Enhancing qDebug()

Arnaud.Clere @ MinMaxMedical.com

Ideas issuing from ModMed: an applied research project to MODel and verify MEDical Cyber-Physical Systems

http://modmed.minmaxmedical.com/

VASCO
Research Team

Technology
Provider

Medical Devices
Manufacturer

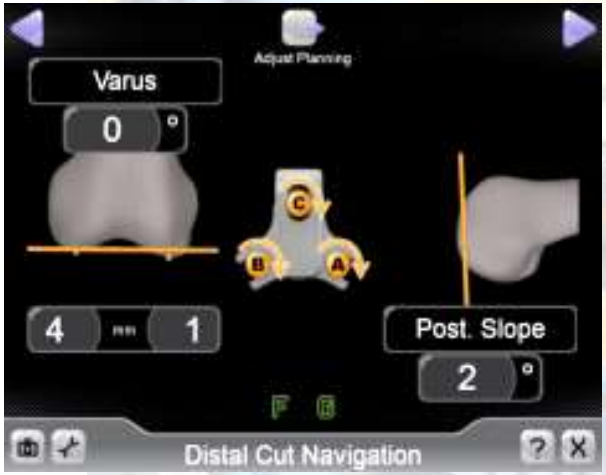
ANR-15-CE25-0010

Agenda

Qt in the Operating Room

Medical Cyber-Physical System =

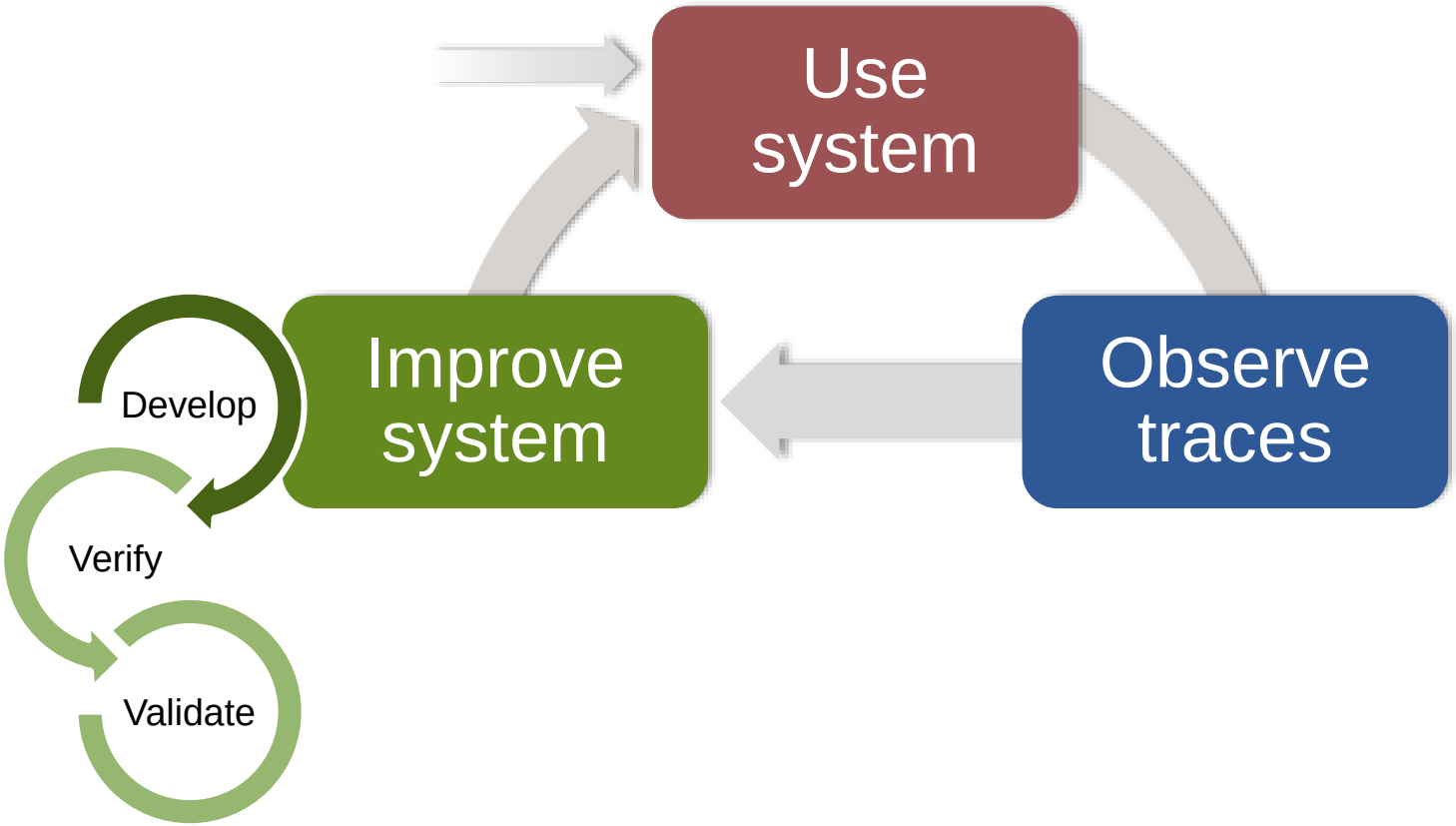
Computing power + UI



+ Sensors (3D camera)

[+ Medical images, Robots, iPhones]

Importance of Traces

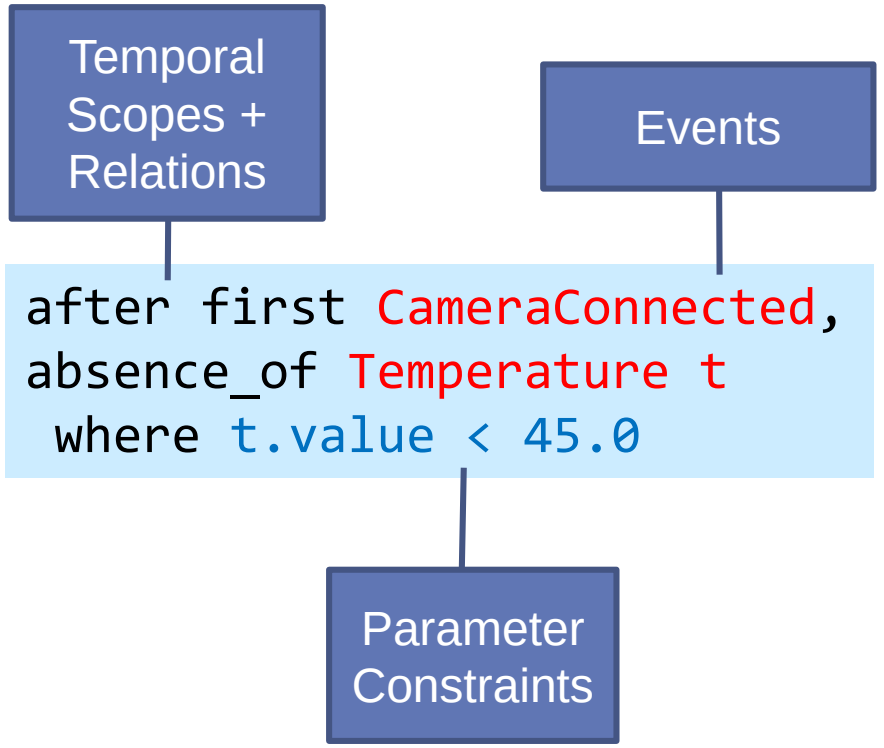


Simple Trace Property

Problem: *3D camera localizations' precision is affected by camera's temperature* → **Patient Risk**

ModMed approach:

1. Formalise (i.e., code) a temporal property (ParTraP language)
2. Evaluate it on traces



Motivating Results

Property:

The surgeon never skips a screen

Analysis of 1000 real-world surgeries' traces:

1. 35% DO NOT satisfy this property

2. After traces analysis and property refinement,
   still 10% of problematic traces that may reveal either:
   - A tool deficiency → **Redesign**
   - An incorrect tool handling → **Training**

Problems

```
...
MSG 2015.11.11-03:07:09.969 | [EventHandler::performStateExit] Exiting state : mainCasp.sTec
