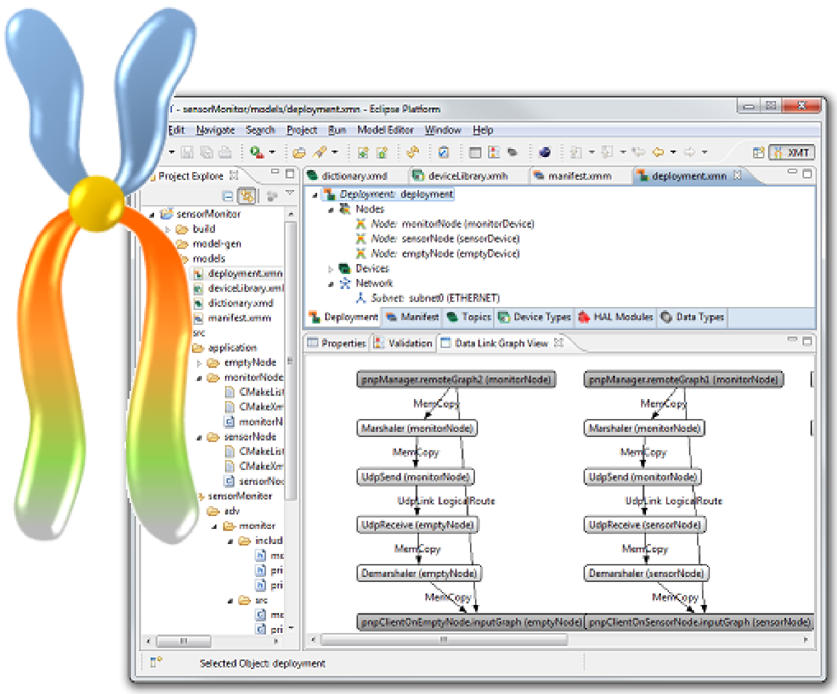
CHROMOSOME Installation & Tutorial



sensorMonitor/models/deployment.xmn - Eclipse Platform
Edit Navigate Search Project Run Model Editor Window Help
Project Explorer
dictionary.xmd
deviceLibrary.xmh
manifest.xmm
deployment.xmn
Deployment: deployment
Nodes
Node: monitorNode (monitorDevice)
Node: sensorNode (sensorDevice)
Node: emptyNode (emptyDevice)
Devices
Network
Subnet: subnet0 (ETHERNET)
Deployment
Manifest
Topics
Device Types
HAL Modules
Data Types
Properties
Validation
Data Link Graph View
pnpManager.remoteGraph2 (monitorNode)
pnpManager.remoteGraph1 (monitorNode)
MemCopy
Marshaler (monitorNode)
UdpSend (monitorNode)
UdpLink LogicalRoute
UdpReceive (emptyNode)
UdpReceive (sensorNode)
Demarshaler (emptyNode)
Demarshaler (sensorNode)
pnpClientOnEmptyNode.inputGraph (emptyNode)
pnpClientOnSensorNode.inputGraph (sensorNode)
Selected Object: deployment

fortiss GmbH

An-Institut an der
Technischen Universität München

Guerickestr. 25
80805 München

chromosome@fortiss.org

Version 0.8 RC1
March 14, 2014

Contents

1 Introduction

For a long time, the focus of embedded systems development has been the implementation of isolated systems with clearly defined boundaries and interfaces. Recently, the trend of integrating such independent systems into larger *systems of systems* arises. For example, nodes in a wireless sensor network connect *ad-hoc* to other nodes, manufacturing plants get connected with logistics and intelligent cars communicate with each other and the infrastructure. Due to the different life cycles of the involved systems, adaptability such as plug & play becomes more and more important. Systems must allow integration with each other even if the exact type and structure of their counterparts is not known at design time without losing their safety, security and real-time capabilities.

In order to achieve this, a powerful domain-independent software platform is required, which can flexibly be adapted to various application scenarios. CHROMOSOME[1] is a middleware and runtime system intended to meet these requirements. It combines features known from the embedded domain such as determinism with adaptivity known from internet technologies. CHROMOSOME treats extra-functional requirements as first-class entity and provides according mechanisms to fulfill such requirements.

CHROMOSOME has a large set of designated features and is designed to evolve over time. Its runtime system is completely open source and hence transparent to developers and end users.

1.1 In this Release

This release of CHROMOSOME consists of two main parts:

1. The CHROMOSOME (XME) runtime system with hardware abstraction layer (HAL) and platform support for Linux and Windows.

2. The CHROMOSOME Modeling Tool (XMT), an Eclipse-based model-driven design tool with automatic code generation capabilities for static configuration of the target system.

The objective of the current release is to allow users to *model the distributed system* in a graphical way in XMT and allow the modeled system to evolve at runtime using *plug and play*.

Based on the XMT model, code is generated for every device (node) in the network of the distributed application. If user-defined components are used, the user then develops the necessary code that is based on the CHROMOSOME runtime system. This makes the code portable between the various target platforms. Finally, the code is compiled using CHROMOSOME's powerful build system infrastructure that is based on CMake and transferred to the respective target systems (this is currently still a manual process). At runtime, new software components may be added to the nodes using plug and play concepts.

The described features are a subset of the features that will be available in future versions. Appendix C provides an overview of the anticipated feature roadmap and the respective timeframe.

The following tutorial will introduce you to the main concepts of CHROMOSOME and illustrate them with examples that are easy to understand. Including installation of required prerequisites, this tutorial will take about two hours to complete.

[1] http://chromosome.fortiss.org/

1.2 The Name CHROMOSOME

CHROMOSOME stands for **Cro**ss-domain **M**odular **O**perating **S**ystem **o**r **M**iddlewar**e**.[2] The name expresses the vision of CHROMOSOME:

1. We believe that in the future cross-domain solutions for networked embedded systems are required.

2. As different applications might have many different requirements and the middleware will be deployed to very heterogeneous platforms, a scalable and modular solution is required.

3. CHROMOSOME will sometimes operate on top of an operating system (OS), but it might also replace an OS due to resource constraints. We think that in the future, the boundary between an operating system and a middleware will be blurry.

Similar to biology, a CHROMOSOME instance is built up by a number of genes (here software components). It is not the intention of CHROMOSOME to invent new genes, but instead to use existing protocols/solutions to implement components for specific tasks. The major idea is that CHROMOSOME offers a flexible blueprint that enables the selection of different solutions to adapt a system to the specific requirements.

Analog to the evolution of mankind, we do *not* believe that the genes provided by CHROMOSOME are initially perfect. Instead we believe in constant improvement based on discussions about specific components. This is one of the reasons why we are offering CHROMOSOME as open-source software. In case you have any questions or suggestions for improvement, we are looking forward to your comments.

1.3 Mailing Lists

We offer two mailing lists for CHROMOSOME-related topics:

- **News and release announcements:** `chromosome-announce@lists.fortiss.org`
  Archives: `https://lists.fortiss.org/pipermail/chromosome-announce/`
  To subscribe, send an (empty) e-mail to `chromosome-announce-subscribe@lists.fortiss.org`. You will then receive an automatic e-mail with further instructions. This list is read-only.

- **Developers:** `chromosome-dev@lists.fortiss.org`
  Archives: `https://lists.fortiss.org/pipermail/chromosome-dev/`
  To subscribe, send an (empty) e-mail to `chromosome-dev-subscribe@lists.fortiss.org`. You will then receive an automatic e-mail with further instructions. To start a discussion, send an e-mail to `chromosome-dev@lists.fortiss.org`. You do not need to subscribe in order to be able to post. In this case, let other people know that you would like to be `CC:`'d in replies.

1.4 Questions and Contact Information

If you have any questions with respect to this tutorial or want to report a bug, please check the mailing list archives and the Frequently Asked Questions in Appendix B. If your question is not answered, please post to the `chromosome-dev@lists.fortiss.org` mailing list.

[2]We borrowed the "H" from somewhere else.

1.5 Acknowledgments

This work is partially funded by the following research grants:

- German Federal Ministry of Economics and Technology (BMWi) research grant 01ME12009 (project RACE[3])
- German Federal Ministry of Economics and Technology (BMWi) research grant 01MA11002 (project AutoPnP[4])
- Bavarian Ministry of Economic Affairs, Infrastructure, Transport and Technology grant programme 1330/686 (Vorlaufforschung fortiss)

[3] http://www.projekt-race.de/
[4] http://www.autopnp.com/

2 Prerequisites (30 minutes)

CHROMOSOME has been developed with platform independence in mind. This tutorial covers both Linux[5] and Windows development and target platforms. While installing the proposed build environment, you might want to start reading Section 3 giving an overview on CHROMOSOME and its features.

2.1 Prerequisite Installation on Linux

Apart from the CHROMOSOME source archive, the following packages are recommended for building CHROMOSOME applications on Linux.

- **GCC Toolchain**: GCC offers a complete compiler toolchain for various languages and is recommended for compiling C/C++ programs natively on Linux. Install it as well as some tools by issuing the following command:[6]

  ```
  sudo apt-get install gcc g++ gdb make
  ```

- **CMake** (at least version 2.8.5): CMake is a cross-platform Makefile generator and is used to manage the build system. Output of CMake are the build system configurations (here UNIX Makefiles or Eclipse CDT projects). CHROMOSOME provides a customized set of macros to deal with components, dependencies, executables and documentation. First try whether CMake is available as a package in your distribution:

  ```
  sudo apt-get install cmake
  ```

  It is also meaningful to install `ccmake`, an `ncurses`-based GUI for CMake:

  ```
  sudo apt-get install cmake-curses-gui
  ```

  If CMake is not available in your distribution, it is very easy to build and install it from source. Download the source package from the Kitware website[7] and follow the instructions. CHROMOSOME currently requires at least CMake version 2.8.5. If installation worked properly, this command should tell you the installed version of CMake:

  ```
  cmake --version
  ```

The following packages are recommended in addition if you plan to develop applications with CHROMOSOME.

- **Perl**: Various maintenance script and tools use the Perl scripting language. Even if not mandatory for CHROMOSOME, Perl might be worth a look:

  ```
  sudo apt-get install perl
  ```

- **Doxygen**: Doxygen is used to automatically generate source code-level documentation that is meaningful for users of a given API. For this purpose, the CHROMOSOME source files have been annotated with respective comments. If your distro offers the `doxygen` package, all you have to do is:

[5]When we say Linux, we actually mean the many distributions that are out there. CHROMOSOME has been specifically tested with Debian-like environments such as Ubuntu and Xubuntu. However, we believe that there are no fundamental hurdles that would prevent it from running on other Linux distributions. The CHROMOSOME runtime system by default has no GUI and should also run on plain UNIX-based systems, provided enough resources are available. For using the CHROMOSOME Modeling Tool, a graphical desktop environment is required, however. If you experience problems, or got it to work on an exotic platform, please let us know!

[6]The `apt-get` command is specific to Debian-like Linux distributions. Use the respective command on other distros.

[7]http://www.cmake.org/cmake/resources/software.html

```
sudo apt-get install doxygen
```

Otherwise, please download and install manually from the Doxygen website.[8]

- **Graphviz**: A graph drawing package. Doxygen uses this to enhance the generated documentation with inclusion graphs, dependency graphs and class diagrams:

  ```
  sudo apt-get install graphviz
  ```

  Otherwise, please download and install manually from the Graphviz website.[9]

- **Expect**: Some automation tasks for testing the integrity of our example projects interact dynamically with the respective console programs. For this purpose, `expect` is used. Install it as follows:[10]

  ```
  sudo apt-get install expect
  ```

  Otherwise, please download and install manually from the Expect website.[11]

2.2 Prerequisite Installation on Windows

Apart from the CHROMOSOME source archive, the following tools are required for building CHROMOSOME applications on Windows.

- **Visual Studio C++** (Express, Professional, Premium or Ultimate, preferably 2008 or 2010 versions): Visual Studio is Microsoft's platform for multi-language development. The so-called "Express Edition" is available free for evaluation purposes.[12] Note that CHROMOSOME has not yet been tested with newer versions of Visual Studio than the 2010 release. You only have to install the C/C++ compiler. Additional packages like Silverlight or SQL Server are not required. Consult Appendix D for details.

  In case you want to use a different compiler toolchain, you are welcome to try it out. We offer some initial support for building under Cygwin using GCC or MinGW.[13]

- **CMake** (at least version 2.8.5): CMake is a cross-platform Makefile generator and is used to manage the build system. Output of CMake are the build system configurations (here a Microsoft Visual Studio Solution). CHROMOSOME provides a customized set of macros to deal with components, dependencies, executables and documentation. CMake can be downloaded for free.[14] CHROMOSOME currently requires at least CMake version 2.8.5. It is *not* necessary to add CMake to the system search path. Consult Appendix E for details.

The following optional tools are recommended in addition if you plan to develop applications with CHROMOSOME.

- **Cygwin**: Cygwin is a UNIX emulation environment that comes with a set of useful tools. You can find the download and installation instructions on its website.[15] Although the tools are based on an emulation layer, the performance is still acceptable and maintenance is simple. The drawback is that the versions offered are typically a little bit outdated.

[8]`http://doxygen.org/`

[9]`http://www.graphviz.org/`

[10]This will also install `tcl`, the Tool Command Language. If you've never heard of it, you might want to explore it and its graphical user interface (GUI) addon `tk`. It's great for (GUI) applications that should be nice, platform independent and well scriptable at the same time.

[11]`http://expect.sourceforge.net/`

[12]`http://www.microsoft.com/visualstudio/en-us/products/2010-editions/visual-cpp-express`

[13]Let us know about your experiences. If you prefer using different compilers, we may add them to our portfolio.

[14]`http://www.cmake.org/cmake/resources/software.html`

[15]`http://www.cygwin.com/`

In case this is not acceptable, download and install the tools natively from their respective websites. It is recommended to explicitly install the following Cygwin packages (see respective items in Section 2.1 for more information):

`perl`, `expect`

- **Doxygen**: Doxygen is used to automatically generate source code-level documentation that is meaningful for users of a given API. For this purpose, the CHROMOSOME source files have been annotated with respective comments. If you choose not to install `doxygen` as a Cygwin package, please download and install manually from the Doxygen website.[16]
- **Graphviz**: A graph drawing package. Doxygen uses this to enhance the generated documentation with inclusion graphs, dependency graphs and class diagrams. Please download and install manually from the Graphviz website.[17]

[16] `http://doxygen.org/`
[17] `http://www.graphviz.org/`

3 CHROMOSOME in a Nutshell (30 minutes)

CHROMOSOME (often abbreviated by "XME") is a domain-independent, data-centric[18] middleware for cyber-physical systems. From the point of view of an application component, CHROMOSOME abstracts from basic functionality that is traditionally found in operating systems and middlewares, like scheduling and communication. Apart from that, it offers model-driven design tools with code generation capabilities that allow a user to design the distributed system in an abstract way. This section will give the reader a very high level overview of the most important CHROMOSOME features. For in-depth information and concrete specifications, the reader is kindly referred to the commented source code of CHROMOSOME.

3.1 Goals of CHROMOSOME

CHROMOSOME is designed to match the requirements of future cyber-physical systems. As a research prototype, the main intention is to serve as a basis for discussion how future middleware/operating system architectures may look like. The development is done in a demand-driven way. Depending on the requirements of projects where CHROMOSOME is applied, new features in CHROMOSOME will be introduced.

Our point of view is that future middleware will not operate on a specific class of devices, but will run on very heterogeneous platforms. The exaggerated goal statement could be to "develop a middleware serving a range from 8-bit controllers to cloud servers".[19] Hence, scalability and modularity of the middleware is of highest importance. It must be easy to remove or add components from/to the middleware. Similar to micro-kernel operating systems, the core of CHROMOSOME must be very small and only contain the absolutely necessary features. Since the boundaries of operating systems, middleware, and applications are vanishing, one unique mechanism must be supported to add components on all these levels.

Data-Centric Design A very successful concept to achieve scalability is message-orientation as for example demonstrated by the QNX[20] neutrino micro-kernel. However, one major drawback of this concept is the necessity of requiring knowledge of both the sender and the receiver. In the context of systems-of-systems where the communication partners are not known at design time, message orientation cannot be applied. Therefore, CHROMOSOME is based on the very powerful concept of data-centric design. Components specify which data they consume and produce and depending on this information, the middleware calculates the communication routes.

It is important to note that in contrast to prominent other middleware systems relying on data-centric design, CHROMOSOME does not broadcast the data, instead it calculates specific routes based on the requirements. This allows for a very efficient implementation. Furthermore, the routes are calculated only when components are added or removed. Based on the experience in embedded systems, we assume that reconfiguration takes place less often than the transmission of data items. Similar to concepts from multimedia protocols, our initial assumption is that data transmission has to happen within real-time, while reconfiguration is not time-critical or has only soft real-time requirements.

Data is grouped by so-called *topics*. A topic is a data type with a certain structure. Examples for topics are *temperature*, *rotation speed* or *GPS position*. Topics are defined for a specific domain. This enables the exchange of data between different applications. Topics are organized in so-called *topic dictionaries*. Each dictionary contains topic definitions for a specific domain and/or application.

[18] For details, please refer to http://www.omg.org/news/whitepapers/Intro_To_DDS.pdf

[19] This is a very striking motto. The functionality offered by the middleware will of course vary depending on the resource constraints of the target platform.

[20] http://www.qnx.com/

Refinement of Topics with Attributes (partially implemented) Exchange of data can only be automated if the requirements on data are specified explicitly. This includes information about the quality of data such as *age*, *accuracy*, *confidence* and *safety level*. In the respective topic dictionary, each topic can be annotated with so-called *attributes*.[21] An attribute consists of a *key* (its name) and its *value*. Depending on the type of attribute, the concrete value may be constant, known at configuration time or dynamically calculated during runtime. Using attributes, application components describe their requirements respectively guarantees on data. This information can be used to select appropriate communication streams while still retaining independence between sender and receiver.

Model of Execution (partially implemented) Correct timing is a major concern for cyber-physical systems. CHROMOSOME abstracts the concrete implementation, but currently offers mechanisms to specify schedules for time-triggered execution. In future versions, end-to-end-timing and jitter requirements for data paths will also be supported.

The reason to abstract the concrete implementation/concrete configuration in CHROMOSOME is on the one hand to reduce the development effort by automatically deriving a concrete configuration satisfying the timing requirements (if a configuration exists) and on the other hand to support plug & play capability. The state of the art in embedded system design requires the developer to configure the application in a correct way. Several configurations might be valid, but typically only one configuration is specified. It is not possible for the run-time system to change the configuration, as the real requirements are not present any more and the system can not calculate alternative configurations. By explicitly stating the requirements and leaving configuration issues to the run-time system, this problem can be avoided. The system can support new components by calculating a new configuration that satisfies both the requirements of already running applications and the newly installed applications.

It is important to note that CHROMOSOME can of course only offer guarantees that are enabled by the underlying platform. In case CHROMOSOME runs on a Linux or Windows system without real-time kernel/extensions, CHROMOSOME will not be able to satisfy jitter requirements in the range of microseconds. Nevertheless, certain implementations on top of such systems may offer very good guarantees. The major concept here is to use algorithms that offer determinism sacrificing average-case performance. Examples are the implementation of a time-triggered communication scheme on top of Ethernet. This implementation guarantees collision free communication, but comes with a lower bandwidth and flexibility.

Plug & Play One of CHROMOSOME's major goals is to allow dynamic reconfiguration of a distributed application at runtime. In order to combine this concept with real-time capabilities and determinism, we distinguish between the so-called *plug phase* and the *play phase*. The plug phase, which is entered upon startup and can on demand be triggered during runtime, is used to analyze the requirements of all components and to develop a plan for configuring the runtime system in order to meet the dependencies. The plug phase is not real-time capable, i.e., its exact duration is not guaranteed. During the plug phase, component requirements are checked, binaries are copied, data paths are calculated and the configurations for every element in the system are prepared and deployed. For this purpose, plug & play related components on all nodes in the network cooperate.

When the configuration is finished, the system transitions to the play phase. The play phase is deterministic with respect to the requirements stated by the components, such as the worst-case execution time.

Subsequent plug & play events may be triggered in the play phase, for example node plugout (i.e., the removal of nodes from the XME ecosystem) or removal of individual components.

[21] Called *meta-data* in previous versions of CHROMOSOME.

Health Monitoring (partially implemented) Although the system is deterministic during the play phase, a runtime health monitoring is required, because an application component might violate its asserted requirements. For example, an application component might violate its worst-case execution time or try to access a resource where it has no access to. Similarly, a network communication problem might occur. In such cases, the system is supposed to fall back into a safe mode in which the basic functionality of the application is retained. This allows for safe "shutdown".

3.2 Glossary

This glossary explains the most important terms in CHROMOSOME in alphabetical order.

Component A component is a modular software unit that has input and output interfaces, so-called *ports*. A component consist of at least one *function*.

