دانشگاه پیام نور
دانشکده فنی و مهندسی
گروه مهندسی کامپیوترو فناوری اطلاعات

ارائه‌ی مدلی برای جنبه کاوی با استفاده از ماشین خودکار محدودیت

نگارش:
بهنوش امینی

استاد راهنما:
دکتر حسین شیرازی

استاد مشاور:
دکتر طاهره یعقوبی

پایان‌نامه
برای دریافت درجهٔ کارشناسی ارشد
در رشتهٔ مهندسی‌کامپیوتر - گرایش نرم‌افزار

بهمن ماه ۱۳۹۱


بسم الله الرحمن الرحيم

قدردانی

به پاس احترام به دانش و صاحبان آن، بدین‌وسیله لازم می دانم از زحمات و راهنمایی‌های استاد بزرگوارم، جناب اقای دکتر شیرازی، که در طول دوره کارشناسی ارشد از دانش، بزرگواری و مهربانی ایشان بهره‌مند شدم قدردانی نمایم و هم‌چنین از زحمات سرکار خانم دکتر یعقوبی در جهت راهنمایی این پروژه تشکر نمایم.

هم‌چنین لازم می دانم تا از زحمات استاد ارجمندم جناب آقای دکتر پوروطن برای ارائه راهنمایی های ارزنده در طول انجام این پایان نامه قدردانی نمایم. همچنین از اعضاء خانواده ام و کلیه عزیزانی که به نحوی در انجام این پایان نامه مرا یاری نموده اند تشکر می کنم.

تقدیم به

پدر بزرگوار و مادر مهربان و همسر عزیزم

اعلان منحصر به فرد بودن پایان نامه

بدینوسیله اعلان می گردد که مطالب مندرج در پایان نامه تا کنون برای دریافت هیچ گونه مدرکی توسط اینجانب و فرد دیگری ارائه نشده است.

بهنوش امینی

امضاء

چکیده

شی‌گرایی متداولترین رویکرد توسعه‌ی سیستم‌های نرم‌افزاری می‌باشد، اما ناتوانی این روش در پیمانه‌بندی کانسرن‌های متقاطع، مشکل اصلی تمامی سیستم‌های شی‌گرا می‌باشد. تکنولوژی جنبه‌گرا این امکان را با فراهم نمودن یک واحد به نام جنبه مهیا ساخته است. یکی از بحث‌های اصلی جنبه-گرایی، فرآیند جنبه‌کاوی است. گرچه دیدگاه‌های جنبه‌کاوی بسیاری تاکنون ارائه شده است ولی هیچ یک از این دیدگاه‌ها نتوانستند کانسرن‌های متقاطع را به صورت خودکار و بدون نیاز به درگیری برنامه‌نویس شناسایی نمایند. وابستگی به قواعد نام‌گذاری و اتلاف زمان بسیار برای حذف نمودن نتایج پیشنهادی اشتباه توسط ابزار از دیگر مشکلات دیدگاه‌های موجود است. مطالعه‌ی همه‌ی این موارد زمینه ساز ایجاد روشی جدید در این پایان‌نامه گردید. هدف روش پیشنهادی در این پژوهش این است که راهکاری مناسب جهت مدلسازی مطلوب برای جنبه‌کاوی ارائه نموده که به طور خودکار و بدون نیاز به درگیری کاربر و مستقل از قواعد نام‌گذاری بتواند کانسرن‌های متقاطع را شناسایی و سپس آن کانسرن را استخراج نماید که این کار در قالب ارائه‌ی سه روش انجام گرفت. در روش پیشنهادی اول با ارائه‌ی الگوریتمی به پیدا نمودن کانسرن متقاطع در صورت یکسان بودن نام متغیر های دو ماشین خودکار محدود شده‌ی ورودی پرداخته گردید که این الگوریتم قادر به یافتن کانسرن مشترک در صورتی که رفتار کانسرن مشترک مساوی، اما نام متغیرهای کانسرن مشترک در دو ماشین متفاوت باشد نبود. لذا با توسعه‌ی روش پیشنهادی اول، در روش پیشنهادی دوم، با ارائه‌ی الگوریتمی به ترفیع مشکل روش پیشنهادی اول پرداخته شد که با مشکل داشتن مرتبهِ زمانی بالا در یافتن کانسرن مشترک مواجه بود. بنابراین در روش پیشنهادی سوم، روش پیشنهادی بهینه، با ارائه‌ی الگوریتم جدیدی به کاهش مرتبه‌ی زمانی در یافتن کانسرن مشترک پرداخته شد. سپس، روش‌های پیشنهادی در محیط جاوا شبیه سازی شده و نمونه‌ای از مثال تجربی به ابزار شبیه سازی شده، داده و کانسرن مشترک آن استخراج گردیده است. جهت اعتبار سنجی نتایج به دست آمده، این نتایج با نتایج حاصل از اجرای الگوریتم به صورت دستی مقایسه گردیده است. در پایان جهت تایید بهینه بودن روش پیشنهادی سوم، از لحاظ مرتبه‌ی زمانی نتایج این روش با روش پیشنهادی تکمیل یافته مقایسه گردیده است.

