que ayuden a desarrollar productos de software muy cercanos a cero defectos.

La estructura del artículo es la siguiente: primero se presentan las nociones básicas del diseño por contrato y el PSP (secciones II y III). Luego en la sección IV se presenta una revisión sistemática de la literatura realizada con el propósito de conocer las adaptaciones existentes al PSP que incorporan métodos formales. En la sección V se presenta el PSP_{VDC} y la sección VI presenta algunas medidas de calidad del PSP_{VDC}. Finalmente, se indican las conclusiones obtenidas (sección VII).

II. DISEÑO POR CONTRATO

Los métodos formales son un conjunto de técnicas para especificar, desarrollar y verificar sistemas de software mediante el uso del lenguaje matemático. Consisten en demostrar matemáticamente que los programas producidos cumplen sus especificaciones.

El Diseño por Contratos (DBC) es un método propuesto por Bertrand Meyer, e inspirado en las ternas de Hoare, para especificar el comportamiento de módulos de software [4], [5].

La idea principal detrás de DBC es que una clase y sus clientes tienen un "contrato" entre sí. Los clientes deben garantizar ciertas condiciones antes de llamar a un método definido por la clase, y en cambio la clase da como garantías a los clientes ciertas propiedades que se cumplen luego de finalizada la ejecución del método invocado.

Ciertas implementaciones del DbC permiten que los contratos sean verificables en tiempo de ejecución. Los contratos son incluidos en el código del programa en una determinada sintaxis, y se traducen a código ejecutable por el compilador. Por lo tanto, cualquier violación del contrato que se produce mientras se ejecuta el programa puede ser detectado inmediatamente.

Los contratos de software se especifican mediante la utilización de expresiones lógicas denominadas aserciones. Existen diferentes tipos de aserciones, entre ellas se encuentran las precondiciones y las poscondiciones de programas. En lo que refiere a los contratos en la orientación a objetos las precondiciones y poscondiciones normalmente se definen a nivel de los métodos de las clases.

Una precondición establece lo que espera recibir el proveedor. Visto de otro modo, define las condiciones que debe garantizar el cliente al momento de pedirle al proveedor cierto servicio.

Una poscondición da como garantías al cliente ciertas propiedades que se cumplen después de la llamada al método. En definitiva, especifica lo que se cumple luego de que se ejecuta el método. Visto desde el lado del proveedor, la poscondición es lo que éste debe asegurar que se cumpla, siempre y cuando la precondición se respete al momento de invocar el servicio.

El Java Modeling Language, abreviado JML y en español «Lenguaje de Modelado para Java» es un lenguaje de especificación para programas Java. Las especificaciones JML describen formalmente el comportamiento de clases y métodos; se escriben como comentarios de anotación Java en el código fuente, que luego pueden compilarse con cualquier compilador de Java [6].

Añadir especificaciones JML a un programa ayuda a comprender qué función debe realizar un método o una clase. También ayuda a encontrar defectos en los programas, puesto que se puede comprobar si un método cumple con su especificación cada vez que se ejecuta.

Existen herramientas que permiten compilar la especificación formal para ciertos lenguajes de especificación. Para el JML existe el compilador JMLC, este es una extensión de un compilador Java y compila los programas Java con especificaciones JML. En el código generado se agregan instrucciones ejecutables que chequean en tiempo de ejecución si los métodos cumplen con sus especificaciones. En caso de que una especificación no se cumpla, se interrumpe la ejecución del programa y se notifica cuál es la especificación no respetada. Esto permite detectar defectos y por ende se puede utilizar como una forma de probar el programa durante el desarrollo de software.

El JML dispone de dos cláusulas específicas para indicar las precondiciones y las poscondiciones de un método: requires y ensures. Estas cláusulas deben ir situadas justo antes de la declaración del método.

Además, el JML añade un conjunto de operadores que hace más cómodo en muchos casos construir aserciones complejas. Entre estos operadores se incluyen los cuantificadores más comunes (\forall, \exists, \sum, \product, \max, \min, etc) que hacen del JML un lenguaje similar al lenguaje de predicados de lógica.

También el JML define dos seudo variables que pueden ser utilizadas en las poscondiciones de los métodos; \result: valor devuelto por el método y \old(E): valor de la expresión E al comenzar la ejecución del método.

Presentamos a continuación una forma de especificar utilizando JML el método **merge**. Dicho método recibe como parámetros dos vectores de enteros ordenados de menor a mayor. El método retorna otro vector con esos elementos ordenados de menor a mayor. Además, los vectores de entrada no deben ser alterados, es decir, sus valores se mantienen incambiados al salir del método.
La firma del método es la siguiente:

```
public int[] getMerge(int[] a, int[] b)
```

Al invocar al método se deben pasar dos vectores ordenados de menor a mayor, lo que nos indica una precondición que se debe cumplir. Para cumplir con dicha precondición es necesario que para todos los elementos del vector se cumpla que el elemento que está en la posición i del vector sea menor o igual al elemento que se encuentra en la posición i+1.

Una forma de especificar formalmente dicha precondición se muestra en las siguientes dos aserciones.

```
/*@ requires (\forall int i; i >= 0 && i < a.length-1;
 a[i] <= a[i+1]); @*/
/*@ requires (\forall int j; j >= 0 && j < b.length-1;
b[j] <= b[j+1]); @*/
```

El cuantificador \forall sirve para recorrer el vector desde una posición y en determinado rango, haciendo cumplir determinada condición. En este caso se recorren los vectores y chequea que para todos elementos el valor de a[i] sea menor o igual al de a[i+1]. La forma de leer la precondición es “Para todo i entre cero (inclusive) y el largo del vector menos 1 (sin incluir éste) se cumple que a[i] es menor o igual a a[i+1]”.

Al terminar de ejecutarse el método se debe cumplir que los vectores de entrada no fueron alterados. Esto se especifica como una poscondición y una forma de especificarlo es utilizando la pseudovariable \old(e).

Se recorren los vectores de entrada y verifica que para todos sus elementos el valor en cada posición después de llamar al método es igual al que tenia previo al llamado (a[i] = = \old(a[i])).

```
/*@ ensures (\forall int i; i >= 0 && i <= a.length-1;
 a[i] == \old(a[i])); @*/
```

```
/*@ ensures (\forall int j; j >= 0 && j <= b.length-1;
 b[j] == \old(b[j])); @*/
```

Cabe aclarar, que en ambos casos se podría haber utilizado la misma variable cuantificadora (i) ya que es local al \forall.

Hasta aquí hemos tenido en cuenta las precondiciones y poscondiciones para los vectores de entrada al método. Veamos ahora las condiciones que se deben cumplir en lo que respecta al vector esperado como resultado del método.

Como se mencionó anteriormente el vector devuelto debe contener de forma ordenada los mismos elementos que los vectores pasados por parámetro.

Para referenciar al vector resultado se utiliza la pseudovariable \result y se recorre el mismo, como ya vimos anteriormente, para verificar el ordenamiento de los elementos de menor a mayor.

```
/*@ ensures (\forall int k; k >= 0 && k < \result.length-1;
\result[k] <= \result[k+1]); @*/
```

Una forma de especificar informalmente que el vector devuelto contenga los mismos elementos de los vectores pasados por parámetro y no otros es:

- Si denominamos $|v|_x$ la cantidad de veces que el elemento x aparece en el vector v, entonces: para cada elemento x que aparece en el vector resultado r, se cumple $|r|_x = |a|_x + |b|_x$.
- El largo del vector resultado debe ser igual a la suma del largo de los vectores a y b.