Component Wrapper A component wrapper is the interface between a component and the *Data Handler* core component. For every input port of the respective component, it offers a respective reading function. For every output port of the respective component, it offers a respective writing function. The reading and writing functions are called by the component's functions. The component wrapper enforces the isolation between CHROMOSOME core components and (untrusted) functions with respect to data passing.

Ecosystem A collection of CHROMOSOME nodes that communicate with each other. Nodes can be added to and removed from an CHROMOSOME ecosystem via plug & play.

Function A function is a primitive algorithm that is associated with a *component*. It may read a subset of its *component*'s input *ports* and may write a subset of its *component*'s output ports. Functions are annotated with their worst-case execution time.

Function Wrapper The function wrapper is the interface between the *Execution Manager* component and a function. Whenever the *Execution Manager* runs a function, it does not directly call the function, but instead it unblocks the thread that executes the respective function wrapper. The function wrapper then ensures that all data that the respective function needs are readily available and then calls the function. When the function returns, the function wrapper signals the *Execution Manager* accordingly. The function wrapper enforces the isolation between CHROMOSOME core components and (untrusted) functions with respect to execution.

Node In terms of CHROMOSOME, a node is an instance of the CHROMOSOME runtime system that implements a certain part of a (possibly distributed) application.

Node Identifier A node identifier is a number uniquely identifying a *node* in a CHROMOSOME network. Node identifiers are maintained by the *Login Manager*.

Play Phase Time where the distributed system is running in order to achieve the intended task. Opposite of *plug phase*.

Plug Phase Time where the distributed system is being configured or reconfigured automatically, depending on specified requirements. Plug phase is entered on system startup or on request by the user or *Plug & Play Manager*. Opposite of *play phase*.

Port A port is an input or output interface of a component. Ports are associated with a specific *topic* that defines the type of data expected respectively emitted by the port. Ports may be implemented by a single memory cell, a queue, or a ring buffer, for example. A port specifies whether it is persistent, that is whether the last received value contained in a port will be kept around for further read operations or not.

Topic A topic is a type of data transmitted between *ports* of *components*.

Waypoint A waypoint is a lightweight component that applies a certain transformation to data. For example, marshaling and demarshaling are two waypoints. Likewise, sending and receiving of data according to a specific protocol are represented by two waypoints.[22] Waypoints have a configuration interface which is used to "program" their tasks during *plug phase*. During *play phase*, waypoints deterministically achieve their task according to their configuration.

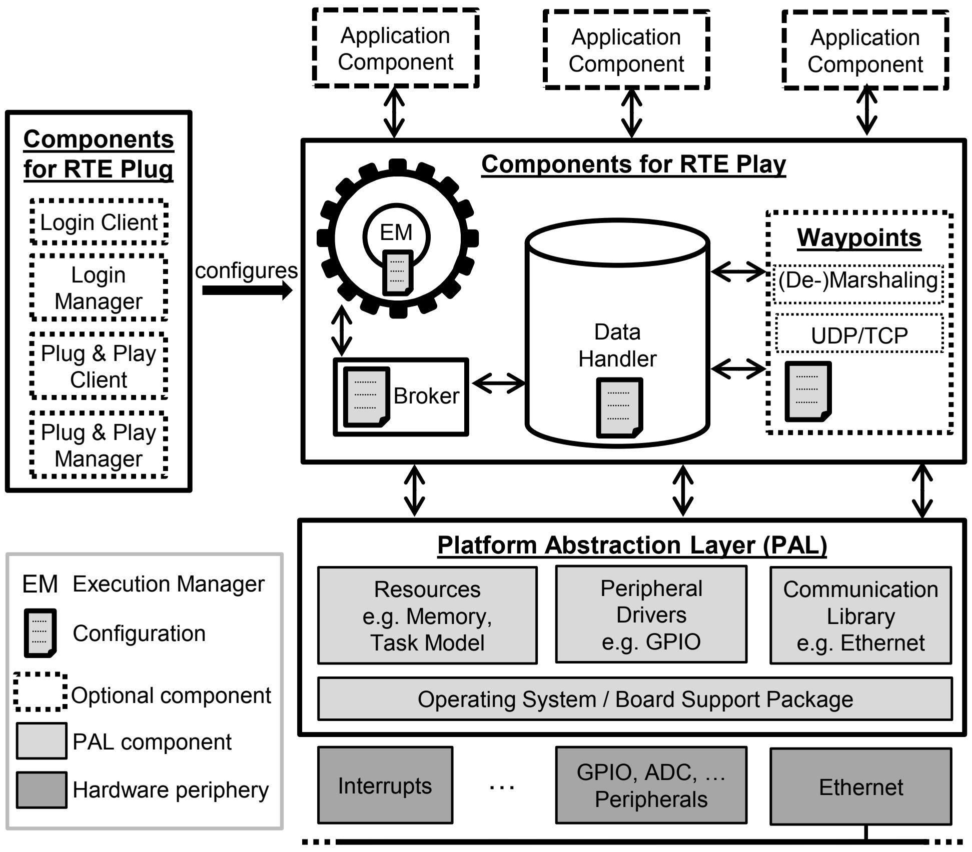


Figure 1: CHROMOSOME architecture overview.

3.3 CHROMOSOME Components

CHROMOSOME is designed in a modular fashion. Its component-oriented architecture is depicted in Figure 1. A (distributed) *application* consists of a set of (software) *components* that interact with each other by exchanging data. Each component is placed on a specific *node* in the network. On operating system based platforms like Linux or Windows, the software of a node is typically represented by a console application. On embedded systems, the software of a node corresponds to the firmware. A component may contain a number of *functions* that

[22]In previous versions of CHROMOSOME, the now obsolete *Interface Manager* was responsible for transferring data over the network.

exchange data via the *ports* of their parent component. A component's *input port* may receive data from other components and forward it to one of its functions for processing. Functions may generate data for publication on their parent component's *output ports*.

The following sections provide brief descriptions of the most important components of CHROMOSOME. See Figure 1 for a graphical illustration.

3.3.1 Mandatory Components for RTE Play

"RTE play" refers to the normal execution of a CHROMOSOME node.

Broker The Broker "knows" which function depends on which data in the input ports of their parent components. As such, it knows when a function has enough data to be executed. When data is available on an output *port* of a *component*, the Broker copies them to the dependent input *ports* by triggering a transfer function in the *Data Handler*. Hence, the Broker offers an interface where the *Execution Manager* can query whether a function is currently ready for execution.[23]

Execution Manager The Execution Manager is responsible for executing functions that are ready for execution. Depending on the model of execution (currently, only time triggered execution is supported), the Execution Manager has a set of schedules (time triggered) or priorities (event triggered), or both (hybrid).

In the time triggered case, the Execution Manager executes checks for every component whether it is ready for execution in a round-robin fashion. For this purpose, it asks the *Broker* whether the data dependencies of the respective function are satisfied. Under all functions that are ready for execution, the Execution Manager chooses one to execute. While a function is executed, it reads data from the ports of the respective component (by calling respective functions in the *Data Handler*). This may lead to the function not being executable any more in the next cycle.

Data Handler The Data Handler offers a central storage for all data exchanged via data centric communication ports of components on a node. The *component wrappers* interact with the Data Handler by reading and writing data and attributes.

3.3.2 Optional Components for RTE Plug

"RTE plug" refers to reconfigurability aspects of CHROMOSOME.

Logical Route Manager The Logical Route Manager (not shown in Figure 1) receives a list of all publications and subscriptions from the *Plug & Play Manager* when the system configuration is to be updated (*plug phase*). Based on this information, the Logical Route Manager calculates logical routes, i.e., it finds sources and sinks with matching topics and attributes. The set of logical routes is forwarded to the *Network Configuration Calculator*.

Network Configuration Calculator The Network Configuration Calculator (not shown in Figure 1) takes the logical routes calculated by the *Logical Route Manager* and enhances them with communication protocol specific transformation steps that are applied to the data in between the sender (publication) and the receiver (subscription). We call such transformations *waypoints* on the data path. Furthermore, the Network Configuration Calculator calculates where in the network the individual waypoints are to be placed. The result is a set of physical routes and deployment information that is sent back to the *Plug & Play Manager*.

[23]In previous versions of CHROMOSOME, the Broker was responsible for delivering data to components. In this version of CHROMOSOME, the actual transfer is achieved by the *Data Handler*.

Plug & Play Manager The Plug & Play Manager is responsible for handling requests for dynamic changes to the running system. It is also responsible for the startup procedure. The Plug & Play Manager is either triggered by a system reset, by the user, or by the login or logout of a node in the network. It calculates the set of changes that will be applied to the system and triggers other components accordingly, such as the *Logical Route Manager*, the *Network Configuration Calculator*, or the *Plug & Play Clients* on remote nodes. Only one Plug & Play Manager should exist in a network.

Plug & Play Client The Plug & Play Client (not shown in Figure 1) receives commands from the *Plug & Play Manager* that is present on a specific node in the network. It interprets those commands and adapts the configuration of local components accordingly, such as the *Broker*, the *Data Handler*, and the *Execution Manager*.

Login Client The Login Client (not shown in Figure 1) is responsible for connecting a node to other nodes in the network and obtaining a so-called *node identifier*. The node identifier is a unique address of the node within a network. The Node Manager periodically tries to obtain such a node identifier from the *Login Manager*, which is present on one specific node in the network. This component is optional, because in statically configured networks, no login or logout might be necessary or supported.

Login Manager The Login Manager receives login requests from the *Login Client*s of all other nodes and handles them by assigning respective *node identifiers*. Only one Login Manager should exist in a network.

Node Manager A node can have an arbitrary amount of software components that implement the application. The Node Manager (not shown in Figure 1) provides management mechanisms for managing existing components and instantiating new components.

3.3.3 Primitive Components

Primitive Components directly access the *Hardware Abstraction Layer* using function calls and offer data obtained from the hardware in a data centric way or forward data to the respective actuators. They are typically the source of physical data obtained from the environment and the sink for control data to the environment.

3.3.4 Platform/Hardware Abstraction Layer

The Platform/Hardware Abstraction Layer, often abbreviated by PAL or HAL, provides a platform independent application programming interface (API). CHROMOSOME *components* are usually implemented against that API and are hence platform independent. The HAL implements hardware-specific functionality such as drivers for microcontroller periphery like general purpose I/O (GPIO) or analog to digital conversion (ADC), but it also provides a lot of software functions similar to an operating system. See the content of the directory `<XME_ROOT>/xme/hal/include`[24] for an overview of the supported functionality. Each header file in this directory corresponds to a HAL module and contains documentation about the provided functionality.

[24]`<XME_ROOT>` specifies the root of the CHROMOSOME source package. See Chapter 4 for details.

3.3.5 Application Components

The actual (distributed) application running in the network is implemented by specific Application Components. They typically operate only on data obtained via data centric communication. In particular, no direct access to the Hardware Abstraction Layer with effects in the environment should be made, instead Application Components should use *Primitive Components*. This design concept leads to modular, reusable components.

4 Example 1: Sensor and Monitor (20 minutes)

We will start by compiling a simple application which is used to collect data from the workstation the application is running on, such as the free space on a certain partition. We will then extend the example subsequently in later chapters.

Two steps are required to build an application based on CHROMOSOME: first, the build system needs to be generated using CMake and second the application needs to be compiled using the generated build system. After the installation of the required prerequisites (see Section 2), the following steps are required:

4.1 Configuration on Linux

1. Download the source archive of CHROMOSOME from the website http://chromosome.fortiss.org/.

2. Extract the archive to a directory of your choice. Either use a GUI tool or type the following commands in the console:

   ```
   user@host: > tar -xvzf xme-0.8-rc1-src.tar.gz
   user@host: > cd xme-0.8-rc1-src
   user@host: /xme-0.8-rc1-src> ls
   AUTHORS.txt  examples  INSTALL.txt  README.txt  THIRDPARTY.txt  VERSION.txt
   doc          external  LICENSE.txt  tests       tools           xme
   ```

   The xme-0.8-rc1-src directory contains LICENSE.txt. From now on, we will call that directory <XME_ROOT>.

3. We will now generate the required build system for building an example node. For this purpose, we first create a build directory and then run CMake.

   ```
   user@host:XME_ROOT> mkdir -p examples/sensorMonitor/build/sensorNode
   user@host:XME_ROOT> cd examples/sensorMonitor/build
   ```

   From now on, we will call that directory <BUILD_ROOT>.

4. The example that we are going to build consists of two nodes. Create a subdirectory in BUILD_ROOT for the first node:

   ```
   user@host:BUILD_ROOT> mkdir sensorNode
   user@host:BUILD_ROOT> cd sensorNode
   ```

5. Generate the required build system for building the sensor node:

   ```
   user@host:BUILD_ROOT/sensorNode> cmake -G "Unix Makefiles" \
                                    ../../src/application/sensorNode
   ```

6. Compile the example:

   ```
   user@host:BUILD_ROOT/sensorNode> make
   ```

7. Run the example application:

   ```
   user@host:BUILD_ROOT/sensorNode> target/sensorNode
   ```

8. Continue in Section 4.4.

4.2 Configuration on Windows

1. Download the source archive of CHROMOSOME from the website `http://chromosome.fortiss.org/`.

2. Extract the archive to a directory of your choice.[25] After extraction you should see a directory that contains `LICENSE.txt`. From now on, we will call that directory `<XME_ROOT>`.

3. Use the *start menu* to run *CMake (cmake-gui)* (compare Figure 2).

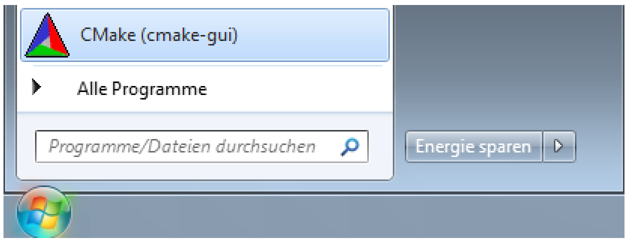
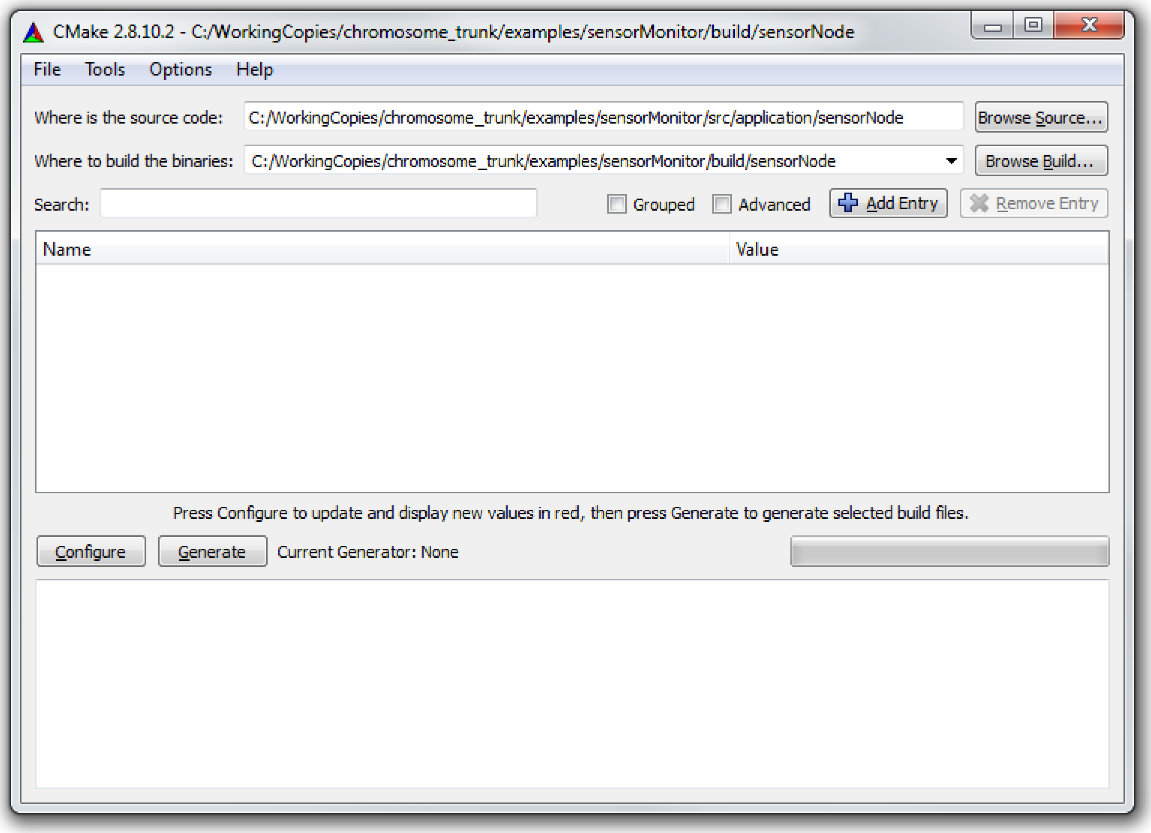


Figure 2: *CMake (cmake-gui)* icon in the start menu.

4. In the *Where is the source code* field, select the full path to the directory `<XME_ROOT>/examples/sensorMonitor/src/application/sensorNode` (compare Figure 3).


CMake 2.8.10.2 - C:/WorkingCopies/chromosome_trunk/examples/sensorMonitor/build/sensorNode

Where is the source code: C:/WorkingCopies/chromosome_trunk/examples/sensorMonitor/src/application/sensorNode

Where to build the binaries: C:/WorkingCopies/chromosome_trunk/examples/sensorMonitor/build/sensorNode

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: None

Figure 3: Source code and build directory specification.

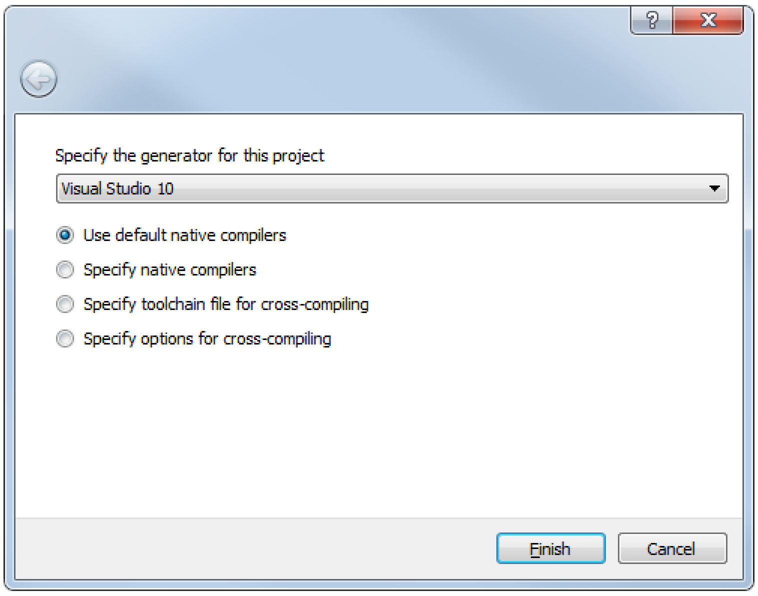
[25]On Windows systems, please ensure that the path name is not extraordinary long, as this could cause issues with the build system (depending on the file system used).

5. In the *Where to build the binaries* field, select the full path to the directory `<XME_ROOT>/examples/sensorMonitor/build/sensorNode` (note the additional `sensorNode` folder, this folder does not yet exist).

6. Click the *Configure* button.

7. If asked whether the build directory should be automatically created, say *Yes*.

8. Choose the *Visual Studio* toolchain that corresponds to your Visual Studio version (do *not* use the 64 bit version, even on a 64 bit system) and click *Finish* (compare Figure 4).



Specify the generator for this project

Visual Studio 10

● Use default native compilers

○ Specify native compilers

○ Specify toolchain file for cross-compiling

○ Specify options for cross-compiling

Figure 4: Toolchain selection in CMake.

9. After CMake has finished its configuration, various configuration variables marked in red should appear in the list (compare Figure 5).

10. Click the *Generate* button. This will generate the Visual Studio solution file, called `sensorNode.sln` which you can open in Visual Studio. The file will be placed in the following directory: `<XME_ROOT>/examples/sensorMonitor/build/sensorNode` (compare Figure 6).