کلمات کلیدی

کانسرن‌های متقاطع، جنبه‌کاوی، ماشین خودکار محدود شده، ریو

فهرست مطالب

موضوع **صفحه**

فهرست جداول

موضوع صفحه

فهرست شکلها

موضوع صفحه

فصل اول

مقدمه

۱-۱- مقدمه

توسعه‌ی نرم‌افزار، همواره با تغییرات بسیاری همراه بوده است. یکی از ابتدایی ترین روش‌های ارائه شده برنامه‌نویسی رویه‌ای است. برنامه‌نویسی رویه‌ای بر روی فرآیند یا الگوریتم حل مسئله تأکید دارد. در این روش مسئله با استفاده از روش‌های بالا به پایین[۱] و پایین به بالا[۲] حل می‌شود و یک مسئله‌ی بزرگ به مسائل کوچک‌تر تقسیم می‌شود و سپس با اجرای پی‌درپی رویه‌ها، مسئله‌ی مذکور حل می-شود. پس از برنامه‌نویسی شی‌گرا، نوبت به برنامه‌نویسی رویه‌ای رسید. عناصر برنامه‌نویسی شی‌گرا به جای رویه و تابع، از اشیاء استفاده می‌کنند. در برنامه‌نویسی شی‌گرا، سیستم شامل اشیاء در حال تعامل است که این اشیاء داده‌ها و عملیات روی آن داده‌ها را محافظت می‌نمایند[۳]. در این روش اشیاء در سلسله مراتب کلاس دسته‌بندی می‌شوند. هر کلاس شامل مجموعه‌ای از خصوصیات است که شامل داده‌ها و عملیاتی است که رفتار این کلاس را تعریف می‌نمایند. محافظت نمودن داده‌ها و عملیات روی این داده‌ها منجر به اثرات جانبی کمتر در زمان ایجاد تغییرات شده و هم‌چنین امکان استفاده‌ی مجدد از اشیاء را آسان‌تر می‌نماید. بعد از برنامه‌نویسی شی‌گرا، برنامه‌نویسی جنبه‌گرا[۴] ارائه شد. برنامه‌نویسی جنبه‌گرا توسعه یافته‌ی زبان‌های شی‌گرایی و رویه‌ای است. برنامه‌نویسی جنبه‌گرا، روشی برای پیمانه‌بندی[۵] مشخصاتی از سیستم که نمی‌توانند در رویه یا کلاس محافظت شوند، فراهم می‌نماید. یکی از مباحث اصلی در زمینه‌ی جنبه گرایی، جنبه‌کاوی است. اگرچه در سال‌های اخیر تکنیک‌های بسیاری برای جنبه‌کاوی پیشنهاد شده است ولی هر کدام دارای محاسن و معایبی می-باشند و هر یک از دیدگاه‌های ارائه شده سعی در بهبود نتایج دیدگاه قبلی خود دارند اما هیچ یک از دیدگاه‌های ارائه شده نتوانستند روشی ارائه نمایند که تمامی کانسرن‌های متقاطع را به صورت خودکار و بدون نیاز به در گیری کاربر شناسایی نمایند. وابستگی به قواعد نام‌گذاری و اتلاف زمان بسیار برای حذف نمودن نتایج پیشنهادی اشتباه توسط ابزار از دیگر مشکلات دیدگاه‌های موجود است. مطالعه‌ی تمامی این موارد زمینه ساز ایجاد روشی جدید در پژوهش حاضر گردید. هدف این پژوهش ارائه‌ی روشی خودکار برای جنبه‌کاوی در جهت رفع مشکلات دیدگاه‌های پیشین است که در ادامه ابتدا به توضیح تعریف کلیدی از کانسرن[۶] پراخته می‌شود.