Especificamos formalmente que el largo del vector resultado es igual a la suma del largo del vector a y el b de la siguiente forma:

```
/*@ ensures \result.length == (a.length + b.length); @*/
```

Para especificar que cada elemento del vector resultado aparece en él tantas veces como en los dos vectores de entrada utilizamos \num_of. Este cuantificador permite contar la cantidad de elementos de los vectores. Especificamos que para todos los elementos del vector resultado la cantidad de elementos repetidos es igual a la suma de los mismos elementos repetidos en a y en b.

Formalmente especificamos como sigue:

```
/*@ ensures (\forall int i;  i>= 0 && i < \result.length;
      (int)(\num_of int j; j>=0 && j<\result.length; \result[j]
      == \result[i]) == (int)(\num_of int j; j>=0 &&
      j<a.length; a[j] == \result[i]) + (int)(\num_of int j;
      j>=0 && j<b.length; b[j] == \result[i])   ); @*/
```

La especificación completa para el método getMerge se muestra de forma más clara en la figura 2.

La especificación de contratos ejecutables permite introducir pruebas en el propio código y son muy efectivas para encontrar defectos durante el desarrollo de software. Sin embargo, como se desprende de este ejemplo, las especificaciones formales no son sencillas y tienen, como cualquier actividad, un costo asociado. Como con cualquier otra técnica se debe estudiar el costo-beneficio previamente a aplicarla.

III. PERSONAL SOFTWARE PROCESS

El Personal Software Process, conocido por sus siglas como PSP, es una metodología de desarrollo de software creada por el Instituto de Ingeniería del Software (SEI). Está dirigido a los ingenieros, permitiendo mejorar la forma en la que construyen software. Considera aspectos como la planificación, calidad, estimación de costos y productividad.

El PSP fue diseñado para ayudar a los ingenieros a seguir un proceso disciplinado de ingeniería del software, enseñándoles a planificar y dar seguimiento al trabajo, utilizando un proceso bien definido y medido. Se caracteriza por ser de uso personal y se aplica a programas pequeños de menos de 10.000 líneas de código. Se centra en la administración del tiempo y en la administración de la calidad a través de la eliminación temprana de defectos [2], [3].

Los principios del PSP son:

- Cada ingeniero es diferente, para ser más eficiente, debe planificar su trabajo basándose en su experiencia personal.
- Usar procesos bien definidos y cuantificados.
- Los ingenieros deben asumir la responsabilidad personal de la calidad de sus productos.
- Cuanto antes se detecten y corrijan los errores menos esfuerzo será necesario.
- Es más efectivo evitar los defectos que detectarlos y corregirlos.
- Trabajar bien es siempre la forma más rápida y económica de trabajar.

El PSP define un conjunto de fases que el ingeniero sigue durante el proceso desarrollo. Todas las tareas y actividades realizadas en estas fases están definidas en un conjunto de documentos conocidos como scripts. Los scripts permiten seguir el proceso de forma disciplinada, y ayudan al ingeniero a identificar sus fortalezas y sus debilidades, y crecer a través de un proceso de aprendizaje y mejora.

Las fases son Planning, Design, Design Review, Code, Code Review, Compile, Unit Test, y Post Mortem. La figura 1 muestra las fases de PSP.

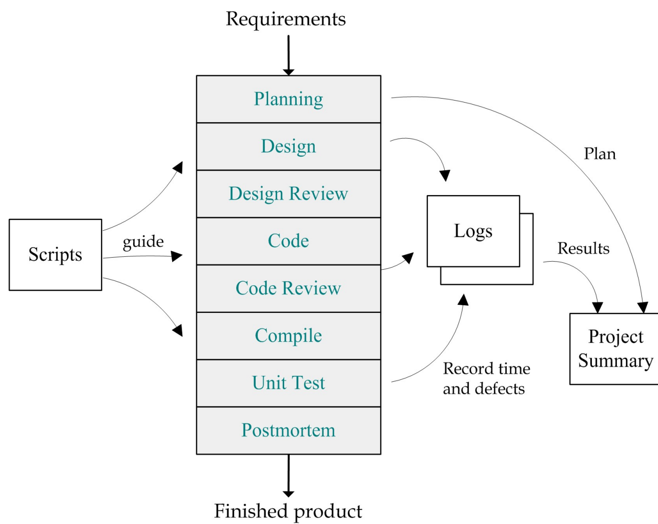

Fig. 1. Fases del PSP

Para cada fase el ingeniero recoge la información del tiempo empleado en la fase y los defectos inyectados y

```
/*@ requires (\forall int i; i >= 0 && i < a.length-1; a[i] <= a[i+1]); @*/

/*@ requires (\forall int j; j >= 0 && j < b.length-1; b[j] <= b[j+1]); @*/

/*@ ensures (\forall int i; i >= 0 && i < a.length-1; a[i] == \old(a[i])); @*/

/*@ ensures (\forall int j; j >= 0 && j < b.length-1; b[j] == \old(b[j])); @*/

/*@ ensures \result.length == (a.length + b.length); @*/

/*@ ensures (\forall int k; k >= 0 && k < \result.length-1; \result[k] <= \result[k+1]); @*/

/*@ ensures (\forall int i; i>= 0 && i < \result.length;
      (int)(\num_of int j; j>=0 && j<\result.length; \result[j] == \result[i]) ==
      (int)(\num_of int j; j>=0 && j<a.length; a[j] == \result[i]) + (int)(\num_of int j; j>=0 && j<b.length; b[j] == \result[i]) );
@*/

public int[] getMerge(int[] a, int[] b)
```

Fig. 2. Especificación método getMerge

removidos. La información sobre los defectos incluye el tipo de defecto, el tiempo en detectar y corregir el defecto, la fase en la que se inyecta y la fase en la que se remueve. El PSP define 10 tipos de defectos a utilizar. La tabla I presenta los tipos de defectos junto con una breve descripción de los defectos que deben registrarse para cada tipo.

TABLA I. TIPOS DE DEFECTOS DEL PSP

Tipo	Nombre	Descripción
10	Documentación	Comentarios o mensajes
20	Sintaxis	Ortografía, puntuación, tipos, formato de instrucciones
30	Componentes o configuración	Administración de cambios, control de versiones, bibliotecas
40	Asignación	Declaración, nombres duplicados, alcance, límites
50	Internas	Llamados y referencias a procedimientos, E/S, formatos de usuarios
60	Chequeo de tipos	Mensajes de error, chequeos inadecuados
70	Datos	Estructura, contenido
80	Función o lógicos	Defectos por lógica, apuntadores, ciclos, recursión, cálculos, funciones
90	Sistema	Configuración, sincronización, memoria
100	Ambiente	Problemas de diseño, compilación, prueba, u otros con los sistemas de desarrollo

El PSP propone una vista de cuatro dimensiones para lograr un diseño completo. En la vista interna-estática se especifica la lógica interna del programa, normalmente el pseudo código. La vista interna-dinámica corresponde a las posibles maquinas de estado, es decir, los estados, las transiciones y acciones entre estados que puede tener el programa. La vista externa-estática corresponde a la estructura de clases y la vista externa-dinámica define las interacciones entre el sistema y el usuario (diagramas de casos de uso, diagrama de secuencia del sistema,

etc). La información recabada en cada una de estas vistas es definida en cuatro templates como se muestra en la tabla II.