To compile the exemplified application, the following steps are required:

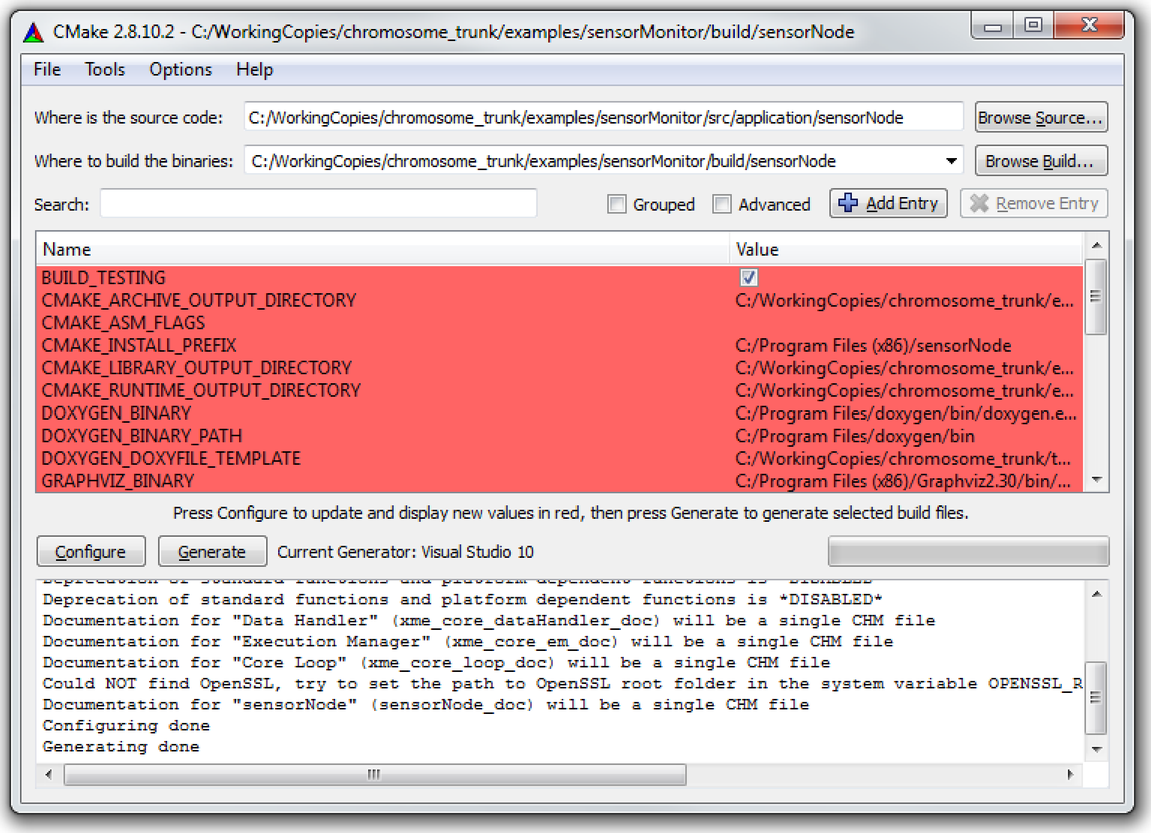
1. Fire up Visual Studio and select *File* → *Open* → *Project/Solution...*

2. Navigate to the `<XME_ROOT>/examples/sensorMonitor/build/sensorNode` directory and select the solution file `sensorNode.sln`.

3. After loading the solution, you will see a project tree in the left-hand pane. Right-click on the `sensorNode` project and choose *Set as StartUp Project* (compare Figure 7).

4. In the tool bar, select the solution configuration you want to build (usually `Debug` or `Release`). The `Debug` build includes debugging information and should be used for development. If in doubt, choose `Debug`.

5. Hit *F7* to compile the whole solution.



Where is the source code: C:/WorkingCopies/chromosome_trunk/examples/sensorMonitor/src/application/sensorNode

Where to build the binaries: C:/WorkingCopies/chromosome_trunk/examples/sensorMonitor/build/sensorNode

Name	Value
BUILD_TESTING	✓
CMAKE_ARCHIVE_OUTPUT_DIRECTORY	C:/WorkingCopies/chromosome_trunk/e...
CMAKE_ASM_FLAGS	
CMAKE_INSTALL_PREFIX	C:/Program Files (x86)/sensorNode
CMAKE_LIBRARY_OUTPUT_DIRECTORY	C:/WorkingCopies/chromosome_trunk/e...
CMAKE_RUNTIME_OUTPUT_DIRECTORY	C:/WorkingCopies/chromosome_trunk/e...
DOXYGEN_BINARY	C:/Program Files/doxygen/bin/doxygen.e...
DOXYGEN_BINARY_PATH	C:/Program Files/doxygen/bin
DOXYGEN_DOXYFILE_TEMPLATE	C:/WorkingCopies/chromosome_trunk/t...
GRAPHVIZ_BINARY	C:/Program Files (x86)/Graphviz2.30/bin/...

Press Configure to update and display new values in red, then press Generate to generate selected build files.

Current Generator: Visual Studio 10

```
Deprecation of standard functions and platform dependent functions is *DISABLED*
Documentation for "Data Handler" (xme_core_dataHandler_doc) will be a single CHM file
Documentation for "Execution Manager" (xme_core_em_doc) will be a single CHM file
Documentation for "Core Loop" (xme_core_loop_doc) will be a single CHM file
Could NOT find OpenSSL, try to set the path to OpenSSL root folder in the system variable OPENSSL_R
Documentation for "sensorNode" (sensorNode_doc) will be a single CHM file
Configuring done
Generating done
```

Figure 5: Configuration and build system generation in CMake.

6. Debug/run the project as usual (e.g., hit *F5* to debug or *Ctrl+F5* to run without debugging). If you get prompted whether to rebuild the out-of-date `ZERO_CHECK` project (compare Figure 8), you may select the *Do not show this dialog box again* check box and choose *Yes*.

7. Continue in Section 4.4.

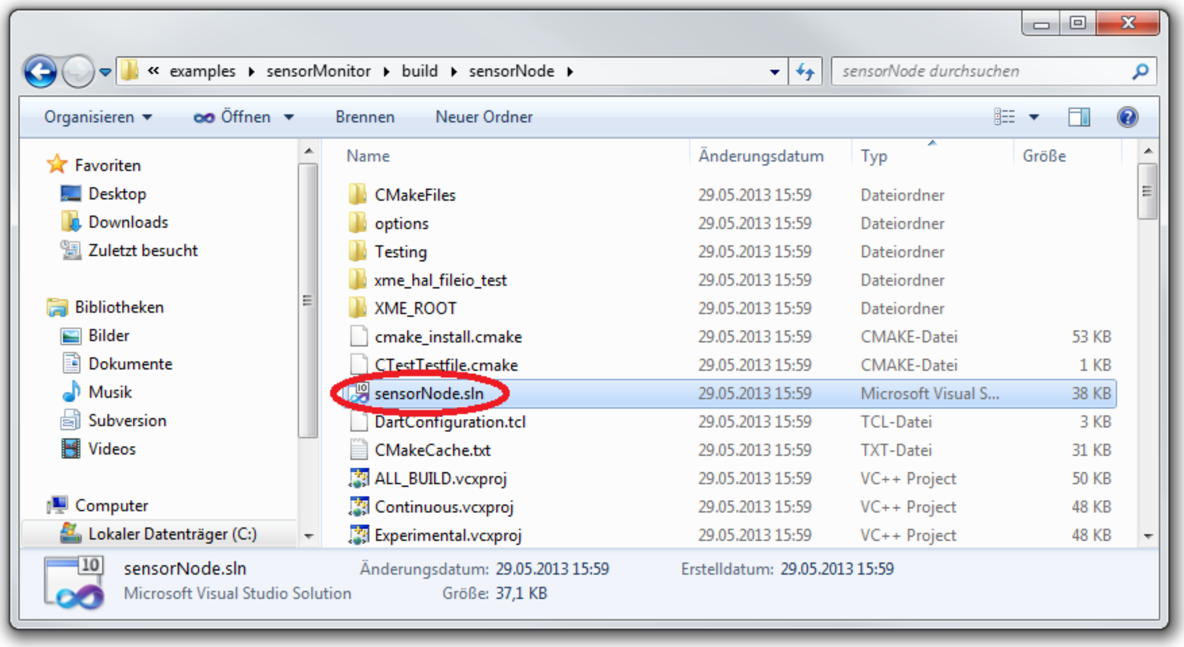
4.3 Notice on Project-global `CMakeLists.txt` Files

In CHROMOSOME, a separate root `CMakeLists.txt` file is provided for every node. Thsi is because it could be that each node is to be built for a different target system using a different compiler toolchain. However, since version 0.8, CHROMOSOME also support a so-called project-global `CMakeLists.txt` file per project. That file is usually located in the root directory of the respective example, say `<XME_ROOT>/examples/sensorMonitor/CMakeLists.txt`. That file can roughly be undestood as to include all node-specific `CMakeLists.txt` files.[26] Use the project-global `CMakeLists.txt` file if all nodes of a project are to be built using the same toolchain for the same target platform. In this case, it is recommended to name the build directory in the same way as the example itself is named, for example:
`<XME_ROOT>/examples/sensorMonitor/build/sensorMonitor`

4.4 Running the Example Application

What you have just compiled is an application that queries simple data from the host it is running on and makes them accessible to other nodes in the network. In this case, the collected

[26] With a few exceptions, for example when multiple deployment models exist for a project, see Section 5.



Name	Änderungsdatum	Typ	Größe
CMakeFiles	29.05.2013 15:59	Dateiordner	
options	29.05.2013 15:59	Dateiordner	
Testing	29.05.2013 15:59	Dateiordner	
xme_hal_fileio_test	29.05.2013 15:59	Dateiordner	
XME_ROOT	29.05.2013 15:59	Dateiordner	
cmake_install.cmake	29.05.2013 15:59	CMAKE-Datei	53 KB
CTestTestfile.cmake	29.05.2013 15:59	CMAKE-Datei	1 KB
sensorNode.sln	29.05.2013 15:59	Microsoft Visual S...	38 KB
DartConfiguration.tcl	29.05.2013 15:59	TCL-Datei	3 KB
CMakeCache.txt	29.05.2013 15:59	TXT-Datei	31 KB
ALL_BUILD.vcxproj	29.05.2013 15:59	VC++ Project	50 KB
Continuous.vcxproj	29.05.2013 15:59	VC++ Project	48 KB
Experimental.vcxproj	29.05.2013 15:59	VC++ Project	48 KB

sensorNode.sln Änderungsdatum: 29.05.2013 15:59 Erstelldatum: 29.05.2013 15:59
Microsoft Visual Studio Solution Größe: 37,1 KB

Figure 6: Build directory after build system generation, highlighted in red the Visual Studio solution file.

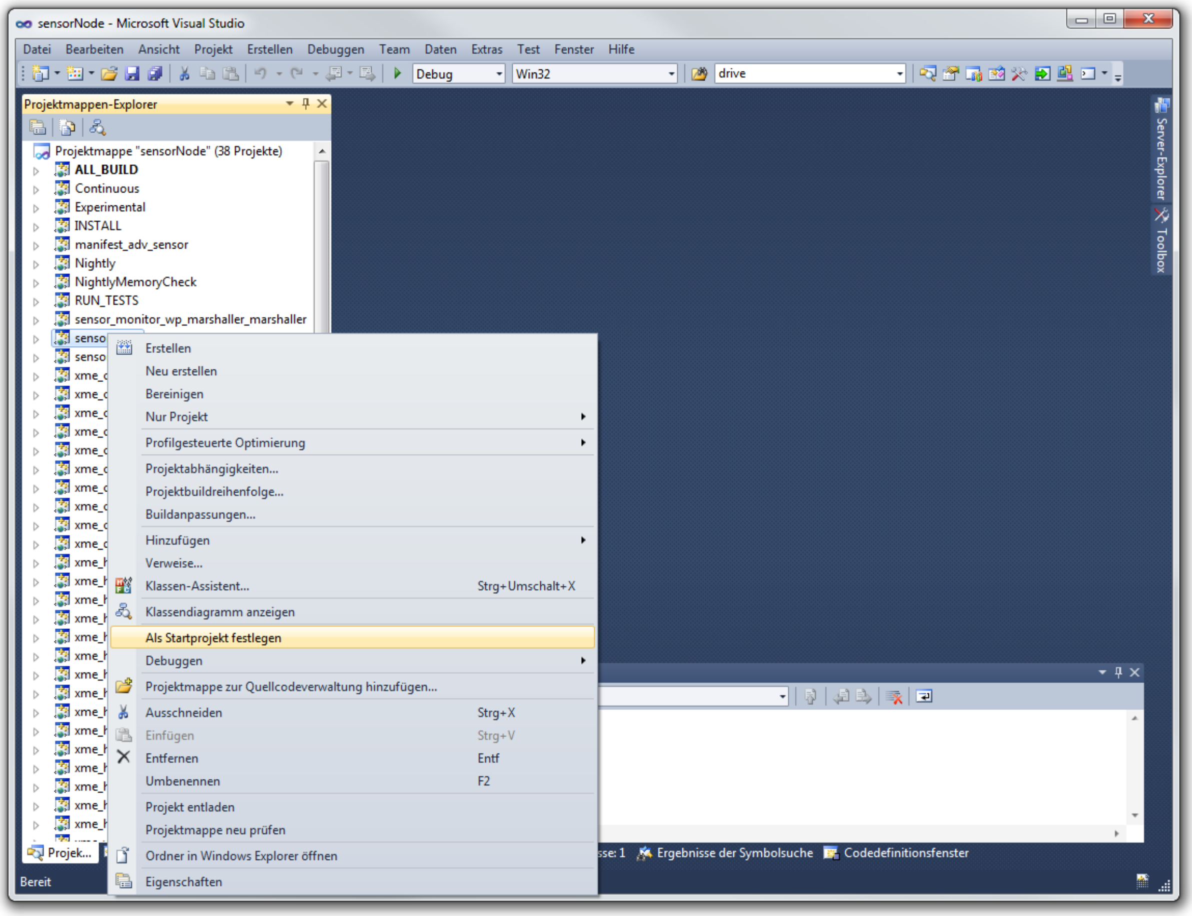
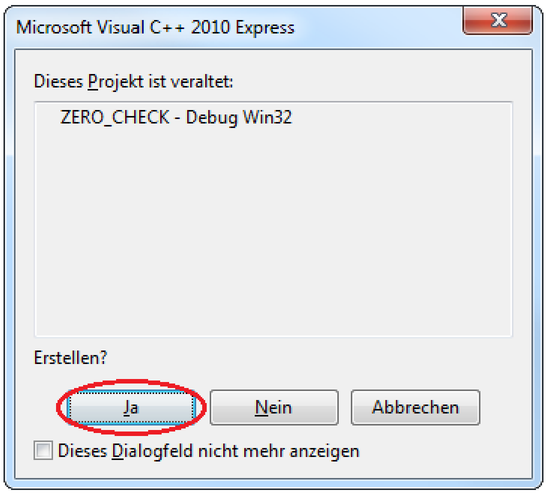


Figure 7: Setting the `sensorNode` project as *StartUp Project*.



Microsoft Visual C++ 2010 Express

Dieses Projekt ist veraltet:

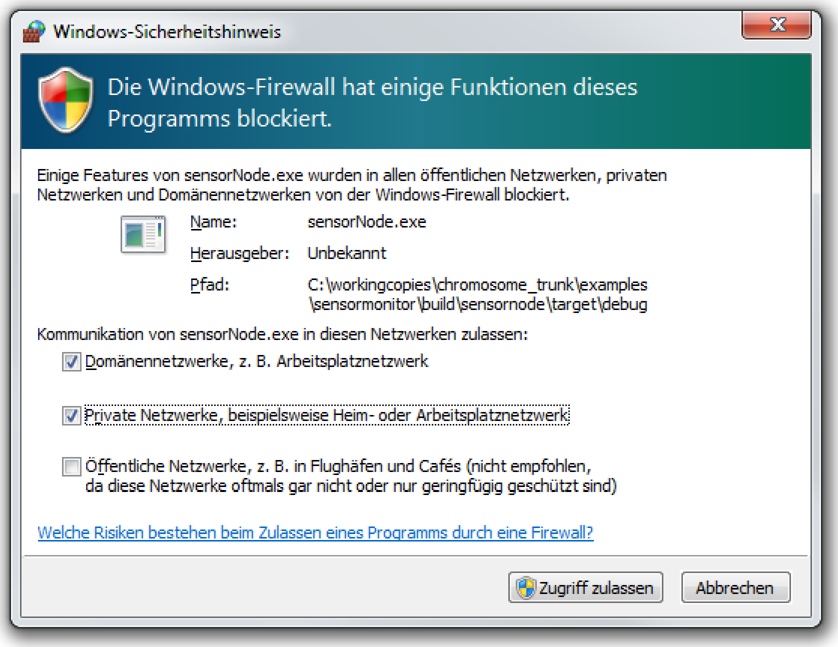
ZERO_CHECK - Debug Win32

Erstellen?

Figure 8: `ZERO_CHECK` project out of date prompt.

data is the amount of free space on a certain partition.

When the application starts, you will probably receive a query from the *Windows Firewall*. CHROMOSOME is designed to talk to other nodes in the network and hence needs a firewall exception for full functionality. If you do not want the CHROMOSOME to talk to other nodes, then simply deny access.[27] In this case, communication will be limited to CHROMOSOME applications on the local computer. If you want CHROMOSOME applications to communicate with each other in the local network, allow access to the *Home or Company Network* as shown in Figure 9.



Windows-Sicherheitshinweis

Die Windows-Firewall hat einige Funktionen dieses Programms blockiert.

Einige Features von sensorNode.exe wurden in allen öffentlichen Netzwerken, privaten Netzwerken und Domänennetzwerken von der Windows-Firewall blockiert.

Name:	sensorNode.exe
Herausgeber:	Unbekannt
Pfad:	C:\workingcopies\chromosome_trunk\examples \sensormonitor\build\sensornode\target\debug

Kommunikation von sensorNode.exe in diesen Netzwerken zulassen:

☑ Domänennetzwerke, z. B. Arbeitsplatznetzwerk

☑ Private Netzwerke, beispielsweise Heim- oder Arbeitsplatznetzwerk

☐ Öffentliche Netzwerke, z. B. in Flughäfen und Cafés (nicht empfohlen, da diese Netzwerke oftmals gar nicht oder nur geringfügig geschützt sind)

Welche Risiken bestehen beim Zulassen eines Programms durch eine Firewall?

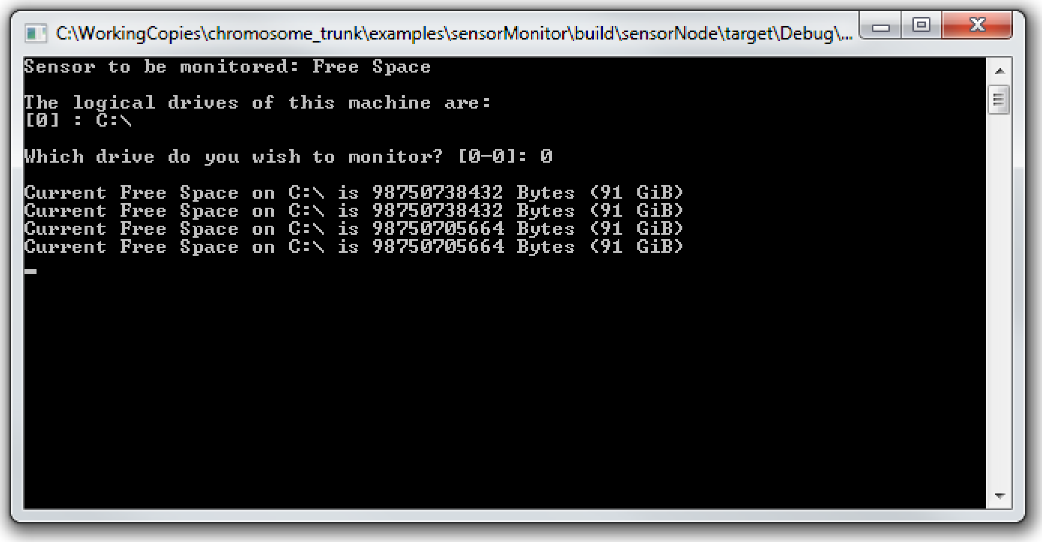
Figure 9: Firewall settings for the `sensorNode` application.

Once the application starts, it presents you with a list of partitions (Linux) or drives (Windows) to choose from. Select a valid item by entering the respective number. The amount of free

[27]If you want to allow access to the network at a later point in time, you can change the firewall settings in the *System Control Panel*.

space is periodically printed on the console window (compare Figure 10) and also transmitted to another CHROMOSOME node (which is not yet present, see Section 4.5). Notice that you can run multiple versions of the sensor node for different partitions/drives on a single host.[28]

The entry point into the executable is in the main source file `<XME_ROOT>/examples/sensorMonitor/src/application/sensorNode/sensorNode.c`, which also defines the CHROMOSOME components used in this application. Notice that this file (and most other files in the example) have been generated from a model. We will see how this is done in Section 5.


