Improving OAI code

- François Taburet, Laurent Thomas
- 11-12-2018
Agenda

1. Architecture proposal
   1. Executable implementation
   2. Module implementation
   3. Build system
2. What has been delivered
3. Going on with code improvement
Executable architecture

1. Main, as other functions should be kept “small”
   - Get the configuration parameters which define the executable function and trigger the required modules which have to be loaded.
   - Initializes the mandatory modules, mainly utilities
   - Initializes the functional modules

2. The executable is a module
   - Has its own internal data and provides set/get API so that other modules can possibly retrieve these parameters
   - Should be the only one using parameters which names are not prefixed

3. Run-time options should always be preferred to build-time options
   - Build time options make test complicated. Non default build time options not included in the CI are often broken and become unusable.
   - It’s a lot easier for CI to test config-time option than build-time options
   - Macro programming can become un-readable
Module architecture (1)

1. Objective: flexibility, reliability, code readability

2. A module embeds its own configuration, read at init-time from its init function.
   - Possibly externalizes get or set functions if other modules need to access these parameters

3. Possibly provides a resource cleanup API if it uses a system shared resource that other “non oai” programs may use (Ex: console for Logs, RF devices...)

4. Interface with other modules is via an API, either compiled function calls or possibly via function pointers when configuration dependent implementation options are considered
   - Switching to function pointer usage is often easy when interfaces are well defined
   - When huge volume of data are to be transferred between modules, the underlying preferred mechanism is message queues (Itti for oai). When data are local, sending meta-data is often enough. The message queue implementation is in charge of the locking mechanisms.
   - When an interface is either remote or local it can be implemented via implementation variants, chosen at run time.
   - Implementation variants can be dynamically loaded or all embedded at build time, if for example, dynamic switch is considered at run-time
Module architecture (2)

5. Multi-threaded module

- Pipeline: dedicated thread for a specific task, use itti to receive/send data from/to previous/next module
- Parallelism: a given task distributed between several threads: more complex, not suited for all tasks.
What has been delivered

1. Code cleanup
   - A lot has been done, by dedicated branch and merge requests
     - Utilities, we try to introduce common tools, removing specific implementations for functionalities everybody needs: configuration, itti, Logging.
     - macro removal
     - executable redefinition
     - large static table removed

2. Lots of new features ready for integration, even if not 100% completed (NB-lot, 5G-NR...)
   - Many of them initiated long time ago
   - Oai evolved a lot compared to initial source branch
   - Integration work might be heavy, risk of “cleanup” regressions is high
   - CI cannot prevent all of them
   - We must find a way to integrate, while preserving the progress we did in code quality
Improving while contributing (1)

1. Code cleanup can now be a shared effort, included in any contribution
   - Use or switch to common tools, don’t develop or maintain specific implementation of something any module could need
   - Enhance existing tools if a new functionality is required, and ask the community before jumping to the implementation
   - Instead of modifying code obviously violating good practices, switch to a clean implementation

2. Dead code removal
   - If(0),
Improving while contributing (2)

3. Remove compile-time options whenever possible
   - Some are useless, remove!!
     - if defined(ENABLE_ITTI: we always need itti (message queues)
   - For some of them an alternative has been developed, but are not yet deployed everywhere
     - DEBUG_XXX build time option: remove or replace by config-time option (if DEBUG_XXX((flag))
     - LOG_M: surround with a config option test (if ( DUMP_XXX(flag))
   - For others we have to find a solution
     - if (LTE_RRC_VERSION >= MAKE_VERSION(14, 0, 0)) do we need, can we support multiple release support?
     - if (S1AP_VERSION >= MAKE_VERSION(13, 0, 0) (idem, redundant?)
   - It should be possible to just keep the build-time option to remove tracing code (T-Tracer, LOGs) and get a “as performant as possible” executable.
   - Don’t introduce new compile-time option without first considering other implementation choices
Improving while contributing (3)

4. Revisit debugging code

- Some sources have redundant debugging code (`rrc_eNB.c in rrc_eNB_generate_SecurityModeCommand`)
  - `T, LOG_DUMPMYSQL, LOG_I, LOG_D, MSC_LOG_TX_MESSAGE`
- `LOG_I` is the first level of debugging trace, it should have limited impact on real-time
- `LOG_D` and `LOG_T` or conditional code blocks should be preferred whenever a high throughput of messages is expected (prach example)