TABLA II. TEMPLATES DE DISEÑO DEL PSP

	Interna	**Externa**
Estática	Logic Specfication Template	Function Specfication Template
Dinámica	State Machine Template	Operational Specification Template

El PSP se enseña mediante un curso. Durante el mismo, los ingenieros construyen programas de software (ejercicios) mientras aprenden progresivamente a planificar, desarrollar y seguir las prácticas del PSP. En el primer ejercicio, se comienza con un proceso simple (el proceso base, llamado PSP 0); a medida que se avanza en el proceso se agregan nuevas fases como Design Review y Code Review, así como también formas de estimar tiempos y tamaño. Los nombres de los procesos y los elementos que se van incorporando durante el curso se muestran en la figura 3. El nivel 2.1 es el proceso completo del PSP.

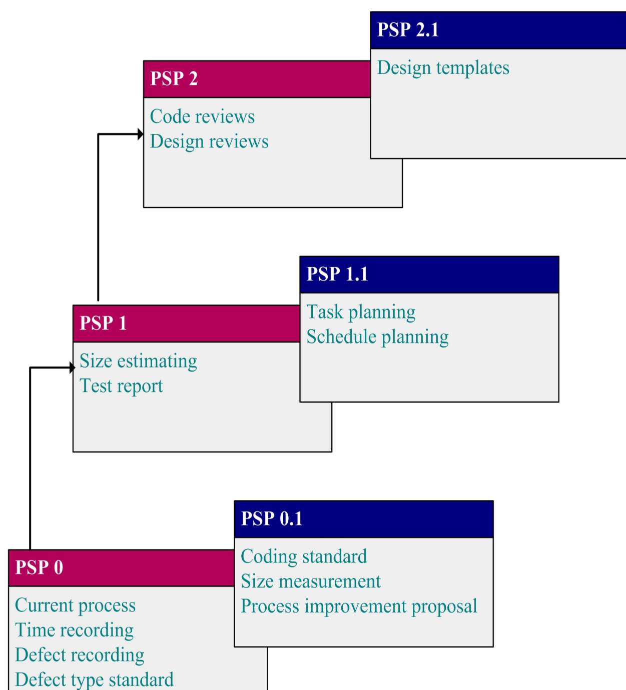

Fig. 3. Niveles del PSP

IV. REVISIÓN SISTEMÁTICA

Es de nuestro interés conocer las adaptaciones al PSP que hayan sido documentadas, en particular nos interesa conocer las adaptaciones que incorporan métodos formales. Presentamos en esta sección una revisión sistemática que busca conocer la literatura existente sobre adaptaciones al PSP.

Una revisión sistemática de la literatura es un medio de identificación, evaluación e interpretación de toda la información que se disponga relativa a una pregunta de investigación, área temática o fenómeno de interés [7].

La revisión sistemática realizada responde a dos preguntas de investigación:

(1) ¿Se han realizado adaptaciones al PSP?

(2) ¿Las adaptaciones propuestas al PSP incorporan métodos formales? ¿Cómo lo hacen?

Para responder a estas preguntamos definimos un protocolo que especifica los métodos que se utilizan para llevar a cabo la revisión sistemática. Se define la cadena de búsqueda que consta de tres partes. La primera parte está relacionada con las formas de nombrar el Personal Software Process, la segunda parte con los posibles sinónimos de la palabra "adaptar", es decir las diferentes formas de presentar un cambio al PSP y la última parte se refiere al uso de métodos formales. Las búsquedas fueron realizadas en bibliografía en idioma inglés, por ende la cadena de búsqueda es en inglés:
("personal software process") **and** ((adapting or extending or over or incorporating) **or** ("formal methods" or "design by contract"))

Las bibliotecas digitales utilizadas para la búsqueda de artículos fueron SCOPUS, Springer, IEEE Xplore y EBSCO. Estos buscadores abarcan las principales colecciones de revistas y actas de conferencias de ingeniería de software.

Además, se realizó una búsqueda de forma manual en "*A Bibliography of the Personal Software Process (PSP) and the Team Software Process (TSP)*" y en las actas de los Simposios TSP realizados desde el 2006 al 2012. *"A Bibliography of the Personal Software Process (PSP) and the Team Software Process (TSP)"* es un documento creado por el *Software Engineers Institute* (SEI) que contiene libros, capítulos, secciones y publicaciones en general referentes a PSP y TSP. Esta búsqueda permite encontrar trabajos realizados con respecto al tema de interés que posiblemente no hayan sido indexados en las bibliotecas digitales.

Cada artículo encontrado en la búsqueda es evaluado para decidir si debe o no incluirse teniendo en cuenta el título, las palabras claves y el resumen. Aquellos artículos que cumplan al menos una de las siguientes condiciones serán excluidos:

- No se centran en el Personal Software Process
- Capítulos de libros o libros
- Duplicados en diferentes fuentes
- No escritos en inglés
- No propongan una adaptación al PSP o no incorporen métodos formales al mismo

Debido a que la cantidad de artículos que se espera encontrar es muy baja optamos por no realizar un control de la calidad de los artículos.

La síntesis de datos consistirá en agrupar a los artículos en base a las dos preguntas de investigación planteadas, es decir, si es un artículo que adapta a PSP mediante la incorporación de métodos formales, si es un artículo que adapta a PSP en cualquier otro aspecto o si es un artículo que utiliza métodos formales con PSP pero sin adaptarlo.

Al aplicar la cadena de búsqueda en la base de IEEE se obtienen 31 resultados. En esta base se filtra la búsqueda eliminando los contenidos de tipo *Books & eBooks*.

Utilizando EBSCO se eliminan los filtros que buscan en las bases *CAB Abstracts 1990-Present*, *Dentistry & Oral Sciences Source*, MEDLINE y *Ovid Journals* y se obtienen 34 resultados.

Para SCOPUS se agrega el filtro de excluir los artículos pertenecientes a las aéreas *Life Sciences* y *Health Sciences* obteniendo 20 resultados.

Por último en la base de Springer se filtran sólo artículos, de la disciplina *Computer Science* y en inglés, obteniendo 20 resultados.

En "*A Bibliography of the Personal Software Process (PSP) and the Team Software Process (TSP)*" se encuentra 1 artículo que adapta PSP utilizando métodos formales. Este artículo también fue encontrado con EBSCO e IEEE.

En los Simposios de TSP se encontró 1 artículo que no fue encontrado por ninguna otra fuente así como una presentación (es decir, copias de diapostitivas). Ambos son del Simposio 2012, el artículo combina métodos formales con PSP y la presentación incorpora al PSP técnicas basadas en modelos.

Algunos artículos fueron publicados en más de una revista/conferencia y/o están repetidos en más de un buscador, por lo que se tienen en cuenta una sola vez a estos artículos resultando:

- EBSCO 34 artículos
- Springer 19 artículos
- Scopus 18 artículos
- IEEE 29 artículos
- Simposio TSP 2013 1 artículos
- Simposio TSP 2013 1 presentación.

Finalmente al aplicar los criterios de inclusión/exclusión para los resultados de cada buscador el resultado es:

- Scopus 2 artículos
- IEEE 1 artículo
- Simposio TSP 2013 1 artículo
- Simposio TSP 2013 1 presentación

Tres de los cinco trabajos encontrados adaptan el PSP para incorporar métodos formales.

Babar y Potter proponen un nuevo proceso denominado B-PSP que incorpora el método formal B al proceso de desarrollo personal [8]. Suzumori, Kaiya y Kaijiri proponen la combinación del método formal VDM y PSP (VDM *over* PSP) [9] al igual que la propuesta de Kusakabe, Omori and Araki (VDM-PSP) [10]. Estos tres trabajos son presentados en las siguientes secciones.

De los otros dos trabajos encontrados, uno modifica al PSP para llevar adelante un proceso de desarrollo de software de a pares (*pair-programming*) [11] y el otro propone una adaptación el PSP incorporando técnicas basadas en modelos.