---

[1] Top Down
[2] Buttom Up
[3] Encapsoulate
[4] Aspect Oriented Programing
[5] Modularity
[6] Concern

۱-۲- تعریف مساله

تشخیص خودکار عناصر برنامه از پیچیده ترین فعالیتهای طراحی و ساخت نرم افزار بوده که تشخیص آنها گرچه با توصیه‌هایی در مهندسی نرم افزار همراه می‌باشد، ولی تعریف آنها به مهارت‌های فردی سازنده نرم افزار بستگی دارد. این مهم زمانی مسئله ساز خواهد بود که ویژگی‌هایی از جمله کارآیی، سهولت در پشتیبانی و یا دیگر عوامل کیفی نرم افزار به حدی حائز اهمیت باشند که انتخاب سطحی از هم‌بستگی در ساخت محصول موثر باشد. در این حالت تکیه بر مهارتهای فردی، ضامن نیل به محصول با کیفیت نخواهد بود. دراین پژوهش با ارائه‌ی روشی ریاضی عناصر برنامه به طور خودکار تعریف شده تا طراحی به مطلوب‌ترین شیوه و مستقل از توانایی‌ها و خلاقیت‌های فردی پایان بپذیرد. در این راستا سوالات زیر مطرح‌اند:

۱- آیا روش‌های موجود برای تشخیص جنبه ها نیاز سازندگان نرم افزار ها را پاسخگو هستند؟

۲- آیا روشی خودکار و عام برای تشخیص جنبه ها وجود دارد؟

۳- در صورتی‌که کانسرن‌های متقاطع الگویی برای تشخیص جنبه ها باشند، چگونه این کانسرن-ها بایستی شناسائی شوند؟

۴- روشهای مدل سازی مطلوب برای برنامه‌نویسی جنبه گرا چیست؟

۵- چگونه می‌توان از روشهای فرمال و ریاضی برای تشخیص کانسرن‌های متقاطع سود برد؟

۶- آیا میتوان الگوریتمی ارائه نمود تا به بهترین حالت کانسرن های متقاطع را با استفاده از روش‌های ریاضی تشخیص دهد؟

۷- آیا می‌توان پس از تشخیص کانسرن‌های متقاطع، واحد حاوی آن را به طور خودکار استخراج نمود؟

لذا تلاش شده که در این پژوهش به پاسخ‌گویی سوالات بالا پرداخته شود. قبل از ادامه‌ی پژوهش در ابتدا لازم است که با مفهوم کانسرن و جنبه آشنا شده که در ادامه ابتدا به این تعاریف می‌پردازیم.

