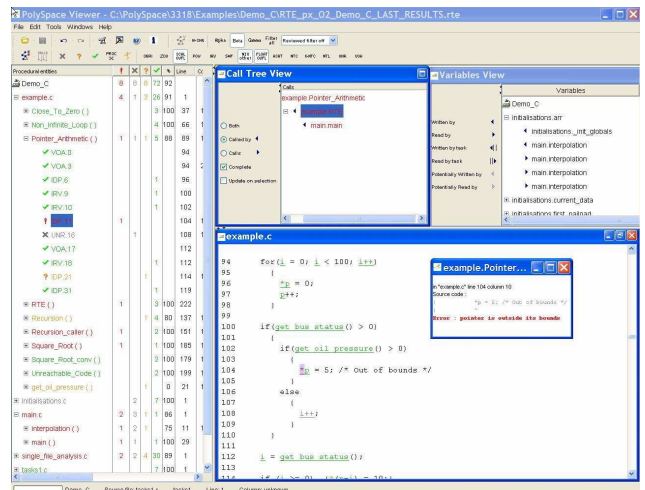


## Statische Code-Analyse mit PolySpace

|                           |   |
|---------------------------|---|
| <b>Kunde</b>              | Wärtsilä  |
| <b>Ziel-System</b>        | Embedded Systeme und<br>PC-Applikationen  |
| <b>Programmiersprache</b> | C / C++<br>Matlab-Simulink/Stateflow  |
| <b>Technologie</b>        | Test und Verifikation   |
| <b>Speziell</b>           | Frühzeitige Detektion von<br>Laufzeitfehlern (ohne<br>Ausführung des SourceCodes) |



Quelle: TheMathworks

### Aufgabe

Sotronik erhielt den Auftrag, die Software für die elektronische Steuerung eines Grossdieselmotors auf mögliche Laufzeitfehler hin zu testen. Da die Software sehr umfangreich ist (>130'000Zeilen ANSI-C-Code), wurde nach einem Werkzeug gesucht, das durch geeignete Automatismen den Testaufwand minimiert und gleichzeitig zuverlässige Ergebnisse liefert. Nach einer Evaluationsphase entschied man sich schliesslich für das statische Code-Verifikations-Tool *PolySpace* der Firma *TheMathworks*.

### Eigenschaften von PolySpace

*PolySpace* bietet statische, code-basierte Software-Verifikation, die auf der Methode der abstrakten Interpretation beruht. Mit dieser Methode versetzt *PolySpace* den Benutzer in die Lage die Abwesenheit von bedingten Laufzeitfehlern nachzuweisen. Hierbei bietet das Werkzeug auch Unterstützung in Bezug auf die Industriestandards IEC61508 und DO-178.

In der folgenden Tabelle sind Ursachen für potentielle Laufzeitfehler aufgelistet, die mit *PolySpace*, vor der eigentlichen Ausführung des SourceCodes auf dem Zielsystem, detektiert werden.

| Detektierte Fehler in C / C++              | in C++ zusätzlich   |
|--|---|
| Out of Bound Array Index                   | Function Returns a Value or raises an exception (FRV)                     |
| Not Initialized Return Value               | Strictly Positive Array Size (PAS)  |
| Illegal Dereference Pointer                | Pure Virtual Call (PVC)   |
| Shift Error                                | Non Null Receivers (NNR)  |
| Negative Power                             | Correct Type for Receiver (CTR)   |
| Arithmetic Exception Error                 | Correct Type for Pointer to member function (CTP)                         |
| Zero Division                              | Non Null Pointers to members (NMP)  |
| Not Initialized Variable Local             | Incorrect type id argument (TID)  |
| Not Initialized Variable Global            | Incorrect dynamic_cast calls (DCT)  |
| Assertion Error                            | Unexpected exception raised (EXU)   |
| Float Overflow/Underflow                   | Throws during catch parameter construction (THR)                          |
| Scalar Overflow/Underflow                  | Logical & Runtime exceptions (LOE/RUE)                                    |
| Not Initialized Pointer                    | exception not raised in destructor (EXD)                                  |
| Non-terminating Call                       | Call never raises an exception (CRE/FRE)                                  |
| Non-terminating Loop                       | main, tasks and C standard library functions do not raise exception (PCF) |
| Zusätzlich:<br>Unreachable Code (Coverage) |   |