La propuesta de adaptación del PSP para el desarrollo de software de a pares, denominada Collaborative Software Process (CSP) describe los cambios necesarios para distinguir las tareas que debe seguir cada rol durante el desarrollo. Los autores explican cómo los scripts, templates y formularios del PSP son ajustados, describiendo en particular los cambios necesarios para distinguir las tareas que debe el rol desarrollador, el rol observador y los momentos en los que se debe realizar el intercambio de roles. Otro cambio propuesto es la utilización de dos *checklists* de diseño y dos *checklists* de código, uno para cada rol.

La presentación del Simposio TSP propone una adaptación al PSP incorporando técnicas de integración basadas en modelos. El proceso propuesto modifica la fase de diseño para desarrollar un modelo de diseño que describa la estructura externa del sistema y el comportamiento y luego refina este modelo para describir la estructura interna del sistema y el comportamiento. La fase de revisión de diseño incorpora la comprobación del modelo mediante una herramienta de análisis estático. Por último las fases de código y Unit Test incorporan la generación de código parcial y la generación de los test en base a los modelos respectivamente.

Estas dos últimas adaptaciones no serán explicadas en mayor profundidad ya que no hacen uso de los métodos formales.

A. Propuesta B-PSP [8]

Los autores de la propuesta B-PSP combinan los enfoques de métodos formales y PSP con el objetivo de construir software fiable y de alta calidad dentro de los plazos y presupuesto esperado.

B-PSP incorpora al PSP las fases *Specification*, Auto *Prover*, *Animation* y *Auto Proof*. Durante la fase de *Specification* se especifican las B-machines utilizando el lenguaje formal B. Posteriormente las fases de *Auto Prover* y *Animation* se realizan con la ayuda del B toolkit, controlando la sintaxis básica y la dependencia entre machines, generando las obligaciones de pruebas y brindando animación. Durante la fase de *Auto Proof* se realiza una demostración interactiva de la prueba mediante el B toolkit.

B-PSP propone además la generación de código automáticamente, aunque no explica durante que fase se realiza. La propuesta elimina las fases de *design*, *design review*, *code, code review, code compile* y *Unit Test*, pero no

aclara si las actividades realizadas durante estas se realizan en alguna de las nuevas fases.

La propuesta B-PSP mantiene la idea de PSP de fomentar a los desarrolladores individuales a medir y evaluar, reconocer las causas de la introducción de defectos y evitarlas en el futuro. Al igual que PSP se recogen los datos de tiempo y esfuerzo; en esta propuesta se registran además las líneas de especificación formal (LOS) y la cantidad de pruebas (# Proof).

Planning
Specification
Auto prover
Animation
Auto Proof
Postmortem

Fig. 4. Fases de B-PSP

B. Propuesta VDM over PSP [9]

La propuesta VDM *over* PSP propone combinar el método formal VDM y PSP con el objetivo de mejorar la gestión de la calidad. Esta gestión considera dos aspectos: la prevención de defectos y la eliminación de defectos. Para la prevención de defectos en VDM *over* PSP se utilizan *checklist* y *guidelines* de revisión al igual que en PSP, mientras que la estrategia para eliminar defectos es agregar actividades que permitan logar una mayor calidad en el diseño. Estas actividades se proponen en las nuevas fases *VDM-SL syntax review, syntax check, type check* y *validation*. .

VDM *over* PSP agrega y modifica varias fases al PSP, específicamente modifica la fase de diseño para incorporar la especificación formal en el lenguaje VDM-SL. Luego de la especificación formal se realiza la revisión de diseño de PSP y a posterior agrega las nuevas fases *VDM-SL syntax review, syntax check, type check* y *validation*.

Durante *VDM-SL syntax review* el usuario chequea las especificaciones en búsqueda de defectos de sintaxis. Las fases *syntax review, type check* and *validation* son ejecutadas por el Toolbox comprobando la sintaxis, el chequeo de tipos e invariantes y precondiciones de forma dinámica. En esta última fase no aplican técnicas de verificación formal ya que la propuesta VDM-PSP es utilizada por estudiantes principiantes.

Los autores proponen introducir VDM *over* PSP en 4 niveles, de forma de introducir las técnicas de VDM poco a poco.

- VDM *over* PSP0: corresponde al proceso base de PSP a utilizar, es decir PSP 2.1
- VDM *over* PSP1: se introducen las especificaciones funcionales en VDM-SL. Además de VDM *over* PSP0, el usuario utiliza VDM-SL para especificar tipos de datos, pre-condiciones, descripciones de funciones implícitas e invariantes de cada tipo de datos.
- VDM *over* PSP2: se introducen las especificaciones lógicas en VDM-SL. Además de VDM *over* PSP1, el usuario especifica la especificación interna de cada función.
- VDM *over* PSP3: se introduce la *Toolbox* para validar las especificaciones escritas en VDM-SL. Este nivel corresponde al proceso completo VDM *over* PSP.

Planning
Design
Design review
VDM-SL syntax review
Syntax check
Type check
Validation
Code
Code review
Compile
Unit Test
Postmortem

Fig. 5. Fases de VDM *over* PSP

C. Propuesta VDM-PSP [10]

La propuesta VDM-PSP combina VDM y PSP basándose en la importancia de producir software de alta calidad siguiendo un proceso de desarrollo eficaz y eficiente. Los autores creen que a pesar de que PSP proporciona cuatro plantillas de diseño para aumentar la confianza en el diseño, el uso de los métodos formales con sus lenguajes de especificación formal y sus herramientas serán eficaces en la reducción de los defectos.

VDM-PSP incorpora el método formal VDM que dispone de varios lenguajes de especificación formal y la VDM toolkit que permite el chequeo de sintaxis, chequeo de tipos, la utilización del intérprete y la generación de obligaciones de pruebas. La propuesta planteada mantiene la mismas fases que PSP, modificando la fase de diseño para incorporar la especificación formal utilizando alguno de los lenguajes que provee VDM (por ejemplo, VDM ++) y la fase de revisión de diseño para incorporar la herramienta VDM toolkit. No queda clara en la propuesta en qué fase se realiza la generación de las obligaciones de la prueba.

Planning
Design
Design review
Code
Code review
Compile
Unit Test
Postmortem

Fig. 6. Fases de VDM-PSP

La revisión sistemática realizada permite conocer la existencia de las adaptaciones realizadas al PSP que incorporan métodos formales. Los trabajos encontrados presentan diferentes formas de incorporar métodos formales al PSP, agregando nuevas fases, modificando fases o simplemente introduciendo actividades al PSP (manteniendo sus fases).

Sin embargo, ninguno de los trabajos encontrados incorpora el enfoque de Diseño por Contrato (DbC). Este método formal tiene sus características propias que difieren tanto de B como de VDM. Por esto, es necesaria una adaptación particular al PSP de forma de poder usar el DbC en conjunto con el PSP.

V. El PSP_{VDC}

El PSP_{VDC} es una adaptación al PSP que incorpora el enfoque de diseño por contrato verificado. El objetivo de esta adaptación es mejorar la calidad (defectos/kloc) del producto de software desarrollado, en comparación con el PSP.

A. Incorporando VDbC al PSP

Para incorporar VDbC se deben realizar las actividades de Especificar Formalmente y Demostrar Formalmente la correctitud del código con respecto a dicha Especificación. Incorporar estas actividades al proceso puede realizarse de diversas formas; proponiendo fases nuevas, proponiendo nuevos pasos a fases ya existentes en PSP o combinando ambas. En PSP_{VDC} incorporamos dos nuevas fases para dar soporte a este enfoque, la fase de Formal Specification y la fase de Proof.