C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor\build\sensorNode\target\Debug\...

```
Sensor to be monitored: Free Space

The logical drives of this machine are:
[0] : C:\

Which drive do you wish to monitor? [0-0]: 0

Current Free Space on C:\ is 98750738432 Bytes (91 GiB)
Current Free Space on C:\ is 98750738432 Bytes (91 GiB)
Current Free Space on C:\ is 98750705664 Bytes (91 GiB)
Current Free Space on C:\ is 98750705664 Bytes (91 GiB)
_
```

Figure 10: Sensor node printing free space on drive C: (here on Windows).

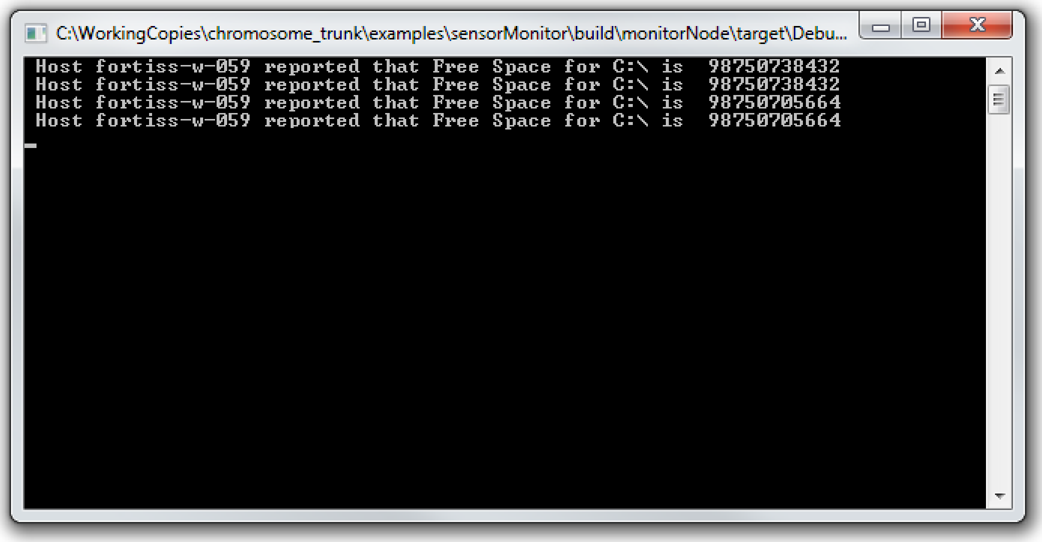
4.5 Compiling and Running the Monitor

Now it's time to compile and run the monitor node that will receive the measurements from the sensor. Follow the same steps as in Section 4.1 or Section 4.2 (depending on your operating system), but exchange all references to `sensorNode` with `monitorNode`. On Windows, it is recommended to open a separate Visual Studio instance in order to be able to debug both applications side by side.

When the application starts, you will probably receive another query from the *Windows Firewall*. Proceed in the same way than for the `sensorNode`.

When launching the `monitorNode` application, you should see the collected data from all sensor nodes on the *current host* being displayed (compare Figure 11). We will shortly come back to remote data transmission in Section 5.5.

[28]On Windows, when launching the application from within Visual Studio, only one instance can be run at a time. In this case, manually navigate to the build directory `<BUILD_ROOT>/sensorNode/target` and run the executable from there.


C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor\build\monitorNode\target\Debu...

```
Host fortiss-w-059 reported that Free Space for C:\ is  98750738432
Host fortiss-w-059 reported that Free Space for C:\ is  98750738432
Host fortiss-w-059 reported that Free Space for C:\ is  98750705664
Host fortiss-w-059 reported that Free Space for C:\ is  98750705664
_
```

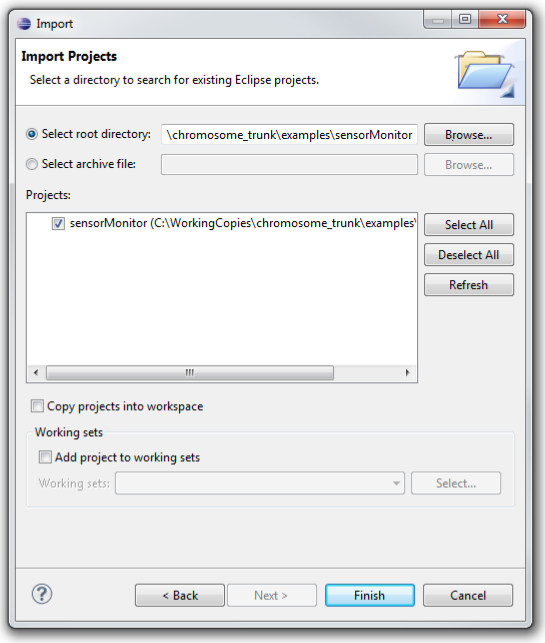
Figure 11: Monitor node printing free space (here on Windows).

5 Example 2: Code Generation with the Modeling Tool (20 minutes)

Only a fraction of the C code from the last example has actually been written manually. CHROMOSOME is released with a model-driven development tool which allows to model an application and subsequently generate code from this model. The CHROMOSOME Modeling Tool (XMT) has been implemented in form of an Eclipse[29] plugin. Refer to appendix F for a description of how to set up Eclipse and the modeling tool. From now on we assume that you have successfully installed the tool and have it already running.

The XMT comes with a dedicated perspective, which will open a set of related views and where certain XMT-related commands are activated. To switch to this perspective go to *Window → Perspective → Other...* and choose *XMT*.

The sensor/monitor example project from Section 4 was generated from an XMT project. We will now import the respective example and its models into the Eclipse workspace. In the menu, choose *File → Import...*, open the *General* category and double-click on *Existing Projects into Workspace*. Make sure *Select root directory* is checked and press the *Browse...* button om the top right. Select the example directory `<XME_ROOT>/examples/sensorMonitor` and press *OK*. The project should appear in the list of *projects* and it should be checked (if not, add the check mark). Make sure that *Copy projects into workspace* is not checked (compare Figure 12). Press *Finish* to import the project, which will show up in the *Project Explorer* pane.



Import Projects

Select a directory to search for existing Eclipse projects.

(●) Select root directory: \chromosome_trunk\examples\sensorMonitor

( ) Select archive file:

Projects:

[✓] sensorMonitor (C:\WorkingCopies\chromosome_trunk\examples\

[ ] Copy projects into workspace

Working sets

[ ] Add project to working sets

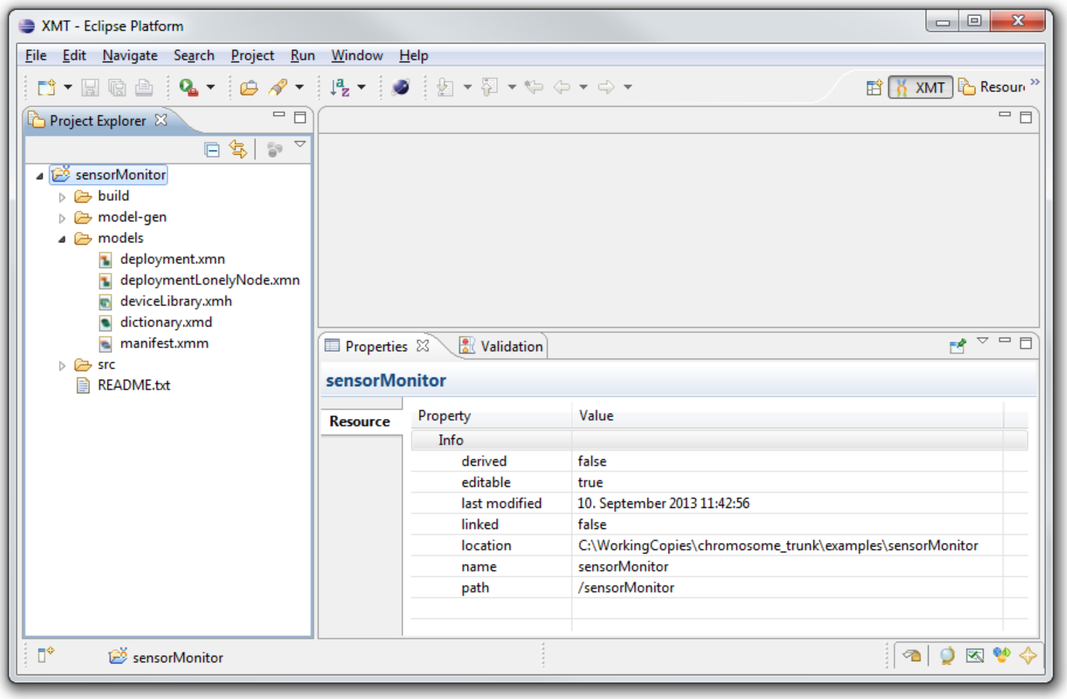
Working sets:

Figure 12: Importing the *sensorMonitor* project into XMT.

You already know the `src/` subdirectory of this example from the previous chapter. Now we will have a closer look at the `models/` directory, which contains the XMT models that describe this application. You can see a screenshot of the imported project with expanded models directory in Figure 13. The directory contains five models. In the *models* folder we can distinguish four different file extensions:

- *xmd* This extension stands for *XMT* topic dictionary, and determines the set of topics and associated attributes to be used for the deployment of components.

[29] `http://www.eclipse.org/`



```
sensorMonitor
  build
  model-gen
  models
    deployment.xmn
    deploymentLonelyNode.xmn
    deviceLibrary.xmh
    dictionary.xmd
    manifest.xmm
  src
  README.txt
```

sensorMonitor

Property	Value
Info	
derived	false
editable	true
last modified	10. September 2013 11:42:56
linked	false
location	C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor
name	sensorMonitor
path	/sensorMonitor

Figure 13: XMT perspective of CHROMOSOME Modeling Tool with imported example project.

- *xmm* This extension stands for *XMT* manifest, and defines components and their interfaces.
- *xmh* This extension stands for *XMT* device types, and is related with hardware-associated properties.
- *xmn* This extension stands for *XMT* deployment model, and defines the mapping of components on devices and the network structure.

In the following subsections we will have a look at these four model types, providing a short description of their purpose and demonstrating how to generate code from them. As a preparation, we create a backup of the `src/` directory in the project. Name it something like `src_backup/`. Then delete the original. We will recreate the content step by step. Most of the files will be generated completely by the tool.

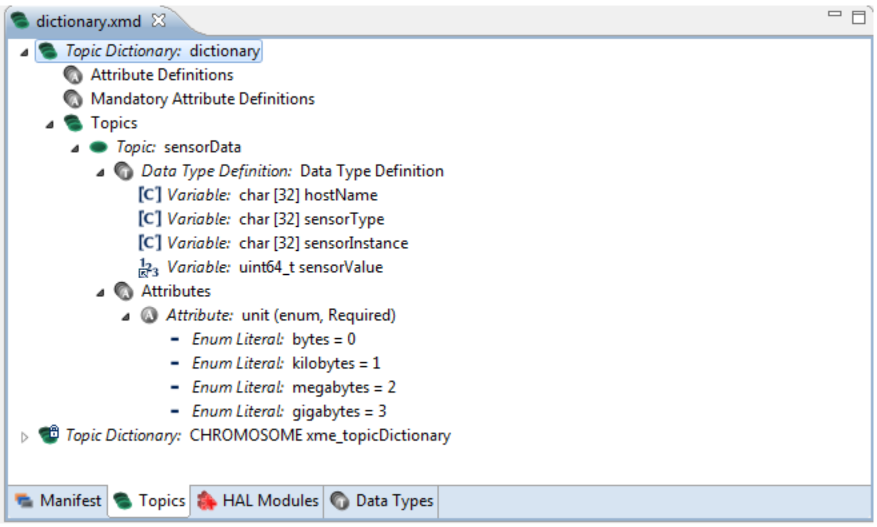
5.1 Topic Dictionary

We begin with the *topic dictionary* model. A *topic dictionary* is the collection of all topics of an application respectively application domain. It also holds the *attribute* definitions. Attributes are attached to topics and specify additional information about the contained data.

We will check how attributes work in the Plug and Play example (see Section 6).

To open the model, double-click on the file `models/dictionary.xmd` in the project explorer. An editor window will open that displays the model content in a tree structure – see Figure 14. Additionally, the CHROMOSOME *xme_topicDictionary* shows internal topics used to CHROMOSOME core components like the *Plug and Play Manager* and *Plug and Play Client*.

You can expand items to see their sub-items. For example, the item *Topic Dictionary: dictionary* contains an item *Topics* under which you can find the topic definitions. Remember that a topic defines a type of data that can be sent between components in a CHROMOSOME network (compare Section 3.2). Topics can consist of primitive C data types or complex data



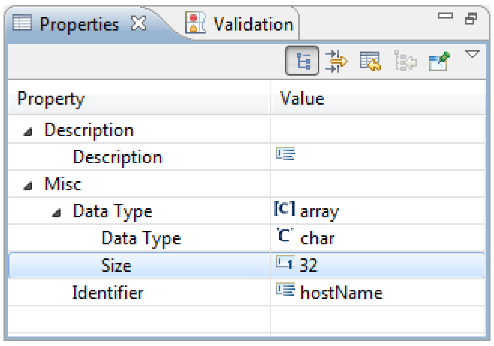
```
dictionary.xmd
◢ Topic Dictionary: dictionary
    Attribute Definitions
    Mandatory Attribute Definitions
  ◢ Topics
    ◢ Topic: sensorData
      ◢ Data Type Definition: Data Type Definition
          [C] Variable: char [32] hostName
          [C] Variable: char [32] sensorType
          [C] Variable: char [32] sensorInstance
              Variable: uint64_t sensorValue
      ◢ Attributes
        ◢ Attribute: unit (enum, Required)
            - Enum Literal: bytes = 0
            - Enum Literal: kilobytes = 1
            - Enum Literal: megabytes = 2
            - Enum Literal: gigabytes = 3
▷ Topic Dictionary: CHROMOSOME xme_topicDictionary

Manifest | Topics | HAL Modules | Data Types
```

Figure 14: Topic dictionary model.

structures. Additionally, we can define for each topic the set of attributes that apply to that topic.

A double-click on an item brings the *Properties* view to the foreground. Here you can see the properties of the selected model item. Properties may be structured in sub-properties. When this is he case you can again expand the entry by clicking on the small triangle on the left side. Figure 15 shows an example for the *hostName* variable of the *sensorData* topic.



Properties | Validation

Property	Value
◢ Description	
Description	
◢ Misc	
◢ Data Type	[C] array
Data Type	C char
Size	32
Identifier	hostName

Figure 15: Sub-properties of a data type in the topic dictionary model.

Now let us have a closer look at the single topic defined in the example, which is called *SensorData*. In its properties, you can see a description and its name. You can also see the ID (short for *identifier*) that is used at runtime to identify this topic and its C identifier that will be used in the generated code.[30] Now expand every entry under the topic. You will see that it has data type definition with several variables. In the generated code, this will be mapped to a C struct definition.

The SensorData topic also has one attribute called "*unit*" which is an enumeration. It describes the measurement unit of the contained sensor value. As in our example we measure the free hard disk size, possible values for this attribute range from *bytes* to *gigabytes*.

For a topic dictionary two code generation commands exist. The first option is to generate the topic headers. To do this, right-click on the *Topic Dictionary* root element and select *Generate Topic Dictionary Headers...* as shown in Figure 16. A dialog will pop up that shows

[30]The identifier is automatically derived from the name of the topic and the name of the dictionary and therefore located in the property category *Read-Only*.

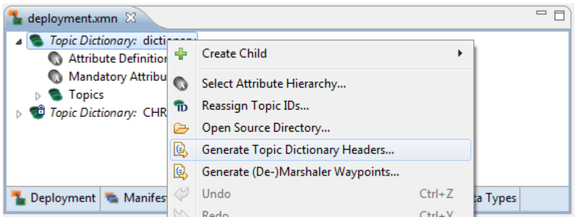
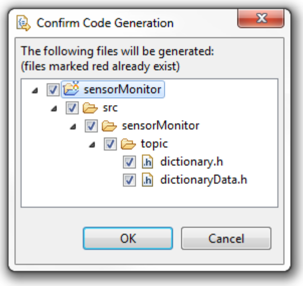


Figure 16: Start topic header generation.


Confirm Code Generation

The following files will be generated:
(files marked red already exist)

Figure 17: Code generation dialog for topic headers.

the files that will be generated – see Figure 17. You can exclude individual files by unchecking them. Here you can see that this command will generate two files: `dictionary.h` and `dictionaryData.h`. These contain the headers topic identifier and data type definitions that you already know from before. Select *Overwrite* and press *OK* to generate the files. After a short moment the Eclipse status line should say *Code generation complete*. You can navigate to the generated files and have a look at them if you are interested. If the files are missing in the Eclipse *Project Explorer* pane, try to refresh the `src/` directory (i.e., right-click and select *Refresh* from the popup menu or select the directory item and press *F5*).

The second generation command *Generate (De-)marshaler waypoints...* creates marshaling and demarshaling functions for the topics of the application. Different CHROMOSOME nodes might reside on different target platforms that interpret data differently. In order to allow these platforms to communicate with each other, the sent data must be converted into a platform independent format.[31] The serialization of sent data to this format (i.e., from platform dependent representation to network representation) is performed by the marshaler; the deserialization respectively by the demarshaler. The serialization algorithm depends on the sent data – and so in our case on the topic definition. To solve this issue, CHROMOSOME uses a code generation approach and generates a (de-)marshaler that is specific to a dictionary. You can start the generation by right-clicking on the *Topic Dictionary* item and select *Generate (De-)Marshaler Waypoints...*. From the generated files the most interesting ones are the `marshaler.c/h` and `demarshaler.c/h` files, which contain the actual (de-)serialization code.

5.2 Manifest

The next model is the so called *manifest*. A manifest can be considered as a set of component blueprints. For each component the published and subscribed topics and its functions are defined. You can create instances of these components later in the *deployment* model.

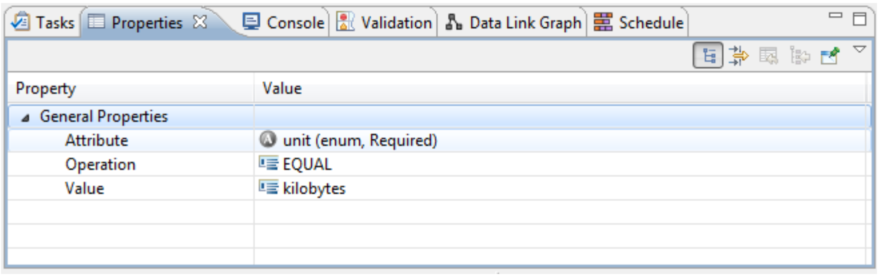
In order to open the manifest model for the *sensorMonitor* example, double-click on the file `models/manifest.xmm` in the *Project Explorer*. When you expand the model items, you can see that our manifest contains the several *Sensor* and *Monitor* component types – see Figure 19. All sensors publish the *sensorData* topic. The only difference between the different sensor types are their attribute value definitions. An attribute value definition on a publication sets an attribute of the published topic to a specific value. This means that all data items produced at runtime will have the corresponding attribute set to the the defined value.

- The *sensorB* sets the value of attribute *unit* to *bytes*.
- The *sensorKB* sets the same attribute to *kilobytes*.
- The *sensorMB* sets it to *megabytes*.

[31] For more information, see also http://en.wikipedia.org/wiki/Endianness.

Likewise, monitors subscribe the *sensorData* topic and they differ by their attribute filters. For example double-click on the filter of *monitorKB*. In the properties view see Figure 18 – you can see the filter specification. An attribute filter on a subscription specifies that the component is only interested in values where the attribute value satisfies certain criteria (e.g. is equal to a certain value).

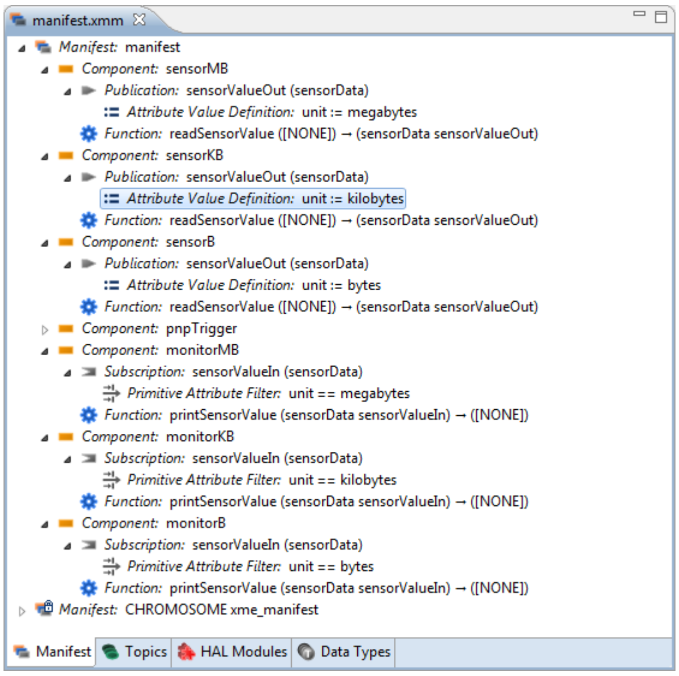
- The *monitorB* component only accepts *sensorData* topic data if the attribute *unit* has its value set to *bytes*.
- The *monitorKB* does the same except that the value needs to be *kilobytes*.
- The *monitorMB* the same again for *megabytes*.