MSG 2015.11.11-03:07:09.979 | [EventHandler::performStateEntry] Entering state : mainCasp.sA
MSG 2015.11.11-03:07:09.979 | [App::enableTracking] 0 mSec to enable tracking
MSG 2015.11.11-03:07:09.979 | [KneeOptimizer::prepositionImplant_boneOnly_noAntCtx] load Tar
MSG 2015.11.11-03:07:09.979 | [KneeOptimizer::computeFemurImplantCut] initial Transfo = 1 0
MSG 2015.11.11-03:07:09.979 | [KneeOptimizer::computeFemurImplantCut] target =
MSG 2015.11.11-03:07:09.979 |     VV     0.00 DEG of VARUS / meca axis
MSG 2015.11.11-03:07:09.979 |     slope  3.00 DEG of POSTERIOR
MSG 2015.11.11-03:07:09.979 |     distCH 10.00 mm, MEDIAL VALID
MSG 2015.11.11-03:07:09.979 |     axRot  0.00 DEG of EXT / Whiteside line
MSG 2015.11.11-03:07:09.979 |     notch  999.00 mm of INVALID
MSG 2015.11.11-03:07:09.979 | ---------------------- iteration #0
MSG 2015.11.11-03:07:09.979 | --- Optimization Gives Position #0
MSG 2015.11.11-03:07:09.979 | Varus    : 0.00 DEG of VARUS / meca axis
MSG 2015.11.11-03:07:09.979 | Slope    : 0.00 DEG of ANTERIOR
MSG 2015.11.11-03:07:09.979 | DistCutM : 18.33 mm, MEDIAL VALID
MSG 2015.11.11-03:07:09.979 | AxialRot : 5.49 DEG of INT / Whiteside line
MSG 2015.11.11-03:07:09.979 | Notching : 999.00 mm of INVALID
MSG 2015.11.11-03:07:09.979 | ML error : 0.00(0.00)
...
```