La fase de Formal Specification se propone luego de finalizado el diseño y realizada la revisión del mismo. Durante esta fase se especifican los métodos de las clases anteriormente diseñados y revisados. Llevar a cabo la especificación en una nueva fase permite conocer el esfuerzo (tiempo) dedicado exclusivamente a especificar y conocer los defectos que se inyectan y remueven al realizar esta actividad.

La fase de Proof es propuesta luego de la compilación de código. Durante esta fase se realiza la demostración de la correctitud del código con respecto a la especificación formal.

Al igual que en la fase de Formal Specification el proponer una nueva fase para la verificación permite conocer los tiempos y defectos específicamente asociados a la verificación formal.

La figura 7 muestra el agregado de estas dos nuevas fases.

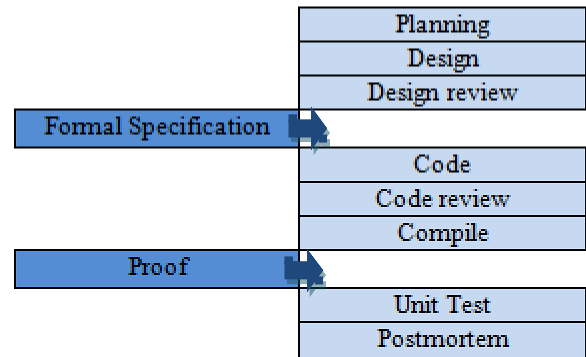

Fig. 7. Incorporando las fases de Formal Specification y Proof

El PSP propone revisiones personales como mecanismo efectivo y rápido de remoción de defectos. Específicamente, propone la revisión del diseño y la revisión del código. Durante estas revisiones el ingeniero revisa el trabajo realizado siguiendo una check list en busca de defectos.

En PSP_{VDC} la nueva fase de Formal Specification puede incurrir a la inyección de defectos. Siguiendo la misma idea de PSP de intentar remover la mayor cantidad de defectos de forma temprana se propone una fase de revisión de la especificación formal. La fase Formal Specification Review se realiza a continuación de la especificación formal y su objetivo es revisar, con la ayuda de una check list, la especificación recientemente construida.

Luego de construida y revisada la especificación formal se propone una fase de compilación de la especificación formal. Durante esta fase se compila la especificación formal utilizando herramientas permitiendo generar código ejecutable además de remover defectos. En la figura 8 se presenta el agregado de estas fases.

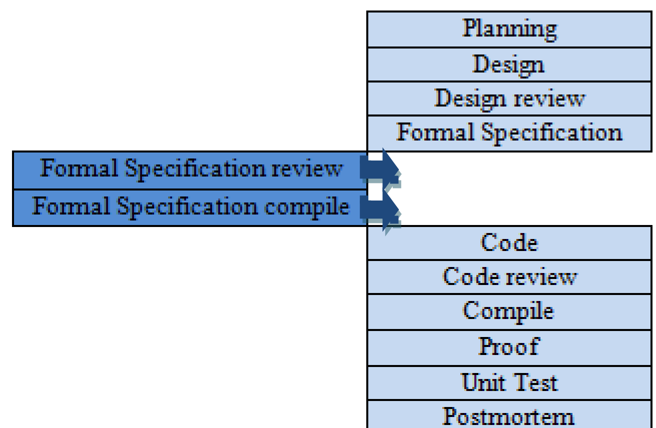

Fig. 8. Incorporando las fases de Formal Specification review y compile

B. Desde Diseño a Codificación

Un proceso de desarrollo de software consiste de un conjunto ordenado de pasos a seguir para lograr un producto de software que resuelva un problema específico. El PSP define a través de las fases de Design, Design Review, Code, Code Review, Compile, Unit Testing y Postmortem un proceso disciplinado donde las salidas de una fase son la entrada directa a la fase inmediatamente posterior.

En el PSP, una de las actividades a realizar durante la fase de Design es el pseudo código. El pseudo código brinda una descripción en alto nivel del programa. En el contexto del PSP_{VDC} consideramos que la especificación formal de los métodos de las clases debe realizarse antes que el pseudo código, por lo que eliminamos la actividad de pseudo código de la fase de diseño e incorporamos una fase nueva de pseudo código a posteriori de la revisión de la especificación formal.

De esta forma mantenemos la idea de un proceso de desarrollo de software bien definido donde la especificación formal realizada es tomada como entrada para realizar el pseudo código y luego éste pseudo código es entrada para la codificación.

Luego de la nueva fase de pseudo codigo en PSP_{VDC} se propone una fase de revisión del pseudo código manteniendo los lineamientos de PSP en lo que refiere a la detección temprana de defectos. El agregado de estas nuevas fases se presenta en la figura 9.

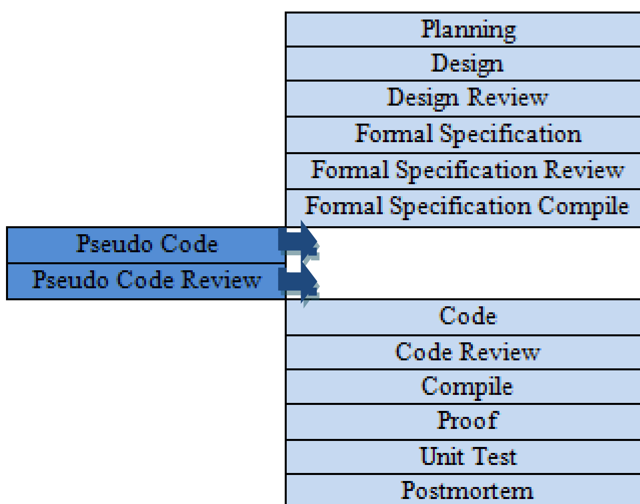

Fig. 9. Incorporando las fases de Pseudo Code y Pseudo Code Review

C. Construcción de los Casos de Prueba

En el PSP la actividad de construcción de los casos de prueba no está claramente definida en los scripts. Durante los cursos del PSP algunos instructores optan por realizar la construcción de los casos de prueba durante la fase de Design, mientras que otros instruyen realizarlos durante la fase de Unit Testing. Esto no permite conocer claramente el tiempo dedicado a diseñar y ejecutar las pruebas, pudiendo estar contenido como tiempo parcial de Diseño más tiempo de Unit Test o exclusivamente como tiempo de Unit Test. En el PSP_{VDC} nos interesa tener claramente diferenciado el tiempo dedicado a la construcción de los casos de pruebas ya que ayudará a conocer y mejorar el proceso. Por lo tanto, decidimos agregar al PSP_{VDC} una nueva fase dedicada a la construcción de los casos de prueba. Esta fase se realiza inmediatamente después de la fase de Design Review. Los casos de prueba buscan probar la especificación informal del programa a construir utilizando el diseño para poder construir las pruebas.

La figura 10 muestra el agregado de la fase de Test Case Construct al PSP_{VDC}.