Property	Value
General Properties	
Attribute	unit (enum, Required)
Operation	EQUAL
Value	kilobytes

Figure 18: Attribute filter properties window.

Additionally, there is a manifest called CHROMOSOME *xme_manifest* which contains *core* components. The code for these components is located in the `<XME_ROOT>/xme/core` directory. We will not need the core components in this section.



```
Manifest: manifest
  Component: sensorMB
    Publication: sensorValueOut (sensorData)
      Attribute Value Definition: unit := megabytes
    Function: readSensorValue ([NONE]) → (sensorData sensorValueOut)
  Component: sensorKB
    Publication: sensorValueOut (sensorData)
      Attribute Value Definition: unit := kilobytes
    Function: readSensorValue ([NONE]) → (sensorData sensorValueOut)
  Component: sensorB
    Publication: sensorValueOut (sensorData)
      Attribute Value Definition: unit := bytes
    Function: readSensorValue ([NONE]) → (sensorData sensorValueOut)
  Component: pnpTrigger
  Component: monitorMB
    Subscription: sensorValueIn (sensorData)
      Primitive Attribute Filter: unit == megabytes
    Function: printSensorValue (sensorData sensorValueIn) → ([NONE])
  Component: monitorKB
    Subscription: sensorValueIn (sensorData)
      Primitive Attribute Filter: unit == kilobytes
    Function: printSensorValue (sensorData sensorValueIn) → ([NONE])
  Component: monitorB
    Subscription: sensorValueIn (sensorData)
      Primitive Attribute Filter: unit == bytes
    Function: printSensorValue (sensorData sensorValueIn) → ([NONE])
Manifest: CHROMOSOME xme_manifest
```

Figure 19: Manifest model.

From the manifest model you can initiate the generation of the so called component and function wrappers and a stub for the function implementation. Right-click on the *Manifest: sensorMonitor* item and select *Generate Component Wrappers...*. In the emerging dialog you can see a list of the generated files.

Wrappers constitute the necessary glue code between the function's implementation and the middleware (see also Section 3.2). These are generated completely from the model and do not need to be adapted. The files `printSensorValueFunction.c` and `readSensorValueFunction.c` are where the actual functionality of the components are implemented. Press *OK* to generate all files and open both source files from the *sensorB*. The generated implementation simply does nothing in its *init()*, *step()* and *fini()* functions. Consider the following excerpt:

```
void
sensorMonitor_adv_sensorB_readSensorValueFunction_step
(
    void* param
)
{
    xme_status_t status[1];

    sensorMonitor_topic_sensorData_t* portSensorValueOutDataPtr =
        &portSensorValueOutData;

    {
        // PROTECTED REGION ID(↙
            ↳ SENSORMONITOR_ADV_SENSORB_READSENSORVALUEFUNCTION_STEP_C) ↙
            ↳ ENABLED START

    // TODO: Auto-generated stub

    XME_UNUSED_PARAMETER(param);
        XME_UNUSED_PARAMETER(status);

    // PROTECTED REGION END
    }

    status[0] = ↙
        ↳ sensorMonitor_adv_sensorB_sensorBComponentWrapper_writePortSensorValueOut↙
        ↳ (portSensorValueOutDataPtr);

    {
        // PROTECTED REGION ID(↙
            ↳ SENSORMONITOR_ADV_SENSORB_READSENSORVALUEFUNCTION_STEP_2_C) ↙
            ↳ ENABLED START

        // TODO: Auto-generated stub
        //       Check return values of writePort calls here

        // PROTECTED REGION END
    }
}
```

The `// TODO: Auto-generated stub` comments mark where you can add your own implementation of the respective function. In this example, these files are the only thing that you need to manually adjust to recreate the example from the previous chapter. Just copy these files over from your backup and replace the generated versions. The code listing above shows the generated `step()` function for the sensor component.

Notice the comments which are marked with `// PROTECTED REGION`. These regions enclose a section of code – marked yellow in the code listing – that will not be modified by the tool when regenerating the code. You can verify this by starting the code generation again. Do not modify or remove the protected region comments. This time before pressing *OK*, have a look at the *Select Merge Strategy* section. Figure 20 shows the bottom of the code generation dialog with the protected regions strategy highlighted. This section tells the tool what to do when a generated file already exists.[32]

[32]Existing files are marked red in the dialog.

1. The *Compare Editor* option will open a compare editor for every differing file. There you can manually select which parts to use from the generated code and which part to keep.
2. The second option *Protected Regions* will preserve any code in the protected regions. Select this option and press *OK*.
3. The third option *Overwrite* will simply overwrite existing files with generated content – use with caution.

Choose Protected Regions and verify that the content in the protected regions is kept intact.

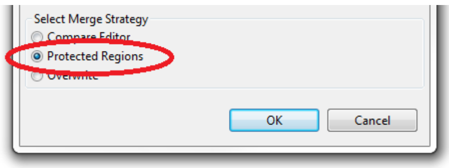
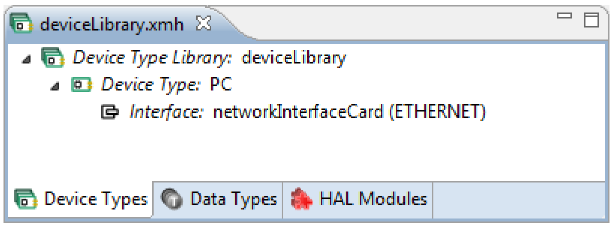


Figure 20: Code generation dialog with protected region merge strategy selected.

5.3 Device Types

The *device types* model is a very simple description of hardware devices on which your CHROMOSOME executables run. A device type has a *Target Identifier* which specifies the platform (e.g., *posix*, *windows*[33]). A device type also has several network interfaces. Currently only *ethernet* is supported. You will be able to create instances of these devices in the deployment model.

To open the model, double-click on the file `models/deviceTypes.xmn` in the project explorer – see Figure 21. It contains only a simple device with a single Ethernet interface. The *target identifier* is deliberately left empty. In this case, the CHROMOSOME build system will automatically detect the build environment and assume it to be the execution environment.


deviceLibrary.xmh

Device Type Library: deviceLibrary
Device Type: PC
Interface: networkInterfaceCard (ETHERNET)

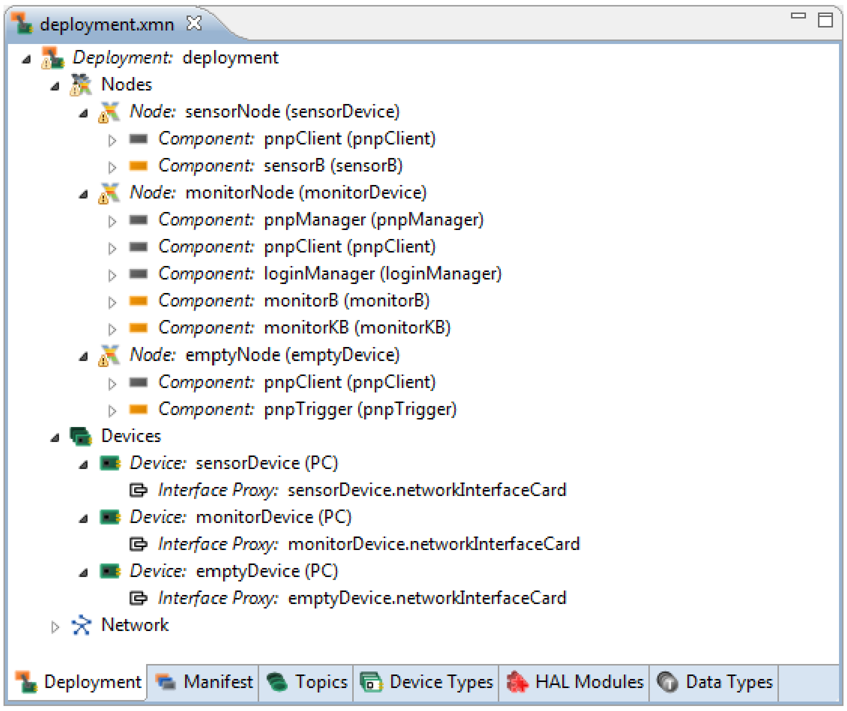
Figure 21: Device types model.

5.4 Deployment

The *deployment* model ties everything together. Here you specify devices and nodes and specify which components are deployed on which node. In the future you will also be able to specify the network structure here, which will allow the tool to calculate routes for your network and configure your nodes accordingly.[34]

To open the model, double-click on the file `models/deployment.xmn` in the project explorer – see Figure 22. This example contains three nodes. The *monitorNode* contains the monitor component and the *monitorNode* contains the sensor component. There are also other components and an additional node that you will get to know in the next chapters.

[33]Currently these are the only officially supported platforms.
[34]Actually you can already define the network structure, but this information is currently not used.



```
deployment.xmn
Deployment: deployment
  Nodes
    Node: sensorNode (sensorDevice)
      Component: pnpClient (pnpClient)
      Component: sensorB (sensorB)
    Node: monitorNode (monitorDevice)
      Component: pnpManager (pnpManager)
      Component: pnpClient (pnpClient)
      Component: loginManager (loginManager)
      Component: monitorB (monitorB)
      Component: monitorKB (monitorKB)
    Node: emptyNode (emptyDevice)
      Component: pnpClient (pnpClient)
      Component: pnpTrigger (pnpTrigger)
  Devices
    Device: sensorDevice (PC)
      Interface Proxy: sensorDevice.networkInterfaceCard
    Device: monitorDevice (PC)
      Interface Proxy: monitorDevice.networkInterfaceCard
    Device: emptyDevice (PC)
      Interface Proxy: emptyDevice.networkInterfaceCard
  Network
```

Figure 22: Fully expanded deployment model.

A node here refers to a CHROMOSOME executable. Components from the manifests can be instantiated on each node. You can also instantiate the same component multiple times on the same node (though the example does not include such a case).

A device represents a hardware device on which one or several nodes can be run. Usually you run only one CHROMOSOME executable per device, but there might be exceptions. A device is an instantiation of a type defined in a device types model. This allows to reuse a device definition. In addition to the information from the type, you have to specify a network address for each interface.

In the deployment model you can generate the main source file for a node, which will start a CHROMOSOME instance. This main source file will also set up the CHROMOSOME configuration as determined by the tool. Additionally the CMake scripts are generated that allow you to build your application. To start the code generation, right-click on the *Deployment: deployment* item and select *Generate Application Code...* (compare Figure 23). There will be

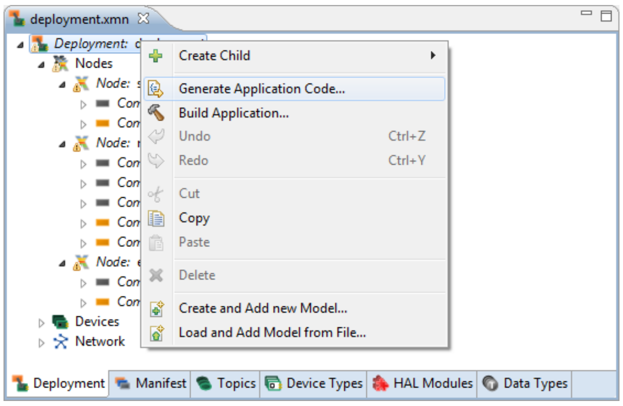


Figure 23: Generation of application code from a deployment model.

three files per node. The `sensorNode.c` respectively `monitorNode.c` are the aforementioned main source files. The `CMakeLists.txt` is the main CMake script for the respective node. The file `CMakeXmeComponents.txt` is included by the main CMake script and contains a definition of all components on that node, so that the build system is aware of them.

When clicking on *Generate Application Code...* you can also see that two new views, called *Data Link Graph* and *Schedule*, pop up. First let us have a look at the data link graph view. Screenshot 24 shows a cutout from this view. The data link graph is a representation of the resulting data paths in the application. Beginning from publications and ending at subscriptions it shows the path along which data is sent. First the XMT searches for matching publications and subscriptions (where the topics are equal and attribute filters match). For each match a so called logical route is added. Afterwards these logical routes are refined by adding waypoints. For example you can have a look at the data path from the publication `sensorB.sensorValueOut` to the subscription `monitorB.sensorValueIn`. As they lie on different nodes, the data has to be sent across the network. For this purpose four waypoints have been added, namely: Marshaler, UdpSend, UdpReceive, and Demarshaler. Data sent from the publication first goes to the Marshaler which serializes the data into a platform independent format. Next it will be processed by the UdpSend waypoint which sends the data across the network via UDP to the correct receiver(s). On the receiving node, the UdpReceive waypoint receives the data and passes it to the Demarshaler who deserializes the data again before giving it to the subscription, where it will be read by the application.

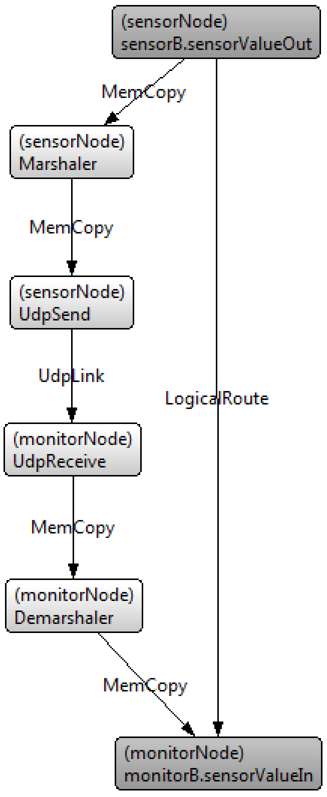


Figure 24: Cutout from Data Link Graph View.

The other new view is the schedule view. Here you can see a visualization of the schedule for each node in the deployment model. For this example it will look like shown in screenshot 25. Each node is displayed on a separate line and each line represents one schedule cycle. Next to the name of a node are colored boxes for each schedule entry. The purple ones correspond to waypoints, the orange one to functions from components. The numbers in the boxes are the slot length in milliseconds and in brackets the number n, where n means that this will be executed only every n-th cycle of the schedule. The length of a box is proportional to the slot length. You can hover the mouse cursor over a schedule entry to see a tool tip with more detailed information.

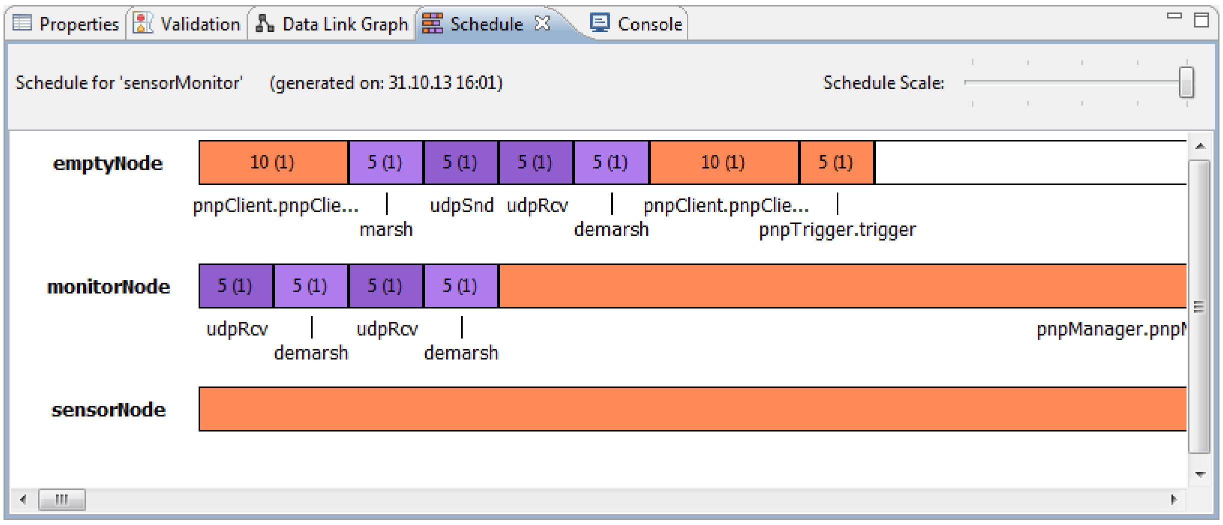


Figure 25: Schedule View.

You may now compile the generated code as presented in the previous section or directly use CHROMOSOME Modeling Tool to build the code as follows: right-click on the *Deployment: deployment* node in the model and select *Build Application...* – see Figure 23.

After clicking on this context menu option, a new window will appear. This window allows to set up the nodes that are going to be compiled. The required CMake build system will be automatically created in this case. Note that the nodes that appear in this window are directly related to the deployment model (compare Figure 26). The first option (*Generate Build System*) will generate all the files needed for building the system[35]. The second option is related to with compilation of the generated build.

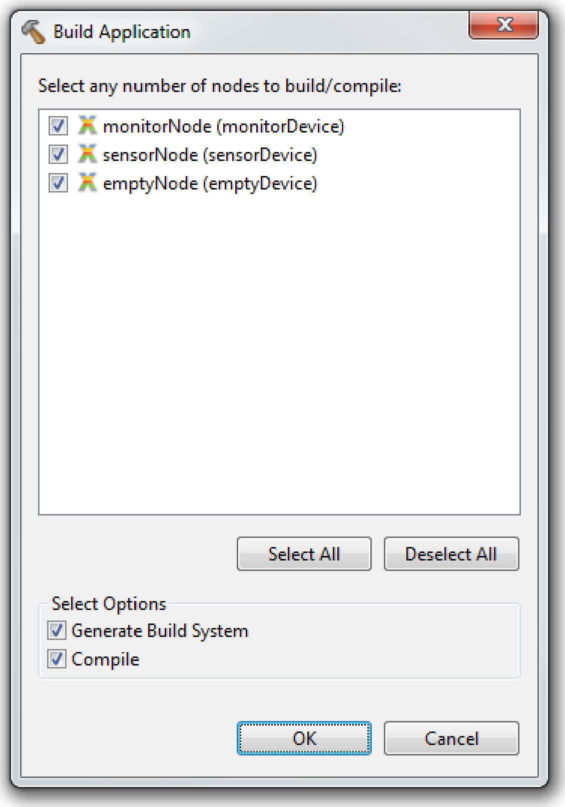
5.5 Conclusion

If you followed all previous steps, you have generated the complete application and should be able to compile and run it again like described in Section 4.

Alternatively you can use also trigger compilation from inside the XMT. For this you first have to enter some CMake related settings. Go to *Window* → *Preferences*. In the preferences dialog expand the *XMT* item and click on the sub-item *CMake*. Set the path to the CMake executable by pressing *Browse...* and navigating to the respective file. Then choose the correct generator from the drop-down list. You only have to do these steps once. Now you can trigger build system generation and compilation. Go to the deployment model and right-click on the deployment item. Choose *Build Application...*. Select the nodes that you want to build and check if you want to generate the build system or compile the selected nodes (or both). When pressing *OK* the tool will start CMake and print its output to the Console view.

You are now free to modify the models, re-generate the source code and change the behavior of the components. One interesting example is to change the IP addresses of the nodes and deploy them on different machines in a local-area network. Notice however that you might need to add respective exceptions in your firewall. The default ports being used in this example are 33221 for *monitor* node, 33222 for *sensor* node and 33223 for *empty* node. More on the topic

[35]Currently, the build system is associated to a build in Visual Studio. In case you want to use a different compiler toolchain, you have to manually adapt the generated `CMakeLists.txt` file.


Build Application

Select any number of nodes to build/compile:

- [✓] monitorNode (monitorDevice)
- [✓] sensorNode (sensorDevice)
- [✓] emptyNode (emptyDevice)

Select Options

- [✓] Generate Build System
- [✓] Compile

Figure 26: XMT build options.

of distributed applications will come in the next release of CHROMOSOME (see our roadmap in Appendix C).

6 Example 3: Plug & Play Showcase (30 minutes)

In Section 3, we shortly explained the planned plug & play features of CHROMOSOME. The current version implements both plug & play and login features. The following showcase illustrates the current state of plug & play implementation.

This example is again based on the previous sensor/monitor example and extends it. The goal is to activate a new sensor component on a third node called "*emptyNode*" and a new monitor on a fourth node called "*monitorKB*". The *emptyNode* is targeted to demonstrate the plug & play and login capabilities in CHROMOSOME, while *monitorKB* is targeted to demonstrate the attribute support during plug & play phase.

In this case, we will use attributes to demonstrate data filtering during plug & play phase of CHROMOSOME. For this purpose, we differentiate between the *monitorNode*, containing one *monitorB* component that filters sensing data received with the measurement unit established to *bytes*, and one *monitorKB* component that filtering data measured in *kilobytes*.