Quick, can you check something for me on femur distal cut optimization?!

Our approach

Trace analysis problems have been tackled in many ways:

syslog, journald, ETW, log4cxx, CLF, CEE,
« Aspect-Oriented-Programming »,
« process mining », « source repository mining » …

Our approach is simple but original:

1. **Put the source code structure and meaning** into traces, rather than rediscover/reinvent it
2. Allow gradual improvement/addition of trace points

➢ **We suggest various enhancements** to

Qt Logging Framework

Use namespace as category

Easily declare a `QLoggingCategory` named after the current namespace to increase category usage and consistency:

```
M_NAMESPACE_LOG_CATEGORY()
void test() { mNDebug() << "category is ''"; }

namespace foo {

    M_NAMESPACE_LOG_CATEGORY()
    void test() { mNDebug() << "category is 'foo'"; }

    namespace bar {

        M_NAMESPACE_LOG_CATEGORY()
        void test() { mNDebug() << "category is 'foo.bar'"; }

    }
}
```

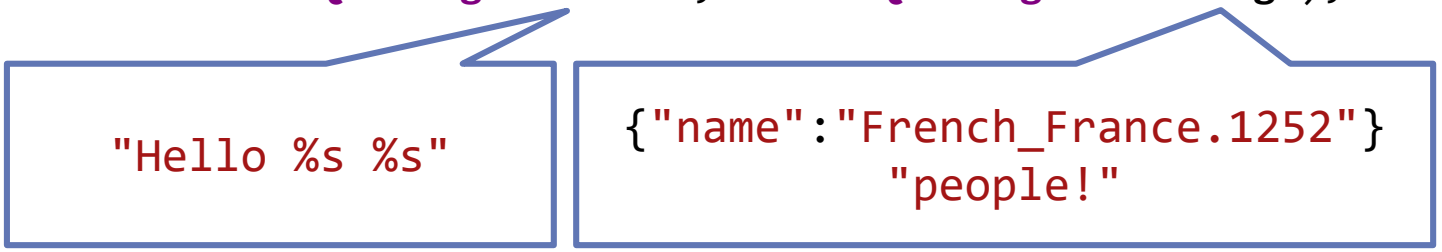
Better use QDebug <<

QDebug << brings type safety but **hides the pattern** (format):

```
mNDebug("C-style is %s %s", "unsafe", "not extensible");
mNDebug() << "Hello" << myLocale << QString("people!");
```

- Keep the constant (abstract) « format » and concrete args apart:

```
typedef void (*QtMessageHandler2)
             (QtMsgType, const QMessageLogContext &,
              const QString & format, const QStringList & args);
```



NB: Format and data may still be merged for console or the existing `QtMessageHandler` and its outputs

Better use source code (1)

Developers are spoiled for choice when writing messages and the resulting traces lack uniformity:

- Provide **macros for tracing variables** using standard “formats”

```
void Service::process(QString input)
{
    mNTraceQObject2(this, input);
    mNTraceOut3(m_submitted, m_rejected, m_processed);
    mNAssumption(QThread::currentThread()==thread()); // not thread-safe!
    ++m_submitted;
    if (!mNAssumption2(input.size() <= 32, input)) {
        ++m_rejected; emit rejected();
    } else {
        // ++m_processed; // uncomment to fullfill requirements
        QString result(input.repeated(2));
        mNRequirement2(result.size() == 2*input.size(), input);
        mNRequirement2(result.contains(input), input);
        emit processed(result, input);
    }
}
```

Better use source code (2)

- Resulting in the following structured trace:

#Trace acme::Service %s objectName %s input %s	0x20083f017d0		.
#Trace **#Assumption #Failure** QThread::currentThread()==thread()			
#Trace **#Finished** m_submitted %s m_rejected %s m_processed %s	0	0	0
#Trace acme::Service %s objectName %s input %s	0x20083f017d0		..
#Trace acme::Service %s objectName %s input %s	0x20083f017d0		....
#Trace acme::Service %s objectName %s input %s	0x20083f017d0		.....
#Trace acme::Service %s objectName %s input %s	0x20083f017d0		.....
#Trace acme::Service %s objectName %s input %s	0x20083f017d0		.....
#Trace acme::Service %s objectName %s input %s	0x20083f017d0		.....
#Trace **#Assumption #Failure** input.size() <= 32 input %s	.....		
#Trace #Finished m_submitted %s m_rejected %s m_processed %s	6	0	0
#Trace #Finished m_submitted %s m_rejected %s m_processed %s	6	0	0
#Trace #Finished m_submitted %s m_rejected %s m_processed %s	6	0	0
#Trace #Finished m_submitted %s m_rejected %s m_processed %s	6	0	0
#Trace #Finished m_submitted %s m_rejected %s m_processed %s	6	0	0
#Trace #Finished m_submitted %s m_rejected %s m_processed %s	6	0	0
#Trace **#Requirement #Failure** m_submitted == m_processed + m_rej	0	1	7

Promote TSV+JSON format

1. **TSV**: line-separated events, tab-separated event context & data
2. **JSON** cells: human-readable, unambiguous, universal standard

_elapsed_s	*[_date_time]*	*[_sev*	*_category*	*[_function]*	*_format*	*_0*	*_1*
0,0026734	2017-10-04T16:20	7		int __cdecl main(int,cha	#Trace QString(argv[0]) %s	C:\\ACL\\modmed_trunk\\modm	
0,0028207					C-style logging is %s and %s	not type-safe (m	not extensible to
0,0028847					Hello %s %s	{"bcp47Name":"	people!
0,0030116		6			started demonstration to the users		
0,0041482		4			unexpected or badly evolving condition: fi	171939	
0,0042169		2			failure affecting the user: %s	unable to make coffee!	
0,0042805		7			#Trace md::Hex(&sfile) %s	0x845573f4c0	
0,0043184					#Trace myLocale %s	{"bcp47Name":"fr","uiLanguages"	
0,0044172				void __cdecl print<int>	#Trace toPrint %s	10	
0,004563				void __cdecl print<clas	#Trace toPrint %s	plop	
1,03935		7	acme	void __cdecl acme::Ser	#Trace acme::Service %s objectName %s	0x2699c5cba70	

Very simple, adapted to **human exploration** & **tool analysis**

- TSV facilitates classification using format and other metadata
- `TsvJsonLogOutput` facilitates exploration by eliminating useless redundancy in metadata columns

Easily log user classes

Define an **internal `bind()` method:**

```
struct Person {
    QString m_firstName, m_lastName; int m_yearOfBirth; Meters m_height;

    template<Operator Op>
    void bind(Item<Op> i) { // provides fluent interface + error-handling
        i.record()
            .sequence("name")
                .bind(m_firstName)    // calls pre-defined Bind<QString>
                .bind(m_lastName)
                .out()                // to go on with enclosing record()
            .number("birth_year", m_yearOfBirth)
            .bind("height", m_height) // calls user-defined Bind<Meters>
            ; // automagically closes every data structure
    }
};
```

Alternatively, define an **external `Bind<_,T>` functor**

Use a *universal* data model

JSON is a good candidate *(implemented in so many languages)*

… except strings may be used to avoid incorrect transmission or interpretation of **unusual numbers**: long long long integers, NaN, arbitrary precision decimals, etc.

Item

Simple data which text representation has implicit meaning

```
"John Doe", 1953, "-Infinity", "1.78m", "8/5/1945"
```

Sequence: 0-n **Item** (fixed order, variable number)

```
["John Doe","Jane Doe"]
```

Record: 0-n **Item** (indexed, variable number)

```
{"name":"John Doe","birth_year":1953,"height":"1.78m"}
```

Offer various formats for free

- **JsonString / Writer / Reader**

```
JsonString().write().bind(person)
{"name":["John","Doe"],"birth_year":1953,"height":"1.78m"}
JsonReader(&json).read().bind(person) // fills person
```

- **SummaryString / Writer** for console output

```
SummaryString().write().bind(person)
{name:[John | Doe] | 1 more | height:1.78m}
```

- **XmlString / Writer** for the plethora of available tools

```
XmlString().write().bind(person)
<r><s key="name"><t>John</t><t>Doe</t></s>
   <t key="birth_year">1953</t><t key="height">1.78m</t></r>
```