Quelle: TheMathworks

Die Verifikation des SourceCodes kann erweitert werden, so dass auch beispielsweise die Einhaltung von MISRA-C-2004-Rules berücksichtigt wird und *PolySpace* je nach Konfiguration mit Fehlermeldungen oder Warnungen auf jeweilige Regelverstösse reagiert.

Getestet wird auf Modulebene (nur wenige C-Funktionen) oder auf der Integrationsebene (mehrere SW-Module mit zahlreichen C/C++ Funktionen). Anomalien in Multitasking-Systemen, die durch Task-Interaktionen entstehen, können hierbei aufgedeckt werden.

### Arbeiten mit PolySpace

Der SourceCode-Entwickler startet direkt im Anschluss an die Implementierung den PolySpace-Test für das neu erstellte Software-Modul. Hierbei hat er die Möglichkeit, nicht zur Verfügung stehende Funktionen, die in diesem Modul benutzt werden, von Hand zu stubben oder automatisch Stub-Funktionen durch PolySpace erstellen zu lassen.

Je nach Codegrösse kann schon nach wenigen Minuten das Testresultat begutachtet werden. PolySpace unterteilt hierbei den getesteten Code farblich in vier Kategorien:

- rot**  
 ...für definitive Laufzeitfehler (das Verhalten der Software an dieser Stelle ist nicht definiert!).
- grau:**  
 ...für nicht erreichbaren Code ("toter Code" im Sinne, dass diese Zeilen nie aufgerufen werden und daher entfernt werden sollten).
- grün:**  
 ...für korrekte Operationen (es treten in keinem Fall Laufzeitfehler auf).
- orange:**  
 ...für potentielle Laufzeitfehler (bei gewissen Konstellationen kann es zu Laufzeitfehlern kommen).

P  
r  
o  
v  
e  
n

```

static void Pointer_Arithmetic (void)
{
    int array[100];
    int i, *p = array;

    for(i = 0; i < 100; i++, p++)
        *p = 0;

    if(get_bus_status() > 0) {
        if (get_oil_pressure() > 0)
            *p = 5;
        else
            i++;
    }

    i = get_bus_status();
    if (i >= 0) { *(p-i) = 10; }

    if ((0 < i) && (i <= 100)) {
        p = p - i;
        *p = 5;
    }
}
            
```

Quelle: TheMathworks

### Nutzen für die Software-Entwicklung

Nach der Einführung von PolySpace in den fortlaufenden Entwicklungsprozess wurden schon bald die Vorteile der Nutzung des Tools ersichtlich. Diese drückten sich insbesondere in den nachfolgend aufgeführten Punkten aus:

- Produktivität und Qualität:**  
 Der Einsatz von PolySpace fördert klare und robuste Programmierweise und daher die bessere Les- und Wartbarkeit neuer SourceCode-Module. Laufzeitfehler werden frühzeitig erkannt und potentiell kritische Bereiche während der automatisierten Tests kenntlich gemacht.
- Zeitersparnis:**  
 "grün"-markierte Zeilen benötigen bezüglich Syntax und Softwarestruktur kein CodeReview mehr. Das bei anderen Tools meist aufwendige Erstellen von Stub-Funktionen kann entfallen. White-Box Tests können bis zu 75% reduziert werden.

### Grenzen

Eigenschaften der Zielhardware werden bei einem Test durch PolySpace nicht berücksichtigt. So sind beispielsweise direkte Hardwarezugriffe auf Register eines Mikrocontrollers per Compile-Switch auszuschalten, was meist eine Änderung des SourceCodes erfordert. Auch können dynamische Stack-Überläufe einer Software oder direkte Linker-Anweisungen nicht getestet werden.