کانسرن جنبه‌ای از مسئله است که برای ذینفع[۷] یا ذینفعانی بحرانی[۸] یا مهم باشد[Kiczales, et al., 2009]. هنگامی که در مورد کانسرن‌ها صحبت می‌شود باید دو جزء اصلی که در این تعریف برای هر کانسرن ارائه شده است در نظر گرفته شوند:

۱. جنبه‌ای از یک مسئله

۲. علاقه‌ی ذینفعان (یعنی هدف[۹] آنان)

برای روشن شدن این تعریف، مثالی ارائه می‌گردد. آیا زیر سیستمی به نام x در سیستم نرم افزاری داده شده، یک کانسرن است؟

[7] Stakeholder
[8] Critical
[9] Goal

۱. آیا زیرسیستم x یک جنبه از مسئله را نشان می‌دهد؟

- خیر، زیرسیستم x بخشی از راه حل نرم‌افزاری است و خود مسئله نیست.

۲. آیا زیر سیستم x مورد علاقه و دارای اهمیت نزد برخی از ذینفعان سیستم می‌باشد؟

- بله، زیرا هر بخشی از راه حل سیستم مورد علاقه و دارای اهمیت نزد حداقل یک ذینفع می‌باشد.

پس زیرسیستم x یک کانسرن نیست. این نتیجه را می‌توان به این نکته تعمیم داد که یک مصنوع[۱۰] در توسعه‌ی سیستم کانسرن نمی‌باشد [Kiczales, et al., 2009].

اغلب کانسرن‌ها با نیازها اشتباه گرفته می‌شوند، اگرچه این دو با یکدیگر متفاوتند [Filman, 2001]. نیازمندی‌ها اغلب مقداری از جنبه‌هایی از یک مسئله‌ی توسعه‌ی نرم افزاری را بیان می‌نمایند. برخی از جنبه‌های مسئله در طول چرخه‌ی عمر نرم‌افزار مطرح می‌شوند. برای مثال وقتی که یک نرم افزار شی‌گرا توسعه داده می‌شود، مسائلی در سطوح مختلف تحلیل، طراحی، پیاده‌سازی و غیره ممکن است پیش آیند. راه حل یک مسئله‌ی تحلیل می‌تواند یک مسئله طراحی را بیان نماید که این خود می‌تواند باعث مطرح شدن یک مسئله‌ی پیاده‌سازی شود. جنبه‌های مختلف این مسائل به خصوص جنبه‌های مسائل طراحی و پیاده‌سازی به طور حتم در نیازمندی‌های سیستم منعکس نشده‌اند با وجود این‌که آنها هرکدام یک کانسرن هستند[Filman, 2001].

سیستم نرم‌افزاری مجموعه‌ای از کانسرن‌هاست. کانسرن‌های مدیریت حساب، محاسبه‌ی بهره، تراکنش‌های درون بانکی، ماندگاری همه‌ی موجودیت‌ها[۱۱]، مجاز شناسی دسترسی به سرویس‌های مختلف[۱۲]، قابلیت ردیابی[۱۳]، قابلیت نگه داری[۱۴] و قابلیت درک[۱۵] نمونه‌هایی از کانسرن‌های یک سیستم بانکداری هستند. کانسرن‌ها به دو گروه تقسیم می‌شوند: کانسرن‌های اصلی که کارکرد اصلی یک پیمانه[۱۶] را شامل می‌شوند و کانسرن‌های سیستمی که نیازهای جانبی و نیازهای در سطح سیستم هستند که بین چندین پیمانه توزیع می‌شوند [O'Regan, 2004].

جنبه‌گرایی مجموعه‌ای از تکنولوژی‌ها با هدف فراهم نمودن جداسازی بهتر کانسرن‌ها می‌باشد که به عنوان جداسازی پیشرفته‌ی کانسرن‌ها نیز شناخته می‌شود و به ما کمک می‌کند که به پیمانه‌بندی بهتری در توسعه‌ی نرم‌افزار دست یابیم. جنبه‌گرایی با تکنیک‌های موجود، مانند شی‌گرایی رقابت نمی‌کند بلکه بر روی آنها بنا شده است و با پیمانه‌بندی کانسرن‌های متقاطع و بهبود توصیف نحوه‌ی تأثیر آن کانسرن‌ها برهم مکمل آنها می‌باشد [Laddad, 2003].