Provide a format-free handler

Necessary to **format depending on output**

```
typedef void (*QtMessageHandler3)
             (QtMsgType, const QMessageLogContext &,
              const QString & format, const Qlist<BindableRef> & args);
```

where `BindableRef` implements a dynamic **`IBindable`** interface:

```
template<typename T>
class BindableRef : public IBindable {
public:
    BindableRef(const T& data);
    virtual void bind(Item<Writer> writer);
        { Bind<Writer,T>::bind(writer, const_cast<T&>(data)); }
protected:
    const T& m_data;
};
```

Performance vs Completeness problem

Trace points are **too sparse** and **incomplete**

Tracing **decreases** performance

The good balance remains a **compromise**

In our domain, CPU is cheap compared to the price of missing information from device use in the field:

- **Our priority** is to facilitate adding and improving trace points

Also, `mNRequirement` / `mNAssumption` trace points are ideal since they only log runtime violations (complimentary to `Q_ASSERT`)

Offer smart trace points

```
for (int i=0; i<250; ++i)
    mNTraceModulo(100, i);

QElapsedTimer t; t.start(); const int ms = 1000000;
for (double elapsed(0); elapsed<1000*ms; elapsed=t.nsecsElapsed())
{
    mNTraceNSecs(20*ms, sin(elapsed/(100*ms)));
    QThread::msleep(19); // slightly faster than sampling
}
```

Automatic id and **count**

Sampling based on

- Occurrences (modulo)
- Elapsed time

_ elapsed	_ id	[_ count]	_format	_0
0,014131	**F**	**0**	#Trace **i** %s	0
0,014839	F	100	#Trace i %s	100
0,015558	F	200	#Trace i %s	200
0,015929	**G**	**0**	#Trace **sin(elapsed**/(100*ms)) %s	0
0,054359	G	2	#Trace sin(elapsed/(100*ms)) %s	0,37453
0,093352	G	4	#Trace sin(elapsed/(100*ms)) %s	0,699056
0,131907	G	6	#Trace sin(elapsed/(100*ms)) %s	0,916585
0,170248	G	8	#Trace sin(elapsed/(100*ms)) %s	0,999612
0,209286	G	10	#Trace sin(elapsed/(100*ms)) %s	0,935
0,248376	G	12	#Trace sin(elapsed/(100*ms)) %s	0,729497
0,287888	G	14	#Trace sin(elapsed/(100*ms)) %s	0,40984
0,327864	G	**16**	#Trace sin(elapsed/(100*ms)) %s	0,022715
0,347957	G		#Trace sin(elapsed/(100*ms)) %s	-0,1 743
0,387887	G	**19**	#Trace sin(elapsed/(100*ms)) %s	-0,54611

So...

modmedLog traces library (Qt-based) and **ParTraP** tool prototype can be reviewed and evaluated at:

https://forge.imag.fr/projects/modmed/

MinMaxMedical (Arnaud Clère, Vivien Delmon, Manon Linder) can spend some time to submit some Qt logging enhancements

What do you think?

Dynamically structured types

The fluent interface can also be used as a structured iterator with serializability guarantees

```
void bindQtDictAsRecord(Item<Writer> i, TDictionary& qtDict)
{
    Record<Writer> r(i.record());
    for (auto it = qtDict.begin(); it != qtDict.end(); ++it)
    {
        r.bind(it.key(), it.value());
    }
}
```

BTW: Current Qt types are already bound in modmedLog

Simple user types

Simple types may need external `Bind<_>` specializations

`Bind<Writer,_>` are usually trivial

```
struct Meters { float meters; };

template<>
void Bind<Writer,Meters>::bind(Item<Writer> item, Meters& value) {
    item.bind(QString::number(value.meters,'f',2) + MetersTag);
}

template<>
void Bind<Reader,Meters>::bind(Item<Reader> item, Meters& value) {
    QString atom; item.bind(atom);
    bool isFloat(false);
    if (item.state()->isOk() && atom.endsWith(MetersTag))
        value.meters = atom.left(MetersTag.size()).toFloat(&isFloat);
    if (!isFloat)
        item.state()->addError(FailBindFrom(atom, "not a number + ‘m’"));
}
```

Prevent format side effects

`QTextStream` manipulators like `QDebug` nospace() noquote() have uncontrolled side effects. `QDebugStateSaver` is useful but cannot guarantee non ambiguous parsing.

Low-level formatting can be specified using C++ types with no side effects like `modmed::data::Hex()`:

```
qint64 size;
mNDebug() << Hex(size) << size;
// logs in hexadecimal without affecting other logged data
```