Hence, after the plug & play phase is finished, the *monitorB* component on *monitorNode* will receive values from the *sensorB* component placed on *sensorNode*, and the *monitorKB* component on *monitorNode* is going to receive sensing data from *sensorKB* component located on *emptyNode*. The first sensing data is measured in *bytes*, while the latter is measured in *kilobytes*.

Additionally, in order to demonstrate logout capabilities, it is explained how the system reacts to a node logout received from "*emptyNode*".

6.1 Adding the emptyNode

The *emptyNode* is already defined in the deployment model. Open the model with the *XMT* tool to have a look at it[36] – compare Figure 27. The assignment of nodes to components is as follows:

monitorNode: *loginManager*, *pnpManager*, *pnpClient*, *monitorB* (displays bytes) and *monitorKB* (displays kilobytes).

sensorNode: *pnpClient* and *sensorB*.

emptyNode: *pnpClient* and *pnpTrigger*.

The *pnpTrigger* on *emptyNode* is an example of a tutorial-specific component performing the following actions:

Requests the automatic instantiation of a *sensorKB* component after the *emptyNode* is logged-in.

Request the logout of the *emptyNode* after having demonstrated the other features.

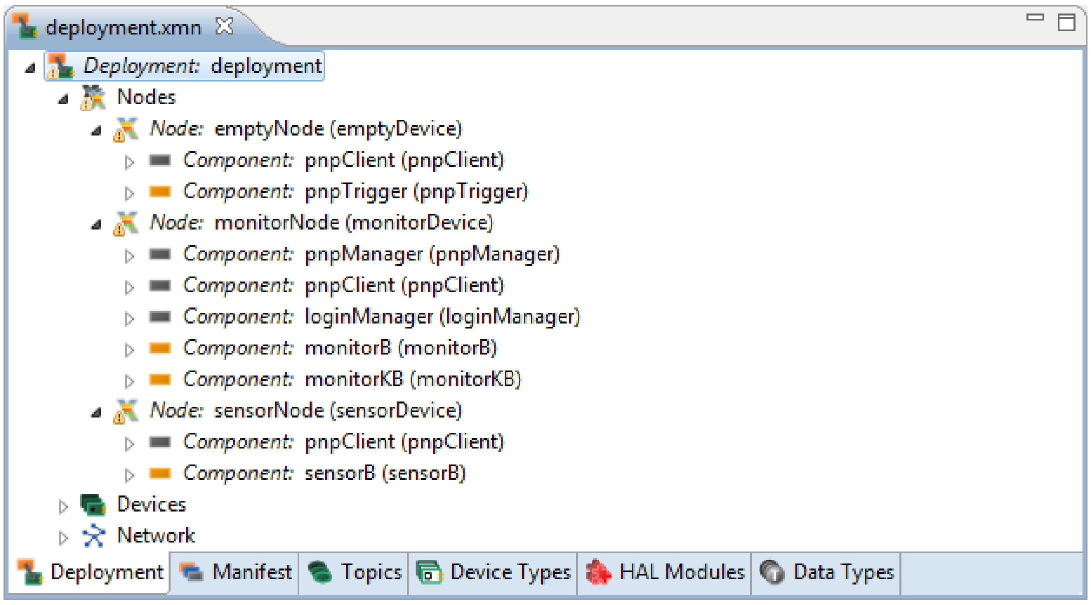
In this subsection, the objective is to demonstrate dynamic plug-in of the *sensorKB* component. The latter will be demonstrated in the following subsection.

The *Plug & Play Manager* component exists at most once in a CHROMOSOME network. This component communicate with *Login Manager* to provide authorization to establish connection between nodes that are already logged in.

Refer to Section 4 for instructions on how to set up the build environment and build the code. To run this example, follow these steps:

1. First start the *monitorNode*. This node contains four key components relevant to plug & play and login process:

[36]See previous example how to do this.


deployment.xmn

- *Deployment:* deployment
  - Nodes
    - *Node:* emptyNode (emptyDevice)
      - *Component:* pnpClient (pnpClient)
      - *Component:* pnpTrigger (pnpTrigger)
    - *Node:* monitorNode (monitorDevice)
      - *Component:* pnpManager (pnpManager)
      - *Component:* pnpClient (pnpClient)
      - *Component:* loginManager (loginManager)
      - *Component:* monitorB (monitorB)
      - *Component:* monitorKB (monitorKB)
    - *Node:* sensorNode (sensorDevice)
      - *Component:* pnpClient (pnpClient)
      - *Component:* sensorB (sensorB)
  - Devices
  - Network

Figure 27: Deployment model highlighting the emptyNode.

- A *monitorB* component, subscribed to any *sensorData* topic sensing data measured in *bytes*.
- A *monitorKB* component, subscribed to any *sensorData* topic sensing data measured in *kilobytes*.
- A *pnpClient* component, listening to requests from *pnpManager* component.
- A *pnpManager* component, which orchestrates the plugin process of additional components to the network, and communication with the *loginManager* to grant access to already logged in nodes.
- A *loginManager* component, which receives requests broadcasted by *Login Client* component and delivers login responses to nodes which requested to login.

2. Start the *sensorNode*. The *sensorNode* contains two components at startup:

   - A *pnpClient* component, listening to requests from *pnpManager* in the network.
   - A *sensorB* component, emitting sensor data in *bytes*.

   Like in the previous examples, the *sensorB* component will ask you which partition to monitor (compare Figure 29). Choose any partition that you want. This will initiate the data production on the sensor side. As we do not have yet any subscriber for the data, the data is not received by any other component.

3. Finally, start the *emptyNode*. This *emptyNode* contains two components:

   - A *pnpClient* component, in charge of sending *component instance manifests* and listening to requests from *pnpManager*.
   - A *pnpTrigger* component, a tutorial-specific component to demonstrate over the time the plug-in of one sensor component and the logout of the *sensorNode*.

The logical steps when *emptyNode* is running are as follows:

1. After five execution cycles, the *pnpTrigger* calls a function in the *pnpClient* to trigger the initiate the plug-in process for a *sensorKB* component in the *emptyNode*.

2. The *pnpClient* on *emptyNode* sends a *Component Instance Manifest* topic message to the *pnpManager* on *monitorNode* to request that the *sensorKB* component should be added.

3. The *pnpManager* receives the *Component Instance Manifest* and process its content.

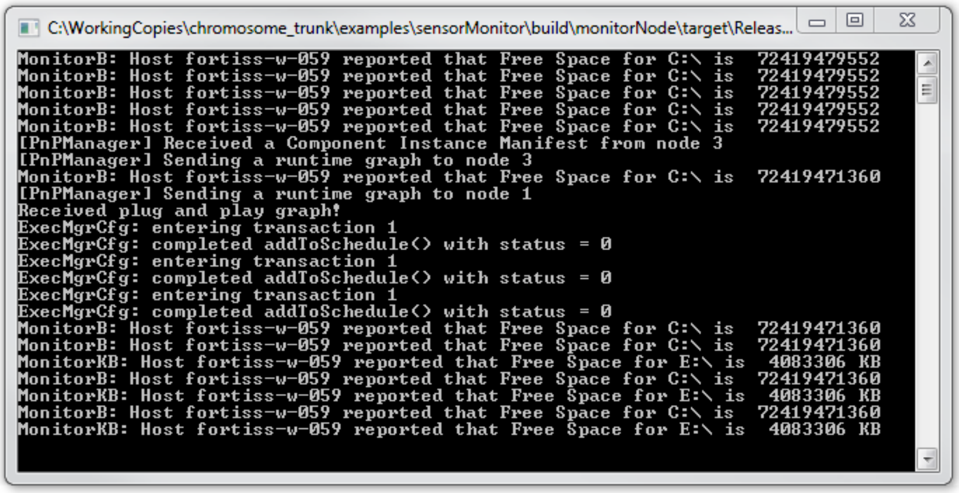
4. The *pnpManager* component calculates internally matches between subscriptions and publications with the current registered nodes and components (compare Figure 28). Additionally, for every matching topic, the attribute filter in subscription should match with attribute definitions associated to that topic in publications.

5. A logical and physical route is calculated by the *pnpManager* component, and the result is sent to the corresponding nodes. In our case, the logical route for the *sensorData* topic between the *sensorKB* component specification on *emptyNode* and the *monitorKB* running component on *monitorNode* is returned back as the result of plug & play process.

6. After receiving the *runtime graph* topic on *emptyNode*, the *sensorKB* component is scheduled and starts running and ask you which partition to observe. Choose any partition – preferably other different partition than the other selected on *sensorNode*. After this, the data production for *sensorData* topic is delivered.

7. After receiving the *runtime graph* topic on *monitorNode*, the *monitorKB* component is scheduled and starts running receiving *sensorData* expressed in *kilobytes*.

8. After this, both sensors (i.e., *sensor – in bytes – on* SensorNode and the newly created *sensor* – in kilobytes – on the (formerly) *emptyNode*) send their measurements to the *monitorB* and *monitorKB* components located on *MonitorNode*. Check the *monitorNode* console window.



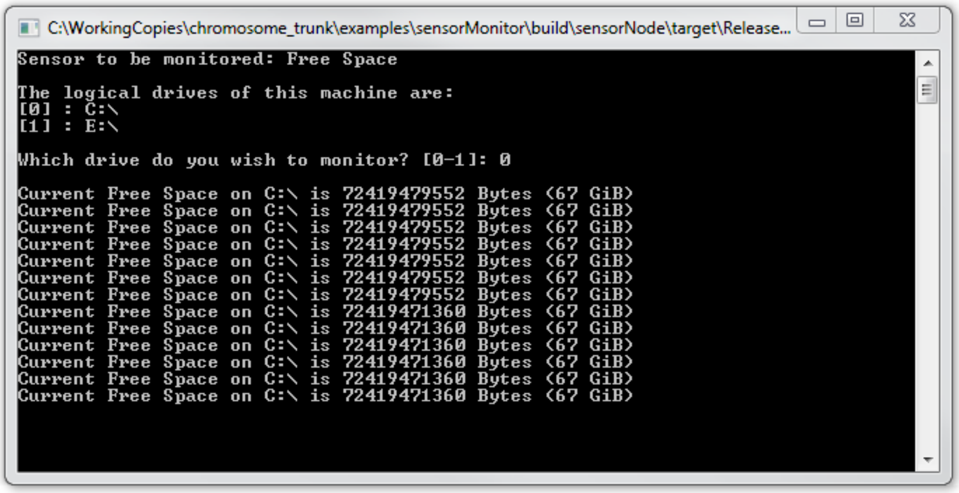
```
C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor\build\monitorNode\target\Releas...
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419479552
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419479552
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419479552
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419479552
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419479552
[PnPManager] Received a Component Instance Manifest from node 3
[PnPManager] Sending a runtime graph to node 3
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419471360
[PnPManager] Sending a runtime graph to node 1
Received plug and play graph!
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419471360
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419471360
MonitorKB: Host fortiss-w-059 reported that Free Space for E:\ is  4083306 KB
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419471360
MonitorKB: Host fortiss-w-059 reported that Free Space for E:\ is  4083306 KB
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419471360
MonitorKB: Host fortiss-w-059 reported that Free Space for E:\ is  4083306 KB
```

Figure 28: Console window of *monitorNode* during plug & play.

There is still one more node that can be added at runtime to the system, the *lonelyNode*. We will explore how to integrate external nodes developed outside the common shared deployment model on *XMT*.

6.2 Self-initiated logout process for emptyNode

In this subsection, it is explained the logout process for a given node, and how to communicate the logout of this node to other running nodes.



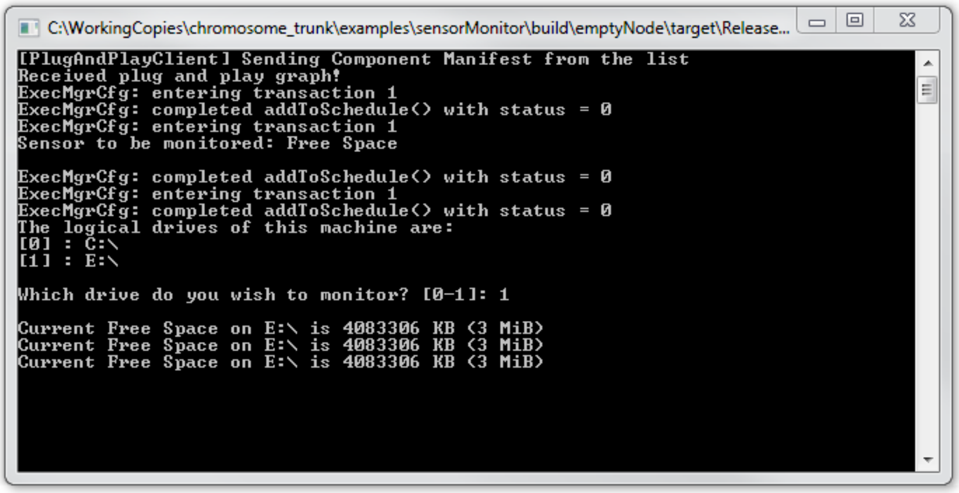
```
C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor\build\sensorNode\target\Release...
Sensor to be monitored: Free Space

The logical drives of this machine are:
[0] : C:\
[1] : E:\

Which drive do you wish to monitor? [0-1]: 0

Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419479552 Bytes (67 GiB)
Current Free Space on C:\ is 72419471360 Bytes (67 GiB)
Current Free Space on C:\ is 72419471360 Bytes (67 GiB)
Current Free Space on C:\ is 72419471360 Bytes (67 GiB)
Current Free Space on C:\ is 72419471360 Bytes (67 GiB)
Current Free Space on C:\ is 72419471360 Bytes (67 GiB)
Current Free Space on C:\ is 72419471360 Bytes (67 GiB)
```

Figure 29: Console window of *sensorNode*.



```
C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor\build\emptyNode\target\Release...
[PlugAndPlayClient] Sending Component Manifest from the list
Received plug and play graph!
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 1
Sensor to be monitored: Free Space

ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
The logical drives of this machine are:
[0] : C:\
[1] : E:\

Which drive do you wish to monitor? [0-1]: 1

Current Free Space on E:\ is 4083306 KB (3 MiB)
Current Free Space on E:\ is 4083306 KB (3 MiB)
Current Free Space on E:\ is 4083306 KB (3 MiB)
```

Figure 30: Console window of *emptyNode* during plug & play.

This logout process for *emptyNode* is a continuation of the execution of the plug-in example of the previous section. The responsible component for initiating the logout process is *pnpTrigger*. The logout process will start after fifty execution cycles.

For the execution of this example, it is expected that the user have already started the *monitorNode*, the *sensorNode* and the *emptyNode*, as described in the previous section.

The logout process for *emptyNode* is executed as follows:

1. The *pnpTrigger* requests to *pnpClient* to trigger the logout process for the current node (in this case, the *emptyNode*).

2. The *pnpClient* generates a logout topic message. This logout topic message is delivered through the network to the *pnpManager* on the *monitorNode*.

3. The *pnpManager* receives the logout message with the node identifier. This will unannounce the node-associated componnent from *pnpManager* (and thus, component ports from *Logical Route Manager*, and calculates the logical routes to be removed.

4. At this point, there are two different set of logical routes:

   - The routes of subscribers obtaining data from the node to disconnect from CHROMOSOME (*emptyNode*). These nodes will receive a *runtime graph* with the directive of removing all physical routes connected to the *emptyNode*.
   - The *emptyNode* will receive all the connected physical routes (subscribers and publishers) to other nodes.

5. The *pnpClient* components in another nodes than *emptyNode*, will reconfigure the scheduling to eliminate the subscription listening coming from *emptyNode* and the publication of messages targeted to *emptyNode*.

6. The *pnpClient* component in *emptyNode*, will receive the confirmation that it can be disconnected safely from CHROMOSOME. Before this complete disconnection, removes all schedules associated to publications and subscriptions, and sends an acknowledgement signal back to *pnpManager*.

6.3 Third-party initiated logout process for sensorNode

The logout process for *sensorNode* is executed in the same was as the logout process in *emptyNode*, with the difference which the process is directly triggered from *pnpTrigger* component in *monitorNode*. The logout process from *pnpTrigger* will start after the seventieth execution cycle. This *pnpTrigger* on *monitorNode* calls directly the logout function in the *pnpManager* to initiate the logout for node *sensorNode*. The following steps are equal to the previous logout process.

In this section, it was demonstrated how to connect and execute non-previously defined components in CHROMOSOME (plug & play), and how to disconnect nodes from the infrastructure at runtime (node logout).

7 Example 4: Integrating External Nodes (20 minutes)

In previous examples, we have generated the code for nodes that have been modeled within the same deployment model in CHROMOSOME Modeling Tool. These nodes were hence known a priori and the generated code is based on this knowledge. This allows to establish all connections from the start.

However, this scenario does not always apply. In case we want to add a new node to an already existing ecosystem of nodes, the new node needs to be developed independently from current established nodes. In this example, we will generate a new deployment model without having information about nodes that are already running in CHROMOSOME.

For modeling the firmware of the new node called *Lonely Node*, we again use CHROMOSOME Modeling Tool. For this purpose, we will start again the Eclipse workspace and import the deployment model *deploymentLonelyNode.xmn* located at `<XME_ROOT>/examples/sensorMonitor` (compare Figure 31). You should note that if you have already imported all models in previous example, the *lonely node* will be already imported in your workspace.

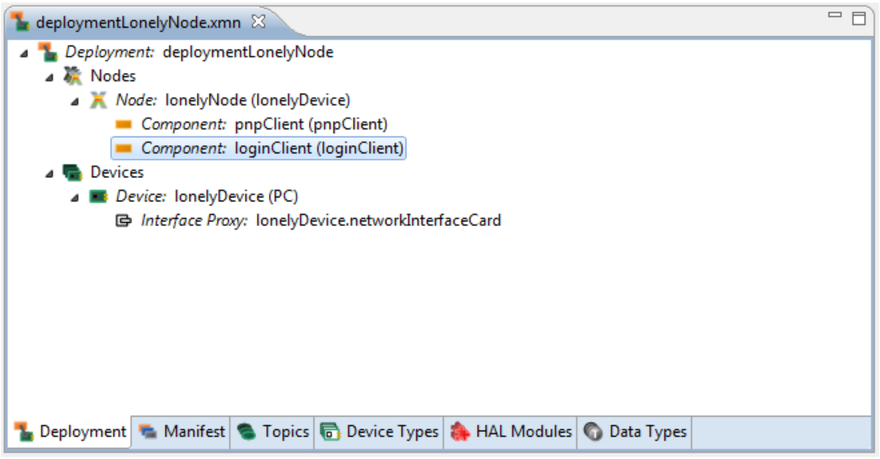

deploymentLonelyNode.xmn
- *Deployment:* deploymentLonelyNode
  - Nodes
    - *Node:* lonelyNode (lonelyDevice)
      - *Component:* pnpClient (pnpClient)
      - *Component:* loginClient (loginClient)
  - Devices
    - *Device:* lonelyDevice (PC)
      - *Interface Proxy:* lonelyDevice.networkInterfaceCard

Figure 31: Lonely Node deployment model.

We assume here that the code for running the *monitorNode*, *sensorNode* and *emptyNode* is already generated and compiled. In case that this is not yet the case, please follow the steps explained in the example in Section 5 before continuing.

7.1 Overview of lonelyNode

For the deployment model of *lonely node*, we already use generated components which are part of CHROMOSOME Modeling Tool, such as *loginClient* and *pnpClient*. These two components play the following roles in this scenario:

- The *loginClient* component establishes a connection with the *Login Manager* (here running on the *monitorNode*) in order to register the *lonelyNode*.
- After login is completed, the *pnpClient* component announces the application components running on the *lonelyNode* to the *Plug and Play Manager* (also running on the *monitorNode*). *Plug and Play Manager* will then establish the required communication routes.

For more information about how the login and plug & play infrastructure works, please refer to Section 6.

As it is generated from CHROMOSOME Modeling Tool, the lonely node contains the code for starting an CHROMOSOME instance of this node. However, as the lonely node does not contain any information about configurations of already running nodes – for example it does not know where the *Login Manager* is located – it needs to discover the network configuration for using the login and plug & play services.

The login operation is performed using broadcasting of a login request and the plug & play operation is completed using the configuration received from the login response, which contains the interface address where the *Plug and Play Manager* is listening to *component instance manifests*.

Now let's generate the code for *lonelyNode*: just like in the previous example, right-click on the *Deployment: deploymentLonelyNode* in the deployment model and click on *Generate Application Code...* – see Figure 32. Confirm the code generation dialog accordingly.