---

[10] Artifact
[11] Persistence
[12] Authorization
[13] Traceability
[14] Maintainability
[15] Comprehensibility
[16] Module

برنامه نویسی جنبه‌گرا در سال ۱۹۹۷ توسط آقای گریگورکیزالس و گروهش معرفی شد [Kiczales, et al., 2009]. این رویکرد در حقیقت به عنوان یک مکمل برای برنامه‌نویسی شی‌گرا عرضه شد تا ضعفی که در قسمت قبل به آن اشاره شد را برطرف نماید. در واقع هدف برنامه‌نویسی جنبه‌گرا بهبود پیمانه‌بندی سیستم با تمرکز بر پیاده‌سازی پیمانه‌ای کانسرن‌های متقاطع می‌باشد. فکر اصلی پشت سر برنامه‌نویسی این است که کانسرن‌ها به نحوی پیمانه‌ای می‌شوند که برنامه‌نویس صرفاً متمرکز بر پیاده‌سازی هر کانسرن در هر پیمانه باشد. درک ساز وکار اصلی کارکرد یک برنامه موضوع بسیار مهمی است که با وجود کانسرن‌های درهم‌پیچیده با آن معمولاً بسیار سخت انجام می‌شود زیرا تمام کد اصلی برنامه با کد کانسرن‌های دیگر آمیخته شده است که علاوه بر مشغول ساختن ذهن برنامه نویس و کند کردن فرآیند اشکال‌زدایی باعث می‌شود تا درک روند کاری اصلی که نیز برای کسی که سعی در فهم آن دارد بسیار مشکل شود. اما راه حل گروه کیزالس برای این موضوع چه بود؟ راه کاری که آنها عرضه داشتند می‌توانست هر کانسرن متقاطع را در یک پیمانه‌ی جداگانه مورد بحث قرار دهد. پیمانه‌ای که علاوه بر عملیات اصلی که باید انجام دهد، شرط و محل اجرای این عملیات را نیز در بر می‌گیرد. مثلاً برای امنیت راه حل ما در مدل شی‌گرایی ساختن یک کلاس (یا یک روش در یک کلاس) برای بررسی دسترسی کاربر مورد نظر است. پس از ساخت یک ماژول برای بررسی کاربر، با اضافه کردن شرط‌ها و عبارت‌های فراخوانی آن در قسمت‌های مختلف برنامه، کار را تکمیل می‌کنیم. اما راه حل برنامه نویسی جنبه‌گرا متفاوت است. در برنامه نویسی جنبه‌گرا کد مربوط به کانسرن متقاطع مورد نظر را در یک پیمانه‌ی جداگانه نوشته و سپس در همان پیمانه شرط‌ها و محل فراخوانی این کد را نیز بیان می‌کنیم (به عنوان مثال، بررسی هویت[17] در مواردی انجام شود که یک روش امن فراخوانی می‌شود). بدین ترتیب تمام روند کاری کانسرن‌های متقاطع از سازوکار اصلی برنامه مجزا می‌شود. کاملاً مشخص است که این تکنیک می‌تواند چه مزیت‌هایی را برای سیستم به ارمغان بیاورد. به عنوان مثال به خصوص در نرم افزارهای بزرگ می‌توان این دو بخش کد را (کد اصلی و کد کانسرن‌های متقاطع) را به دو گروه مختلف برنامه‌نویسی واگذار کرد یا حتی درباره خود کانسرن‌های متقاطع را می‌توان هر بخش را به متخصص آن سپرد. مثلاً بخش بررسی کردن امنیت را به متخصصان امنیت یا بخش تراکنش‌های پایگاه داده[18] را به متخصصان آن واگذار نمود. ممکن است به نظر برسد که تمامی این کارها با مدل شی‌گرا نیز قابل انجام است البته درست است اما باید توجه داشت که در یک مدل شی‌گرا برنامه‌نویس باید در حین توسعه‌ی یک کانسرن متقاطع از نکات زیر آگاهی داشته باشد :

۱. برنامه‌نویس باید از وجود کلاس‌هایی که برای پیاده‌سازی کانسرن‌های متقاطع ساخته شده است خبر داشته باشد.