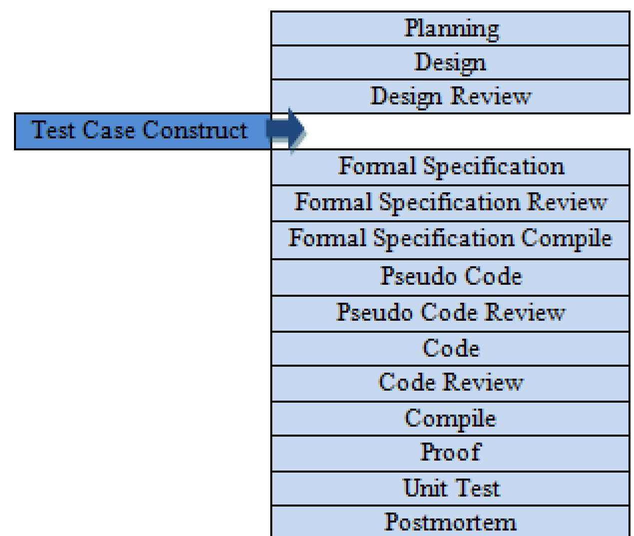

Fig. 10. Incorporando la fase de Test Case Construct

Luego de analizar los resultados experimentales de aplicar PSP_{VDC} se podría mostrar (o no) que la inclusión de VDbC reduce significativamente los defectos que llegan a UT, con lo cual podríamos pensar (analizar costo-beneficio) en eliminar las fases de Test Case Construct y Unit Test del PSP_{VDC}.

D. Las Fases del PSPVDC

El PSP_{VDC} incorpora nuevas fases y modifica otras como fuimos describiendo en las secciones anteriores. La figura 11 ilustra todas las fases del PSP_{VDC}; las cuales se describen a continuación.

Planning

Las actividades a realizar en esta fase son las mismas que en PSP; obtener los requerimientos, estimar el tamaño del producto, estimar el esfuerzo y estimar los defectos.

Design

Durante el diseño se define la estructura del programa, así como las clases, métodos, interfaces, componentes e interacciones entre ellos.

El PSP propone cuatro vistas y templates para mantener la información de un buen diseño. Una de estas cuatro vistas es la interna-estática en la cual se define el pseudo código del programa. En PSP_{VDC} el pseudo código se pospone a una fase posterior, por lo que se elimina de la fase de diseño la utilización del Logic Template utilizado en PSP para el pseudo código.

Además, durante el diseño en PSP se incluye el diseño de los casos de prueba. En PSP_{VDC} se propone realizar esta actividad en una fase posterior dedicada específicamente a generar los casos de prueba. El motivo de esta separación es nuestro interés por disponer del tiempo dedicado exclusivamente a diseñar y el tiempo dedicado exclusivamente a construir los casos de prueba.

Design Review

La revisión de diseño se mantiene incambiada con respecto a PSP, siendo su objetivo detectar defectos inyectados de forma temprana.

Test Case Construct

Se lleva a cabo luego de la revisión de diseño. Durante esta fase se construye el conjunto de casos de prueba a utilizar para probar el programa. Disponer de la información del tiempo dedicado a construir los casos de prueba y a realizar las pruebas unitarias nos permitirá evaluar el beneficio de mantener estas fases en el proceso o eliminarlas.

Formal Specification

Durante esta fase se especifican formalmente los contratos (Design by Contract). Las pre- y post- condiciones de los métodos y el invariante de la clase son especificados en un lenguaje formal.

Formal Specification Review

La revisión de la especificación formal tiene el propósito de detectar lo defectos inyectados durante la especificación formal. Se propone la utilización de una checklist para que el ingeniero revise manualmente la especificación formal en busca de posibles defectos.

Formal Specification Compile

La compilación consiste en el chequeo automático (utilizando una herramienta) de la sintaxis de la especificación formal.

Pseudo Code

Durante esta fase se escribe el pseudo código de los métodos de las clases. Se decide realizar esta fase en este momento para lograr un proceso definido en un orden natural de desarrollo de software. Proponemos realizar el pseudo código utilizando como entrada a esta fase la salida de las fases anteriores (la especificación formal revisada y compilada correctamente).

Pseudo Code Review

En esta fase se revisa el pseudo código anteriormente producido en busca de defectos. Es una actividad manual, donde el ingeniero es guiado por una checklist.

Code/Code Review/Compile

Estas tres fases se mantienen con respecto al PSP. Durante las mismas se codifica el diseño construido utilizando un lenguaje de programación, luego se realiza una revisión manual y por último se compila el código utilizando una herramienta.

Proof

La idea general de esta fase es complementar el código con una prueba formal. El objetivo es que la prueba provea evidencia de la corrección del código con respecto a las especificaciones formales (Diseño por Contrato Verificado). Esta prueba debe ser llevada a cabo con la ayuda de una herramienta.

Unit Test

Durante esta fase se ejecutan los casos de prueba construidos en busca de defectos con respecto a los requerimientos.

En el PSP_{VDC} optamos por mantener la fase de Unit Test aunque se realice una demostración formal. La demostración formal nos brinda evidencia de la correctitud del código con respecto a la especificación formal, pero nada garantiza que la especificación formal no contenga defectos. Al ejecutar los casos de prueba podemos detectar justamente si el comportamiento del programa es correcto con respecto a los requerimientos.

Postmortem

Por último, durante Postmortem el ingeniero analiza su proceso, prestando atención a los errores o dificultades en la aplicación del proceso para mejorar a futuro.

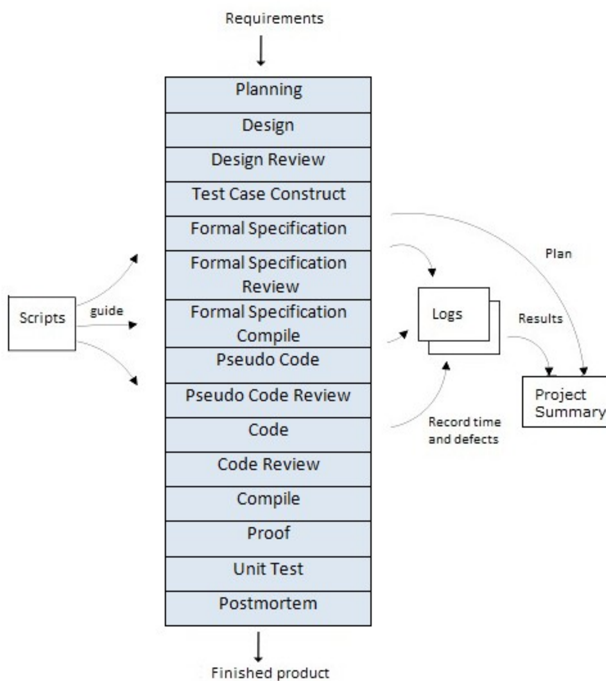

Fig. 11. Fases de PSP_{VDC}

PSP_{VDC} es guiado por un conjunto de scripts que permiten seguir el proceso de forma disciplinada y facilitan la recolección de datos. En la figura 12 se presenta el Process Script y en la figura 13 el Development Script de PSP_{VDC}.

VI. CALIDAD EN PSP_{VDC}

La calidad en PSP incluye:

- estimar el número total de defectos inyectados y removidos
- estimar el número de defectos inyectados y removidos por fase
- estimar el tiempo requerido por fase

Se presenta en esta sección las modificaciones al PSP para planificar la calidad en PSP_{VDC}.

Para estimar el número total de defectos inyectados, PSP utiliza la estimación del tamaño del programa y datos históricos a cerca de los defectos inyectados por KLOC. Para la estimación del número de defectos inyectados y removidos por fase, PSP realiza una distribución del total estimado basándose en datos históricos.

En PSP_{VDC} se deben de tener en cuenta las nuevas fases para las estimaciones de tiempo requerido por fase y el número de defectos. Inicialmente en PSP_{VDC} no se disponen de datos históricos, por lo que la estimación inicial se realiza aplicando juicio de expertos. Después de realizar varios estudios, los datos acumulados estarán disponibles para el empleo en las estimaciones deseadas.