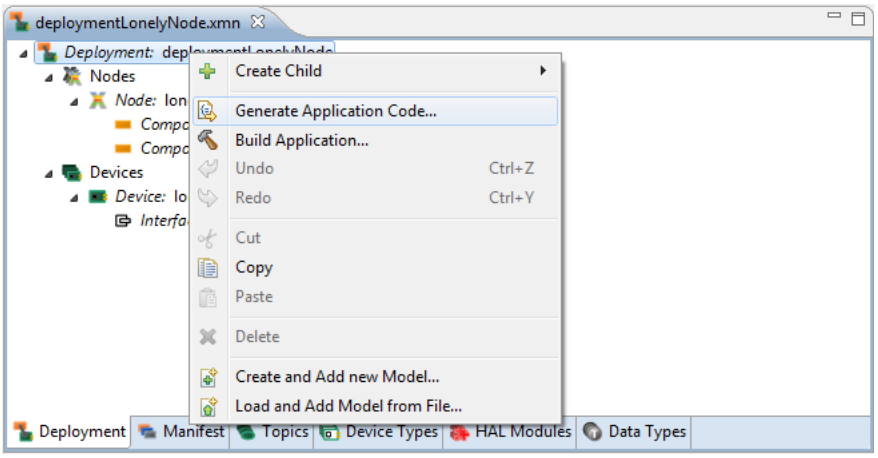


Figure 32: Generation of application code for *lonelyNode*.

In order to compile the generated code, right-click on the *Deployment: deploymentLonelyN-ode* and select *Build Application...* – see Figure 33.

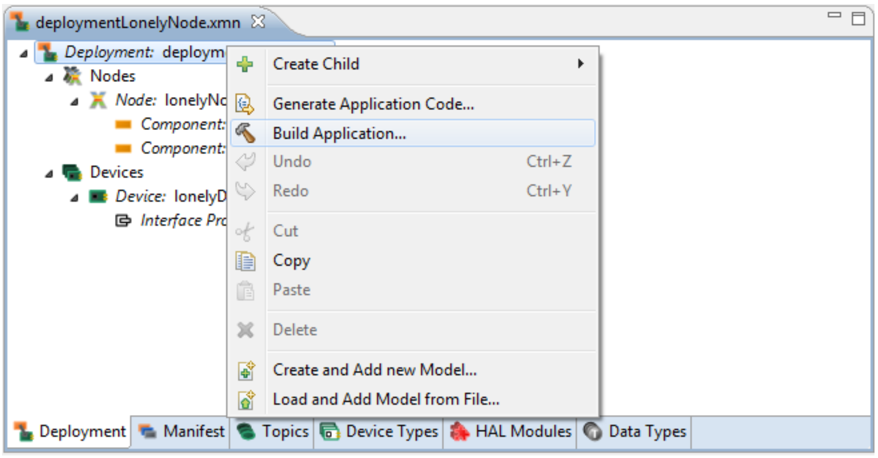
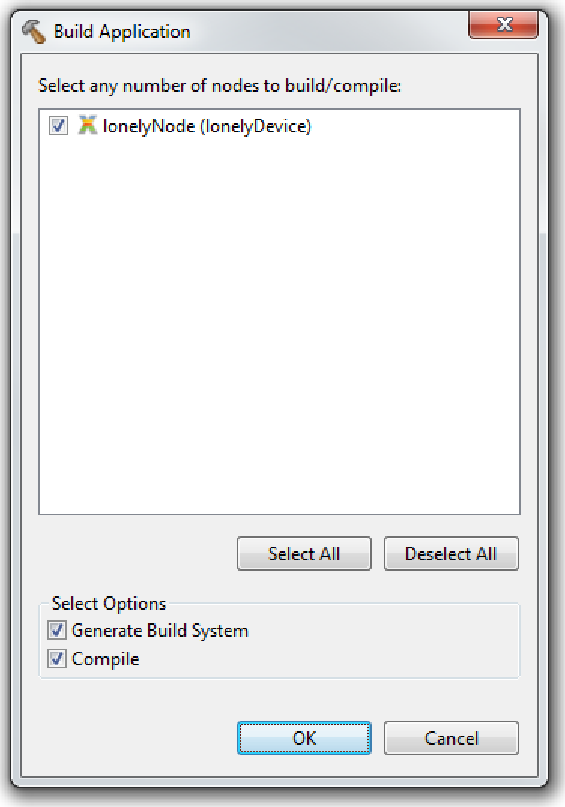


Figure 33: Build of application code for *lonelynode*.

In the build window, select *lonelyNode* and check *Generate Build System* and *Compile* as shown on Figure 34. Then click *OK*.

The output of the build is created in the `build/lonelyNode` folder inside the *sensorMonitor* project in the workspace.


Build Application

Select any number of nodes to build/compile:

☑ lonelyNode (lonelyDevice)

Select Options
☑ Generate Build System
☑ Compile

Figure 34: XMT build options for lonely node.

7.2 Running the lonelyNode

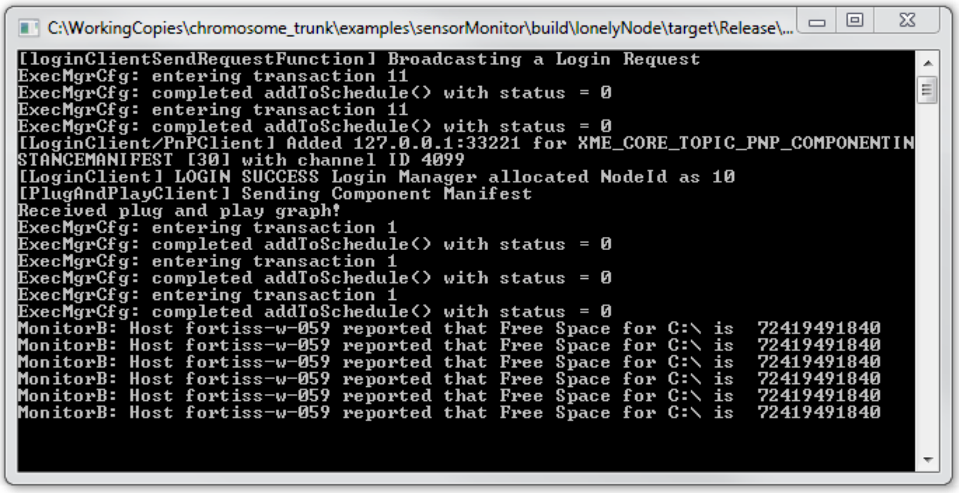
After compilation is finished, the *lonelyNode* is ready to run. Before of running the executable, make sure that at least the node containing *Login Manager* and the *Plug & Play Manager* is currently running. If you followed the instructions in this tutorial, that node is the *monitorNode*.

lonelyNode includes in the manifests for deploying one *sensorB* component.[37] To run the lonely node, just double-click on Eclipse workspace file `/build/lonelyNode/target/Debug/lonelyNode.exe`. As soon as the node is started, it should emit login requests that are handled by the *Login Manager*. After a couple of seconds, login should be complete and the component manifest exchanged. Then, *Plug and Play Manager* should update the communication routes such that the *sensorB* on *lonelyNode* sends its data to the *monitorB* on *monitorNode* (compare Figure 35).

7.3 Conclusion

In this example, we have generated code for a new node which is developed completely independent from from the other nodes in the network. The only shared specification between the nodes is the topic dictionary, which defines the structure of the data being exchanged. The newly node, named *lonelyNode*, completes its registration in *Login Manager* component located at *monitorNode* (or any other node containing the *Login Manager*), exports its *component instance manifest* to the *Plug and Play Manager*, and – in this case – starts the components *sensor* and *monitor* after successful login. In this example, you learned how to develop a new component from scratch, without any connection to an existing infrastructure of CHROMOSOME.

[37]The pnpClient will announce any component type listed in the *pluggable components* attribute of its node. For lonely node this list only contains a *sensorB*.


C:\WorkingCopies\chromosome_trunk\examples\sensorMonitor\build\lonelyNode\target\Release\...

```
[loginClientSendRequestFunction] Broadcasting a Login Request
ExecMgrCfg: entering transaction 11
ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 11
ExecMgrCfg: completed addToSchedule() with status = 0
[LoginClient/PnPClient] Added 127.0.0.1:33221 for XME_CORE_TOPIC_PNP_COMPONENTIN
STANCEMANIFEST [30] with channel ID 4099
[LoginClient] LOGIN SUCCESS Login Manager allocated NodeId as 10
[PlugAndPlayClient] Sending Component Manifest
Received plug and play graph!
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
ExecMgrCfg: entering transaction 1
ExecMgrCfg: completed addToSchedule() with status = 0
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419491840
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419491840
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419491840
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419491840
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419491840
MonitorB: Host fortiss-w-059 reported that Free Space for C:\ is  72419491840
```

Figure 35: Console window of *lonelyNode* during login and plug & play.

8 Example 5: Calculator Server (15 minutes)

The next example demonstrates the use of an alternative communication pattern provided by CHROMOSOME, the so-called request/response (or RR) style of communication. In contrast to publish/subscribe, which has basically a fire & forget semantics, RR works more like client/server: components can send *requests* that can be processed by one or more *request handlers*. On success, the request handler sends a *response* back to the component that issued the request.

For this purpose, two topics are required: a request topic and a response topic. A data packet of type request is sent from the client to the server "on demand" and contains all information necessary for the server to process the request. Subsequently, the server shall reply with a data packet of type response, which contains the server's answer.

Unlike in "normal" publish/subscribe, where all subscribers would receive all kind of data that match their topic, the response is only sent to the component that sent the respective request. Hence, this communication pattern is useful to query a component for a value at an irregular frequency or even once only.

8.1 Inspecting the calculator model

In order to demonstrate the request/response communication pattern, this release of CHROMOSOME ships with a second example, which is called *calculator*.

The scenario in this example is "calculator service" that allows to answer simple calculation queries. The request topic consists of two operands and one of the operations addition, subtraction, multiplication and division. When a request is sent to a calculator server, the server shall reply with the result of the requested operation.

The example contains a calculator server node and two "clients" that send random requests at regular intervals. Both clients and servers display their current status (sent requests, processed requests, and received responses) on their standard output.

In order to inspect the example, import the *calculator* project into XMT as you have previously done it with the *sensorMonitor* example (see Section 5). From the *models* folder, open the file *deployment.xmn*.

Let's first have a look at the topics that are defined in this example by clicking on the *Topics* tab of the loaded deployment (compare Figure 36). As you can see, there is a *calculatorRequest* topic and a *calculatorResponse* topic defined.

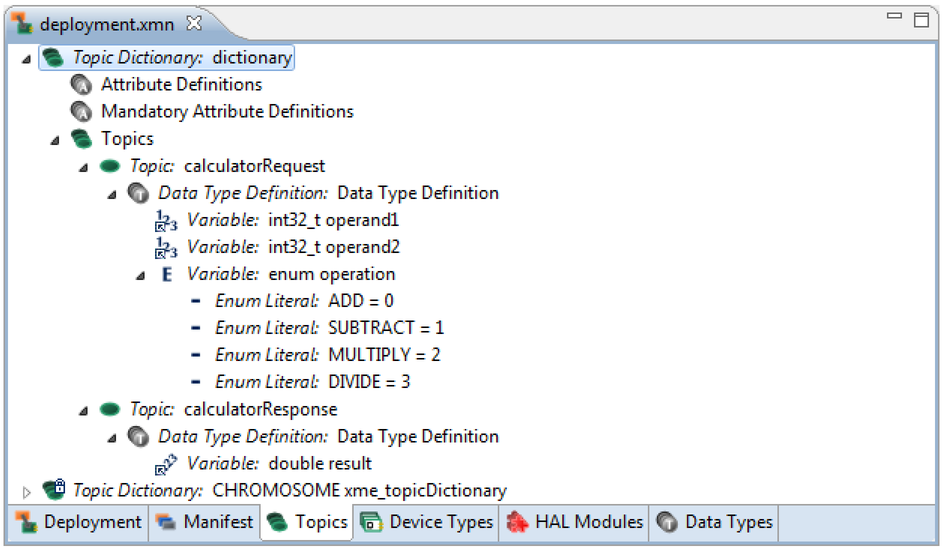
The *calculatorRequest* topic contains two operands of type *int32_t* as well as an enumeration called *operation*, which can be one of the four basic arithmetic operations. The *calculatorResponse* topic just contains a double-precision floating-point value named *result*.

Next, let's have a look at the component manifest of the *calculator* example (compare Figure 37). It lists two application-specific components:

- A *Calculator* component waits for client requests and handles them as they arrive. Furthermore, it provides respective output on the console.

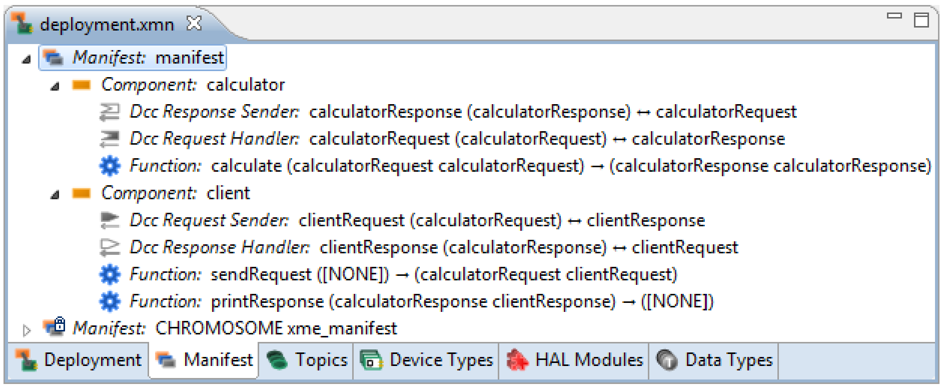
- A *Client* component sends a request using random numbers and operations and displays both request and response in its console.

Using this simple setup, we can build a deployment model with three nodes as shown on Figure 38. Notice that we instantiated the *Client* component twice, both on *client1Node* and on *client2Node*. This means that when we start the applications, requests will be sent to *serverNode* from both client nodes, but a response will only be delivered to the node that actually sent the request. You might wonder why we need two client nodes. Could not we just launch the same compiled node twice? Since the address of a node is hard-coded into its executable in this example, we cannot do that. Otherwise, the server would be unable to distinguish between the



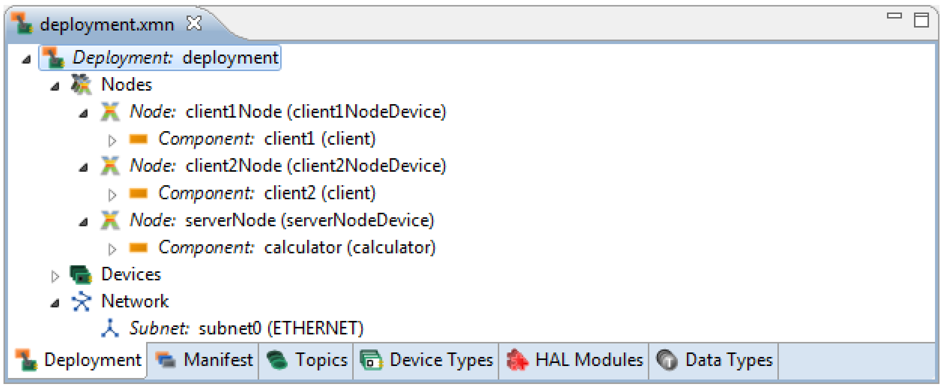
```
Topic Dictionary: dictionary
    Attribute Definitions
    Mandatory Attribute Definitions
    Topics
        Topic: calculatorRequest
            Data Type Definition: Data Type Definition
                Variable: int32_t operand1
                Variable: int32_t operand2
                Variable: enum operation
                    Enum Literal: ADD = 0
                    Enum Literal: SUBTRACT = 1
                    Enum Literal: MULTIPLY = 2
                    Enum Literal: DIVIDE = 3
        Topic: calculatorResponse
            Data Type Definition: Data Type Definition
                Variable: double result
Topic Dictionary: CHROMOSOME xme_topicDictionary
```

Figure 36: Topic model of the *calculator* example.



```
Manifest: manifest
    Component: calculator
        Dcc Response Sender: calculatorResponse (calculatorResponse) ↔ calculatorRequest
        Dcc Request Handler: calculatorRequest (calculatorRequest) ↔ calculatorResponse
        Function: calculate (calculatorRequest calculatorRequest) → (calculatorResponse calculatorResponse)
    Component: client
        Dcc Request Sender: clientRequest (calculatorRequest) ↔ clientResponse
        Dcc Response Handler: clientResponse (calculatorResponse) ↔ clientRequest
        Function: sendRequest ([NONE]) → (calculatorRequest clientRequest)
        Function: printResponse (calculatorResponse clientResponse) → ([NONE])
Manifest: CHROMOSOME xme_manifest
```

Figure 37: Component manifest of the *calculator* example.

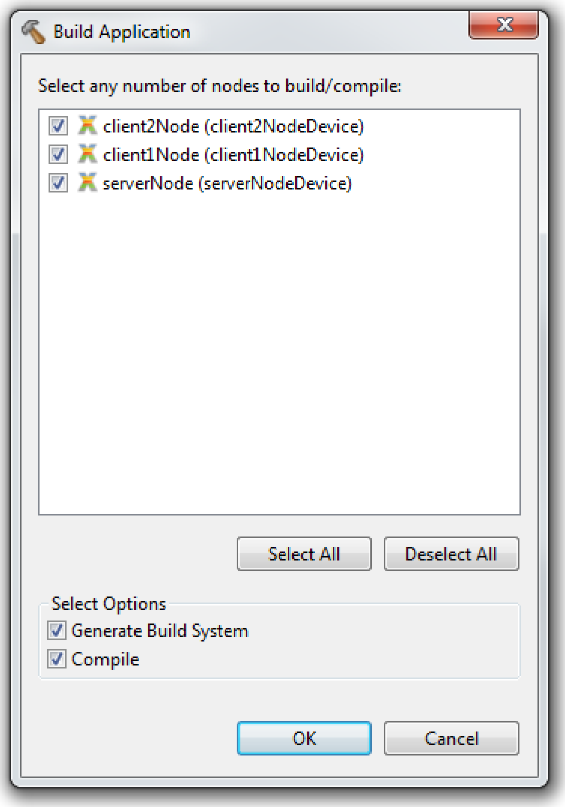


```
Deployment: deployment
    Nodes
        Node: client1Node (client1NodeDevice)
            Component: client1 (client)
        Node: client2Node (client2NodeDevice)
            Component: client2 (client)
        Node: serverNode (serverNodeDevice)
            Component: calculator (calculator)
    Devices
    Network
        Subnet: subnet0 (ETHERNET)
```

Figure 38: Deployment model of the *calculator* example.

two clients and the used UDP port for communication would be already occupied by the other client process.

Now it's time to build the node files and run the example. You should be able to build the nodes from within XMT by right-clicking on the root element in the deployment model and selecting *Build Application...*. In the build window, select the options according to Figure 39. Notice that you need to setup the setting under *Window* → *Preferences* → *XMT* → *CMake* correctly for this to work (using *Visual Studio 10* as generator is recommended if you have Visual Studio 2010 installed). Check the output in the XMT console window for any problems such as error messages.


Build Application

Select any number of nodes to build/compile:

- ☑ client2Node (client2NodeDevice)
- ☑ client1Node (client1NodeDevice)
- ☑ serverNode (serverNodeDevice)

Select Options

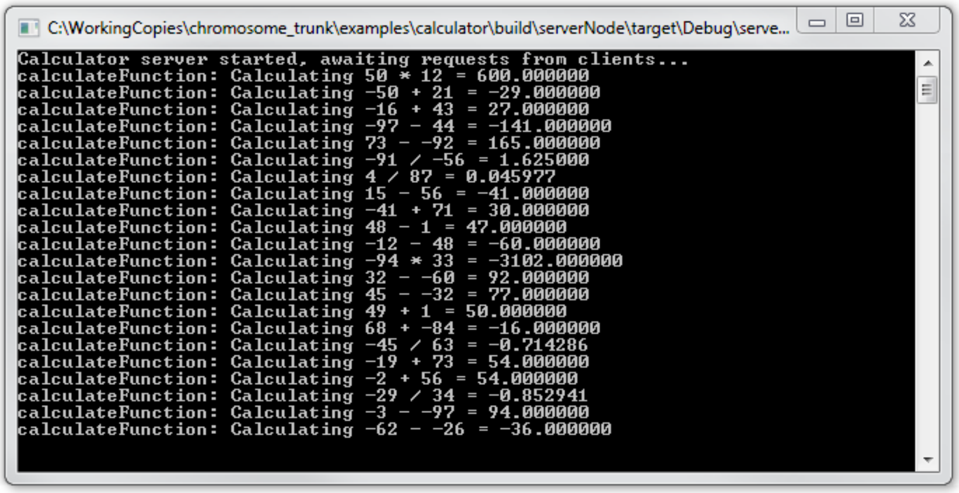
- ☑ Generate Build System
- ☑ Compile

Figure 39: XMT build options for the *calculator* example.