[17] Authentication
[18] Database Transaction

۲. باید مشخصات دقیق آن کلاس را بداند تا بتواند استفاده کند.
۳. باید بداند که متدهای آن کلاس را کجای کد اصلی و تحت چه شرایطی فراخوانی نماید.

در تمام این سه مرحله امکان رخ دادن خطا وجود دارد. به خصوص در قسمت آخر که فراموشی برنامه نویس برای فراخوانی تمام متدهای لازم از شایع‌ترین اشتباهات است. اما با استفاده از برنامه نویسی جنبه‌گرا با جدا شدن نمونه‌های برنامه میان این دو بخش و تمرکز آنها بر توسعه و تست هر کانسرن به صورت جداگانه چنین مشکلاتی اصولاً مطرح نمی‌شوند. به علاوه قابلیت فهم، نگه‌داری، کنترل و استفاده‌ی مجدد از کد بیشتر می‌شود [Laddad, 2003]. مشهورترین زبان برنامه‌نویسی جنبه‌گرا Aspectj است که به دلیل محدودیت این پایان نامه به آن نمی‌پردازیم. یکی از مباحث اصلی در زمینه‌ی جنبه گرایی، جنبه‌کاوی است که شرح کامل آن در فصل دوم آورده شده است و پژوهش اصلی ما نیز در این راستا انجام شده است.

۱-۳- ضرورت انجام تحقیق

شی‌گرایی یکی از متداول‌ترین رویکردهای توسعه‌ی سیستم‌های نرم افزاری می‌باشد اما ناتوانی در پیمانه‌بندی کانسرن‌های متقاطع[۱۹]، مشکل تمامی سیستم‌های شی‌گرا می‌باشد. وقتی می‌گوییم یک کانسرن از نوع متقاطع است این بدان معناست که آن کانسرن، پیمانه‌بندی[۲۰] غالب سیستم را قطع می‌نماید. یک برنامه و سیستم نرم‌افزاری با استفاده از مولفه‌هایی (مانند کلاس‌ها در شی‌گرایی) پیمانه‌بندی می‌شود. پیاده‌سازی انواع بسیاری از کانسرن‌ها که تعدادشان نیز زیاد است با آن نحوه‌ی پیمانه‌بندی تطبیق ندارند و در نتیجه منجر به دو پدیده‌ی درهم پیچیدگی[۲۱] و پراکندگی[۲۲] می‌شوند. چنین کانسرن‌هایی، کانسرن‌های متقاطع نامیده می‌شوند. پیاده‌سازی یک کانسرن، پراکنده شده است هرگاه که آن بین چند پیمانه پخش شده باشد. هم‌چنین هرگاه که یک کانسرن با کد کانسرن دیگری آمیخته شده باشد، می‌گوییم پیاده‌سازی آن کانسرن‌ها درهم پیچیده شده است. پراکندگی و درهم پیچیدگی علائم وجود کانسرن‌های متقاطع می‌باشد. این کانسرن‌ها معمولاً نمی‌توانند در یک پیمانه به طور کامل محافظت شوند. برای مثال اعمال یک سیاست امنیت[۲۳] را در نظر بگیرید. ماهیت کانسرن امنیت به نحوی است که بسیاری از واحدهای پیمانه بندی کاربرد را قطع می‌کند. هم چنین سیاست امنیت باید به طور یکنواخت هنگامی که سیستم تکامل می‌یابد و رشد می‌کند بر هر افزایش

[۱۹] Crosscutting Concerns
[۲۰] Modularity
[۲۱] Tangling
[۲۲] Scattering
[۲۳] Security