La calidad del producto es un aspecto esencial en el PSP. Los desarrolladores deben eliminar los defectos, determinar las causas de su inyección, y aprender a evitar que ocurran nuevamente. El PSP propone revisiones como un método recomendado para la eliminación de defectos, ya que es aún más eficaz que las pruebas unitarias [12], [13], [14]. Para llevar a cabo revisiones eficientes es necesario hacer mediciones [15]. PSP define varias formas de medir la calidad y control de procesos, incluyendo las siguientes:

- *yield*
- *defect removal efficiency*
- *defect removal leverage*

El *yield* para una fase es definido como el porcentaje de defectos encontrados en la fase en relación al número total de defectos que llegan a la fase. Es una medida usualmente utilizada para conocer la efectividad de las revisiones de diseño y código así como la compilación y el *testing*.

En PSP_{VDC} puede utilizarse para medir la efectividad de las nuevas fases de formal specification review (FSR), pseudo code review (PCR), formal specification, compile, y proof.

El *yield* del proceso es calculado como el porcentaje de defectos inyectados y removidos antes de la primera compilación. En PSP_{VDC} ajustamos el cálculo teniendo en cuenta las nuevas fases que preceden a la fase de compilación.

En PSP, el *Defect removal efficiency* es un indicador del numero de defectos removidos por hora en las fases de Design Review, Code Review, Compile, y Test. En PSP_{VDC} es importante conocer el número de defectos removidos por hora en las fases de Formal Specification Review, Pseudo Code Review, Formal Specification Compile (FSC), y Proof (PRF).

El *Defect removal efficiency* para estos casos se define:

$$\text{Defect removal efficiency } (FSR) = 60 \cdot \frac{\text{Defects removed in FSR}}{\text{Time in FSR (minutes)}}$$

$$\text{Defect removal efficiency } (FSC) = 60 \cdot \frac{\text{Defects removed in FSC}}{\text{Time in FSC (minutes)}}$$

$$\text{Defect removal efficiency } (PCR) = 60 \cdot \frac{\text{Defects removed in PCR}}{\text{Time in PCR (minutes)}}$$

$$\text{Defect removal efficiency } (PRF) = 60 \cdot \frac{\text{Defects removed in PRF}}{\text{Time in PRF (minutes)}}$$

El indicador *Defect removal leverage* es el numero de defectos removidos por hora en una fase con respecto a una fase base. Normalmente la fase base es Unit Test (UT). En PSP_{VDC} proponemos incorporar los indicadores DRL (FSR/UT), DRL (PCR/UT), DRL (FSC/UT), y DRL (PRF/UT), que corresponde al número de defectos removidos

por hora en las fases de FSR, PCR, FSC, y PRF respectivamente, con respecto a la fase UT.

El Cost of quality (COQ) es una medida de la calidad del proceso. Los componentes del COQ son los costos por fallo, evaluación y prevención. El costo por fallo (*failure cost*) es el tiempo dedicado a reparar y el re-trabajo, que se corresponde en PSP a las fases de Compile y Testing. El costo de evaluación (*appraisal cost*) es el tiempo empleado en inspeccionar, que en PSP se corresponde con las fases de Design Review y Code Review. Por último, el costo de prevención (*prevention cost*) es el tiempo dedicado a la identificación y resolución de las causas de los defectos.

Siguiendo la misma idea, en el PSP_{VDC} el costo por fallo corresponde al tiempo dedicado a las fases de Code, Compile, Formal Specification Compile, Proof, y Testing. El costo de evaluación, por otro lado, es el tiempo empleado en las fases de Design Review, Code Review, Formal Specification Review y Pseudo Code Review.

El indicador Appraisal Cost of Quality (% Appraisal COQ) es definido en PSP como el porcentaje del tiempo empleado en las fases de Design Review y Code Review respecto al tiempo total de desarrollo. Altos valores de este indicador están asociados a una baja cantidad de defectos en testing y por lo tanto una alta calidad del producto.

En PSP_{VDC}, modificamos el indicador incorporando el tiempo empleado en la fase de Formal Specification Review y Pseudo Code Review. La forma de calcularlo es:

$$\%\ \text{Appraisal COQ} = 100 \cdot \frac{\begin{array}{l}\text{Design Review Time} + \\ \text{Code Review Time} + \\ \text{FSR Time} + \\ \text{PCR Time}\end{array}}{\text{Total Development Time}}$$

El indicador Percent Failure COQ (% Failure COQ) es definido en PSP como el porcentaje de tiempo empleado en las fases de Compile y Testing respecto al tiempo total de desarrollo.

En PSP_{VDC} modificamos el indicador con el propósito de incorporar el tiempo dedicado en las fases de Formal Specification Compile (FSC) y el tiempo empleado en la fase de Proof. El nuevo cálculo es:

$$\%\ \text{Failure COQ} = 100 \cdot \frac{\begin{array}{l}\text{Code Compile Time} + \\ \text{Test Time} + \\ \text{FSC Time} + \\ \text{Proof Time}\end{array}}{\text{Total Development Time}}$$

Una medida útil del COQ es la tasa entre los costos de evaluación y fallo (A/FR). Dicho indicador es modificado de forma implícita en PSP_{VDC} debido a los cambios A y FR.

En PSP, un valor de A/FR superior a 2 es considerado de alta performance. Este valor de benchmark deberá ser ajustado en PSP_{VDC} luego de realizar estudios empíricos debido al posible impacto de las nuevas fases.

Purpose	To guide the development of module-level programs
Entry Criteria	- Problem description - PSP Project Plan Summary form - Size Estimating template - Historical size and time data (estimated and actual) - Time and Defect Recording logs - Defect Type, Coding, and Size Counting standards - Stopwatch (optional)

Step	Activities	Description
1	Planning	- Produce or obtain a requirements statement. - Use the PROBE method to estimate the added and modified size and the size prediction interval of this program. - Complete the Size Estimating template. - Use the PROBE method to estimate the required development time and the time prediction interval. - Complete a Task Planning template. - Complete a Schedule Planning template. - Enter the plan data in the Project Plan Summary form. - Complete the Time Recording log.
2	Development	- Design the program. - Document the design in the design templates. - Review the design and fix and log all defects found. - Design Test cases. - Formally specify the methods of every class introduced at design. - Review the formal specification and fix and log all defects found. - Compile the formal specification and fix and log all defects found. - Write down the pseudo code, using the Logic Template. - Review the pseudo code and fix and log all defects found. - Implement the design. - Review the code and fix and log all defects found. - Compile the program and fix and log all defects found. - Construct a formal proof of correctness of the code with respect to its formal specification. - Test the program and fix and log all defects found. - Complete the Time Recording log.
3	Postmortem	Complete the Project Plan Summary form with actual time, defect, and size data.

Exit Criteria	- A thoroughly tested program - Completed Project Plan Summary form with estimated and actual data - Completed Size Estimating and Task and Schedule Planning templates - Completed Design templates and Formal Specification Templates - Completed Design Review, Formal Specification Review, Pseudo Code Review, and Code Review checklists - Completed Test Report template

	- Completed PIP forms - Completed Time and Defect Recording logs

Fig. 12. Process Script

Purpose	To guide the development of small programs
Entry Criteria	- Requirements statement - Project Plan Summary form with estimated program size and development time - For projects lasting several days or more, completed Task Planning and Schedule Planning templates - Time and Defect Recording logs - Defect Type standard and Coding standard

Step	**Activities**	**Description**
1	Design	- Review the requirements and produce an external specification to meet them. - Complete Functional and Operational Specification templates to record this specification. - Produce a design to meet this specification. - Record the design in Functional, Operational, and State templates. - Record in the Defect Recording log any requirements defects found. - Record time in the Time Recording log.
2	Design Review	- Follow the Design Review script and checklist and review the design. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
3	Test Case Construct	- Design test cases and record them in the TestReport. - Record time in the Time Recording log.
4	Formal Specification	- Implement the design following the Formal Specification standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
5	Formal Specification Review	- Follow the Formal Specification Review script and checklist and review the specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
6	Formal Specification Compile	- Compile the formal specification until there are no compile errors. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.
7	Pseudo Code	- Produce a Pseudo Code to meet the design. - Record the design Logic Specification templates. - Record in the Defect Recording log any defects found. - Record time in the Time Recording log.
8	Pseudo Code Review	- Follow the Pseudo Code Review script and checklist and review the specification.

		- Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
9	Code	- Implement the design following the Coding standard. - Record in the Defect Recording log any requirements or design defects found. - Record time in the Time Recording log.
10	Code Review	- Follow the Code Review script and checklist and review the code. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
11	Compile	- Compile the program until there are no compile errors. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
12	Proof	- Construct a formal proof of correctness of the program with respect to the formal specification. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log.
13	Test	- Test until all tests run without error. - Fix all defects found. - Record defects in the Defect Recording log. - Record time in the Time Recording log. - Complete a Test Report template on the tests conducted and the results obtained.

Exit Criteria	- A thoroughly tested program that conforms to the Coding standard - A formal specification conforming to the Formal Specification Standard - Completed Design and Formal Specification templates - Completed Design Review, Pseudo Code Review, Formal Specification Review and Code Review checklists - Completed Test Report template - Completed Time and Defect Recording logs

Fig. 13. Development Script

VII. CONCLUSIONES Y TRABAJOS A FUTURO

El desarrollo de software es una actividad creativa e intelectual realizada por seres humanos. Es común que durante un proyecto el equipo de desarrollo cometa errores. Esto se debe tanto a la complejidad actual del software como a la propia naturaleza humana. Normalmente estos errores terminan en defectos en el producto de software y cuando el software se está ejecutando estos pueden causar fallos.

La búsqueda de formas de desarrollar software con bajo número de defectos ha dado lugar al desarrollo de un variado número de procesos y métodos. El cometido de dichos procesos es construir software de calidad, en el plazo estipulado y dentro de los costos previstos.

En este artículo presentamos un nuevo proceso de desarrollo de software que combina los enfoques del proceso de desarrollo personal (PSP) y la técnica de diseño por contrato verificado (VDbC).

En el PSP_{VDC} se proponen fases nuevas para dar soporte al VDbC. El objetivo es que el uso de este nuevo proceso logre productos de mejor calidad que los desarrollados con el PSP.

Además, se presenta una revisión sistemática de la literatura que resume la información existente sobre adaptaciones realizadas al PSP y particularmente aquellas que incorporan métodos formales.

Conocer las diferentes adaptaciones realizadas al PSP que incorporar métodos formales brinda conocimiento acerca de las formas posibles de combinar el enfoque de PSP y métodos formales.

Nuestro trabajo a futuro consiste en realizar experimentos controlados que nos permitan comparar la calidad y productividad del PSP y del PSP_{VDC}. Se deberá realizar una buena planificación teniendo en cuenta el armado de los cursos de PSP, $PSP_{VDC,}$ métodos formales y de técnicas de verificación de programas, además se deberá capacitar a los sujetos en la aplicación de los procesos y herramientas de soporte. El análisis de resultados de los experimentos nos permitirá conocer cómo funciona el PSP_{VDC} en la práctica y así poder mejorarlo.

REFERENCES

[1] Ian Sommerville. Ingeniería del Software 7a edición. University of Lancaster. ISBN 8478290745, 2005

[2] Watts S. Humphrey. PSP: A Self-Improvement Process for Software Engineers. Addison-Wesley Professional; March. 2005.

[3] Watts S. Humphrey. Tsp (sm) Coach Dvlp Teams. Pearson Education, 2006

[4] Meyer, Bertrand. “Applying Design by Contract,” IEEE Computer 25, 10 (October 1992): 40-51.

[5] Hoare, C.A.R. “An Axiomatic Basis for Computer Programming,” Communications of the ACM 12, 10 (1969): 576-580.

[6] Gary T. Leavens and Yoonsik Cheon, Design by Contract with JML, 2006

[7] Kitchenham, Guidelines for performing Systematic Literature Reviews in Software Engineering, Version 2.3, EBSE Technical Report EBSE-2007-01

[8] Abdul Babar, John Potter. Adapting the Personal Software Process (PSP) to Formal Methods. Australian Software Engineering Conference (ASWEC'05), 2005, pp.192-201.

[9] Hisayuki Suzumori, Haruhiko Kaiya, Kenji Kaijiri, VDM over PSP: A Pilot Course for VDM Beginners to Confirm its Suitability for Their Development, 27th Annual Inter-national Computer Software and Applications Conference, 2003

[10] Kusakabe, S., Omori, Y. and Araki K, A Combination of a Formal Method and PSP for Improving Software Process: An Initial Report. TSP Symposium 2012, St. Petersburg, FL.

[11] Williams, L. Integrating pair programming into a software development process. Software Engineering Education Conference, Proceedings Jan 1, 2001, p27-36, 10p

[12] Hayes, William; & Over, James. Personal Software Process (PSP): An Empirical Study of the Impact of PSP on Individual Engineers (CMU/SEI-97-TR-001). Soft-ware Engineering Institute, Carnegie Mellon University, 1997. http://www.sei.cmu.edu/library/abstracts/reports/97tr001.cfm

[13] Vallespir, Diego and Nichols, William. “Analysis of Design Defects Injection and Removal in PSP,” 19-25. Proceedings of the TSP Symposium 2011: A Dedication to Excellence. Atlanta, GA, Sept. 2011.

[14] Vallespir, Diego and Nichols, William. “An Analysis of Code Defect Injection and Removal in PSP,” 3-20. TSP Symposium 2012 Proceedings (CMU/SEI-2012-SR- 48 015). Software Engineering Institute, Carnegie Mellon University, 2012. http://www.sei.cmu.edu/library/abstracts/reports/12sr015.cfm

[15] Gilb, Tom and Graham, Dorothy. Software Inspection. Addison-Wesley, 1994 (ISBN 978-0201-631814).

Silvana Moreno. Es Magister en Informática por la Universidad de la República, Facultad de Ingeniería. Es Profesora Asistente del Área de Ingeniería de Software en la Universidad de la República. Sus intereses en investigación son los procesos de desarrollo de software y su combinación con los métodos formales y cómo se pueden combinar estos enfoques para aumentar la calidad de los productos.

Álvaro Tasistro. Es Doctor en Ciencia de la Computación por la Universidad de Gotemburgo. Actualmente dirige la Cátedra de Teoría de la Computación y el grupo de investigación en Computación Teórica de la Escuela de Ingeniería de la Universidad ORT Uruguay. Sus áreas de interés son la Lógica y la Teoría de la Programación.

Diego Vallespir. Es Profesor Adjunto de la Facultad de Ingeniería de la Universidad de la República (UdelaR), Uruguay, es Director del Centro de Posgrados y Actualización Profesional en Informática de la UdelaR, es Director del Grupo de Investigación en Ingeniería de Software de la UdelaR. Es Ingeniero en Computación, Magister en Informática y Doctor en Informática por la Facultad de Ingeniería, UdelaR. Tiene varios artículos publicados en conferencias internacionales. Sus áreas de interés en investigación son ingeniería de software empírica, procesos de desarrollo de software y pruebas de software. Su hobby es divertirse.