سیستم اعمال شود. گذشته از این، سیاست امنیت که در پیمانه‌های مختلف کاربرد اعمال می‌شود خود نیز تکامل می‌یابد و بدین ترتیب باید تمامی آن پیمانه‌ها نیز به هنگام شوند. پیاده‌سازی کانسرن-هایی مانند یک مکانیزم بر قراری امنیت به طرز پیمانه‌ای در یک زبان برنامه‌نویسی سنتی[۲۴](مثال شی‌گرایی) دشوار و مستعد خطا می‌باشد. کانسرن‌هایی مانند امنیت، واحدهای طبیعی پیمانه‌بندی سیستم را قطع می‌کنند این واحدها در زبان‌های برنامه‌نویسی شی‌گرا کلاس‌ها می‌باشند. در این زبان-ها کانسرن‌های متقاطع به راحتی به صورت کلاس پیمانه بندی نمی‌شوند زیرا کدی که این کانسرن‌ها را محقق می‌نماید، در کلاس‌های سیستم پخش شده و با کد کانسرن نظیر هر کلاس درهم‌پیچیده می‌شود، لذا پیاده‌سازی کانسرن‌های متقاطع قابل استفاده مجدد نمی‌باشد، قابل پالایش و ارث بری نمی‌باشد و در سراسر برنامه به صورت بی‌نظم پخش شده است و به طور خلاصه پرداختن به آنها و نگه‌داری آنها به خصوص هنگامی که مقیاس سیستم بزرگ است دشوار می‌باشد. پیاده‌سازی کانسرن-های متقاطع مرحله‌ای است که دیگر مدل شی‌گرا جواب کارآمدی به ما نمی‌دهند زیرا پیاده‌سازی کانسرن‌های متقاطع با کانسرن‌های اصلی ترکیب می‌شود. متدلوژی‌های شی‌گرایی می‌توانند به خوبی کانسرن‌های اصلی را پیمانه‌بندی کنند اما این متدلوژی‌ها در پیمانه‌بندی نمودن کانسرن‌های سیستمی با شکست مواجه می‌شوند زیرا فضای پیاده‌سازی یک فضای یک بعدی است یعنی جریان پیوسته‌ای از فراخوانی‌هاست پس تمرکز اصلی‌اش روی پیاده‌سازی کانسرن‌های اصلی است و پیاده-سازی کانسرن‌های متقاطع با آنها ترکیب می‌شود و در اثر آن، پدیده‌ی پراکندگی کانسرن و درهم پیچیدگی کانسرن‌ها رخ می‌دهد که تبعات آن را در ادامه خواهیم گفت. شیوه‌های پیمانه‌بندی موجود مانند کلاس‌ها و مولفه‌ها به ما کمک می‌کنند که کانسرن‌ها را جدای از یکدیگر پیاده‌سازی کنیم اما تمام این روش‌های موجود در رابطه با پیمانه‌بندی کانسرن‌های متقاطع ناکافی هستند و همراه شاهد این هستیم که قسمت‌های زیادی از سیستم شامل تکه کدهای مربوط به واقعه نگاری[۲۵]، ماندگاری[۲۶]، توزیع‌شدگی، کنترل خطا[۲۷]، هم گام سازی[۲۸] و امثال اینها می‌باشد.

پراکندگی و درهم پیچیدگی کانسرن‌ها نشانه‌ی آن است که پیاده‌سازی کانسرن‌ها به خوبی پیمانه‌بندی نشده است. چنین کانسرنی در واقع واسط[۲۹]های خوش تعریفی[۳۰] را ارائه نکرده است و در نتیجه ارتباط متقابل بین پیاده‌سازی آن و سایر پیمانه‌های سیستم به وضوح تعریف نشده است. در واقع آن کانسرن به صورت ضمنی از طریق وابستگی‌ها و ارتباط بین قسمت‌های کدی که آن را پیاده-سازی می‌کند و پیاده‌سازی سایر پیمانه‌ها محقق گشته است. نبود واسطه‌های خوش‌تعریف بین پیاده-

[۲۴] Traditional
[25] Logging
[26] Persistence
[27] Error handling
[28] Synchronization
[29] Interface
[30] Well Defined