If this does not work as expected, follow the instructions from Section 4.1 (Linux) or 4.2 (Windows) to manually create the build system and compile each of the three nodes (*serverNode*, *client1Node* and *client2Node*).

Now you can find the compiled executables in the following folders, where `<node>` is the name of the respective node: `<XME_ROOT>/examples/calculator/build/<node>/target/`. On Windows, select the subfolder corresponding to the appropriate configuration name (usually *Debug*). You should find an executable in each of these folders, named like the node itself. First run the *serverNode* executable in order to start the calculator "daemon". Subsequently, run *client1Node* and *client2Node* (in arbitrary order) to see them interact with the server. The output should look similar to the one shown in Figures 40, 41 and 42.

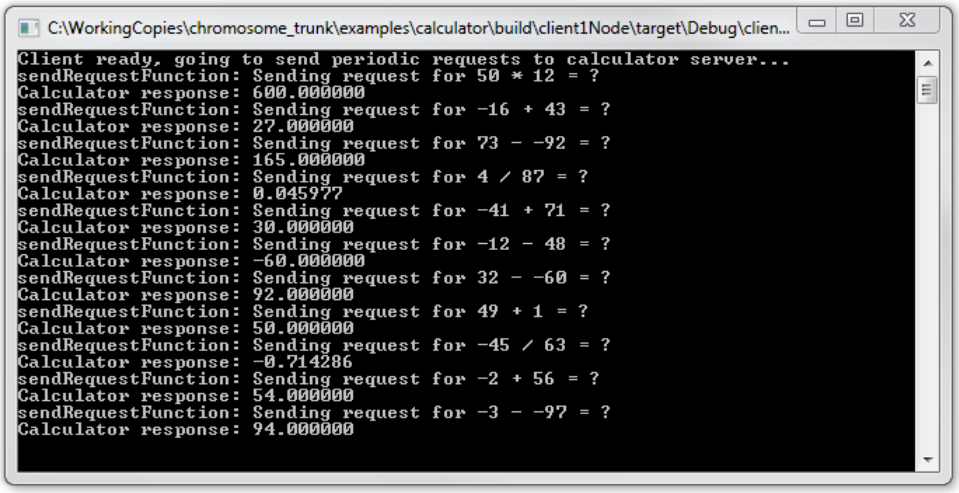
Feel free to inspect the source code of the nodes, which you will find in the directory `<XME_ROOT>/examples/calculator/src`. The individual functions are implemented in the following files inside that directory: `calculator/adv/client/src/sendRequestFunction.c` (*sendRequest* function), `calculator/adv/calculator/src/calculateFunction.c` (*calculate* function) and `calculator/adv/client/src/printResponseFunction.c` (*printResponse* function).



C:\WorkingCopies\chromosome_trunk\examples\calculator\build\serverNode\target\Debug\serve...

```
Calculator server started, awaiting requests from clients...
calculateFunction: Calculating 50 * 12 = 600.000000
calculateFunction: Calculating -50 + 21 = -29.000000
calculateFunction: Calculating -16 + 43 = 27.000000
calculateFunction: Calculating -97 - 44 = -141.000000
calculateFunction: Calculating 73 - -92 = 165.000000
calculateFunction: Calculating -91 / -56 = 1.625000
calculateFunction: Calculating 4 / 87 = 0.045977
calculateFunction: Calculating 15 - 56 = -41.000000
calculateFunction: Calculating -41 + 71 = 30.000000
calculateFunction: Calculating 48 - 1 = 47.000000
calculateFunction: Calculating -12 - 48 = -60.000000
calculateFunction: Calculating -94 * 33 = -3102.000000
calculateFunction: Calculating 32 - -60 = 92.000000
calculateFunction: Calculating 45 - -32 = 77.000000
calculateFunction: Calculating 49 + 1 = 50.000000
calculateFunction: Calculating 68 + -84 = -16.000000
calculateFunction: Calculating -45 / 63 = -0.714286
calculateFunction: Calculating -19 + 73 = 54.000000
calculateFunction: Calculating -2 + 56 = 54.000000
calculateFunction: Calculating -29 / 34 = -0.852941
calculateFunction: Calculating -3 - -97 = 94.000000
calculateFunction: Calculating -62 - -26 = -36.000000
```

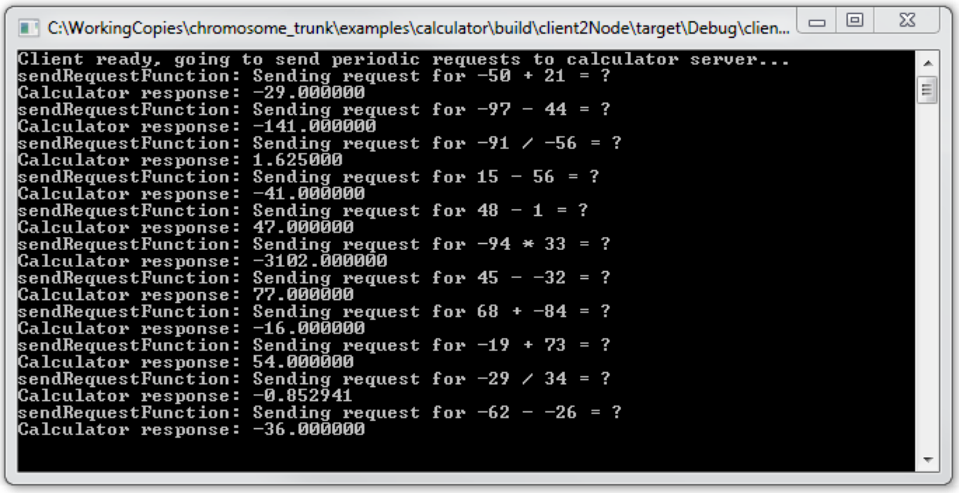
Figure 40: Console window of *serverNode*.



C:\WorkingCopies\chromosome_trunk\examples\calculator\build\client1Node\target\Debug\clien...

```
Client ready, going to send periodic requests to calculator server...
sendRequestFunction: Sending request for 50 * 12 = ?
Calculator response: 600.000000
sendRequestFunction: Sending request for -16 + 43 = ?
Calculator response: 27.000000
sendRequestFunction: Sending request for 73 - -92 = ?
Calculator response: 165.000000
sendRequestFunction: Sending request for 4 / 87 = ?
Calculator response: 0.045977
sendRequestFunction: Sending request for -41 + 71 = ?
Calculator response: 30.000000
sendRequestFunction: Sending request for -12 - 48 = ?
Calculator response: -60.000000
sendRequestFunction: Sending request for 32 - -60 = ?
Calculator response: 92.000000
sendRequestFunction: Sending request for 49 + 1 = ?
Calculator response: 50.000000
sendRequestFunction: Sending request for -45 / 63 = ?
Calculator response: -0.714286
sendRequestFunction: Sending request for -2 + 56 = ?
Calculator response: 54.000000
sendRequestFunction: Sending request for -3 - -97 = ?
Calculator response: 94.000000
```

Figure 41: Console window of *client1Node*.



C:\WorkingCopies\chromosome_trunk\examples\calculator\build\client2Node\target\Debug\clien...

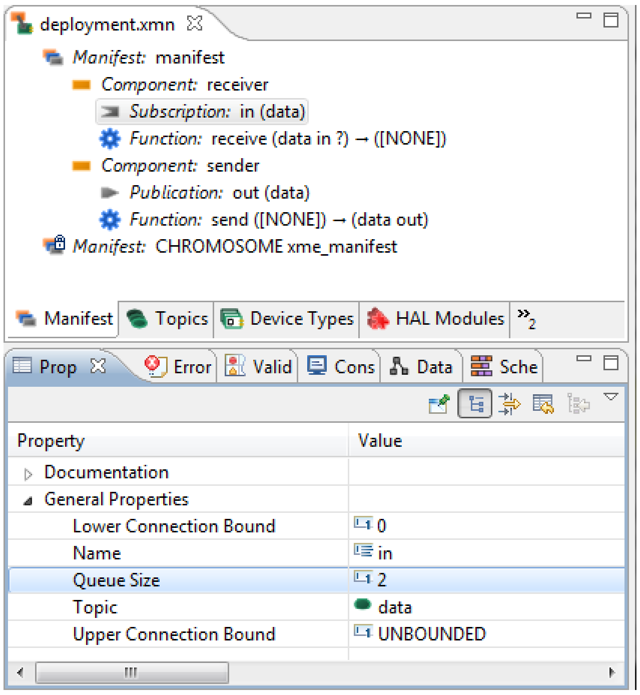
```
Client ready, going to send periodic requests to calculator server...
sendRequestFunction: Sending request for -50 + 21 = ?
Calculator response: -29.000000
sendRequestFunction: Sending request for -97 - 44 = ?
Calculator response: -141.000000
sendRequestFunction: Sending request for -91 / -56 = ?
Calculator response: 1.625000
sendRequestFunction: Sending request for 15 - 56 = ?
Calculator response: -41.000000
sendRequestFunction: Sending request for 48 - 1 = ?
Calculator response: 47.000000
sendRequestFunction: Sending request for -94 * 33 = ?
Calculator response: -3102.000000
sendRequestFunction: Sending request for 45 - -32 = ?
Calculator response: 77.000000
sendRequestFunction: Sending request for 68 + -84 = ?
Calculator response: -16.000000
sendRequestFunction: Sending request for -19 + 73 = ?
Calculator response: 54.000000
sendRequestFunction: Sending request for -29 / 34 = ?
Calculator response: -0.852941
sendRequestFunction: Sending request for -62 - -26 = ?
Calculator response: -36.000000
```

Figure 42: Console window of *client2Node*.

9 Example 6: Consumer with Queue (10 minutes)

This is a simple example of queues used in a subscription. To look at it, import the *queues* example into XMT, as you have previously done it with the *sensorMonitor* example (see Section 5).

Queues on a subscription become necessary when multiple data items are sent to it, before its function(s) can process the previous data item. In this example there are two sender components and one receiver component on the same node. Each sender component has a single publication, and each receiver has a subscription. Therefore two local routes will be constructed that are connected to the subscription. Each sender is executed at a frequency of 1.5 Hz, whereas the receiver is executed twice as often at a frequency of 3.0 Hz. Therefore the receiver will in principle be executed often enough to consume all data items. But it might happen that multiple data items arrive in the same cycle. In this example the senders will each be executed in every other cycle, so that two data items arrive at the receiver in that cycle. The problem is that in a single cycle the receiver will only process a single data item. To resolve this, the subscription of the receiver has its queue size set to *2*, see screenshot 43. So in the subsequent cycle, the receiver will process the second data item (no new data items arrive), which has been buffered in the queue.



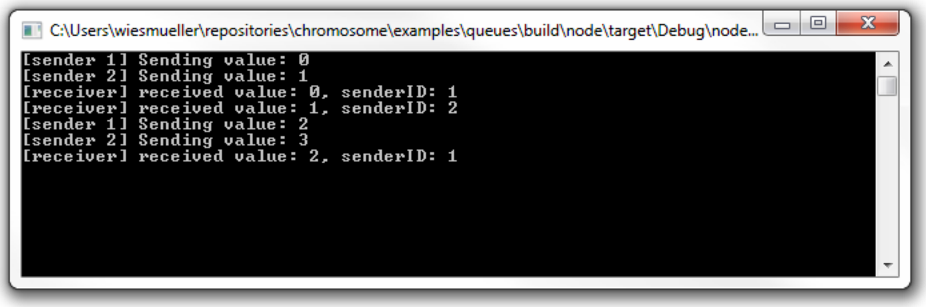
deployment.xmn

- *Manifest:* manifest
  - *Component:* receiver
    - *Subscription:* in (data)
    - *Function:* receive (data in ?) → ([NONE])
  - *Component:* sender
    - *Publication:* out (data)
    - *Function:* send ([NONE]) → (data out)
- *Manifest:* CHROMOSOME xme_manifest

Property	Value
Documentation	
General Properties	
Lower Connection Bound	0
Name	in
Queue Size	2
Topic	data
Upper Connection Bound	UNBOUNDED

Figure 43: Manifest of queues example, highlighting the queue size.

Screenshot 44 shows the console output from the example. On execution you will notice a 3 second delay between each receiver execution. Despite two data items arriving at once it will be able to process both items. Without a queue the second data item would have overwritten the first.


C:\Users\wiesmueller\repositories\chromosome\examples\queues\build\node\target\Debug\node...

```
[sender 1] Sending value: 0
[sender 2] Sending value: 1
[receiver] received value: 0, senderID: 1
[receiver] received value: 1, senderID: 2
[sender 1] Sending value: 2
[sender 2] Sending value: 3
[receiver] received value: 2, senderID: 1
```

Figure 44: Execution of queues example.

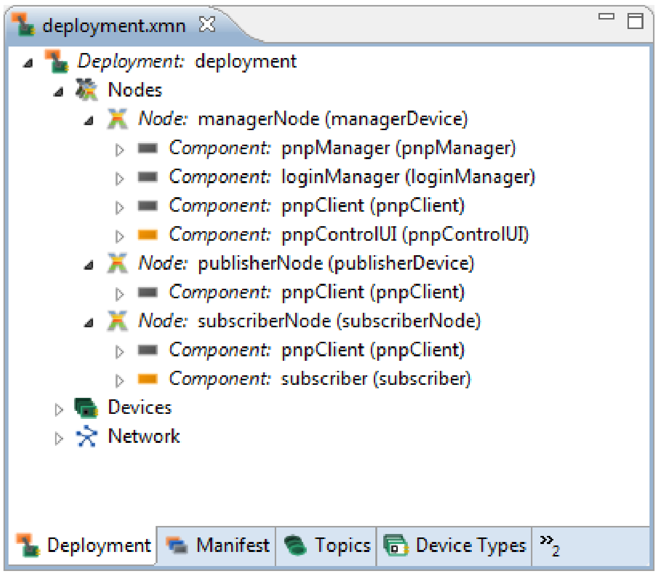
10 Example 7: Configurator Extension (15 minutes)

This examples shows the *configurator extension* concept, together with *lower* and *upper connection bounds* for input ports. It is located in the directory `examples/configuratorExtension/`.

The *configurator extension* component of CHROMOSOME allows to register configurators that react on changes in the system configuration. Currently you can only register configurators for the logical route graph. After each change in the graph, for example after a plug & play event, all registered configurators will be called. In the configurator you can inspect the current routes. You can also add and remove[38] components, as well as removing routes[39].

Connection bounds specify restrictions on the number of routes that an input port is allowed to be connected to. This is specified in the manifest of the component. Currently this information is available at runtime – for example in a configurator – but not enforced yet by CHROMOSOME itself.

This example will register a configurator that uses the connection bound specification and modifies the configuration accordingly. Three nodes have been created for this showcase, see Figure 45.



```
Deployment: deployment
  Nodes
    Node: managerNode (managerDevice)
      Component: pnpManager (pnpManager)
      Component: loginManager (loginManager)
      Component: pnpClient (pnpClient)
      Component: pnpControlUI (pnpControlUI)
    Node: publisherNode (publisherDevice)
      Component: pnpClient (pnpClient)
    Node: subscriberNode (subscriberNode)
      Component: pnpClient (pnpClient)
      Component: subscriber (subscriber)
  Devices
  Network
```

Figure 45: Deployment model of configurator extension example, showing all three nodes.

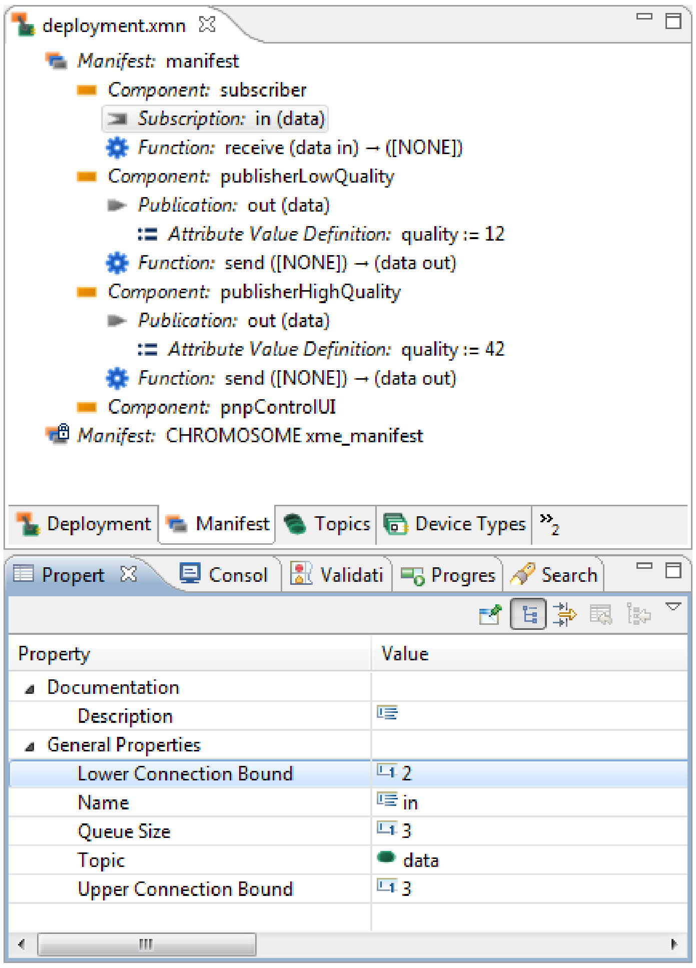
The *subscriberNode* contains a single application-defined component of type *subscriber*. This component simply waits for data on its input port and prints it to the console. When you navigate to the subscription of this component in its manifest – see Figure 46 – you can see that it specifies a lower connection bound of 2 and an upper connection bound of 3.

The *publisherNode* is initially empty except for the *pnpClient*. New components will be added to this node at runtime.

The component *pnpControlUI* on the *managerNode* will prompt the user for plug & play-related commands. We will use this interface to add new *publishers*. There are two kinds of publishers, the *lowQualityPublisher* and the *highQualityPublisher*. Both publish the same topic but assign different values to the *quality* attribute. Again you can have a look at these definition in the manifest, see Figure 46.

In this example a configurator is registered at the configurator extension. See Listing 1 for the respective call in the `managerNode.c` file. The implementation of the configurator can be found in the following directory:

[38]Not supported yet.
[39]Only supported for non-established routes.



deployment.xmn

- *Manifest:* manifest
  - *Component:* subscriber
    - *Subscription:* in (data)
    - *Function:* receive (data in) → ([NONE])
  - *Component:* publisherLowQuality
    - *Publication:* out (data)
      - *Attribute Value Definition:* quality := 12
    - *Function:* send ([NONE]) → (data out)
  - *Component:* publisherHighQuality
    - *Publication:* out (data)
      - *Attribute Value Definition:* quality := 42
    - *Function:* send ([NONE]) → (data out)
  - *Component:* pnpControlUI
- *Manifest:* CHROMOSOME xme_manifest

Property	Value
Documentation	
Description	
General Properties	
Lower Connection Bound	2
Name	in
Queue Size	3
Topic	data
Upper Connection Bound	3

Figure 46: Manifest of configurator extension example.

Listing 1: Registration of configurator.

```
// PROTECTED REGION ID(MANAGERNODE_MAIN_RUN_BEFORE) ENABLED START

xme_core_pnp_configExt_addConfigurator
(
    XME_CORE_PNP_CONFIG_EXT_CONFIGURATORTYPE_LOGICAL_ROUTES,
    &configuratorExtension_configurator_demoConfigurator_callback,
    NULL
);

// PROTECTED REGION END
```

`src/configuratorExtension/configurator/demoConfigurator`. The given configurator, represented by the callback function `configuratorExtension_configurator_demoConfigurator_callback()`, will perform the following two tasks:

1. If any input port is connected to less routes than specified in its **lower connection bound**, all routes from this port are removed. If a new component is to be created that does not satisfy the lower connection bound on one of its ports, component creation will be skipped.

2. If any input port is connected to more routes than specified in its **upper connection bound**, the excessive routes are removed. Routes with lower quality attribute values are removed first. A route with a topic that does not have this attribute is treated as having a quality value of 0.

   Note: The removal of already established routes is currently not yet supported, hence the routes of components that are already created will not be removed. See Issue #3952.

To see this behavior in action, run all three nodes and follow these instructions. You may invoke the `list nodes` and `list types` commands to list active nodes and types, respectively. (the numbers refer to the red numbers in Figure 47).

1. The managerNode asks you to enter a command.
   Enter: `add 2 4100`
   This will add a new component of type *highQualityPublisher* (type ID `4100`) on the *publisher* node (node ID `2`).

2. The configurator informs you that the lower connection bound of the subscriber component is not satisfied. Therefore it removes the route and the component is not created yet (no output appears on the other nodes).

3. Now enter: `add 2 4099`
   This will add a new component of type *lowQualityPublisher* (type ID `4099`) on the publisher node.

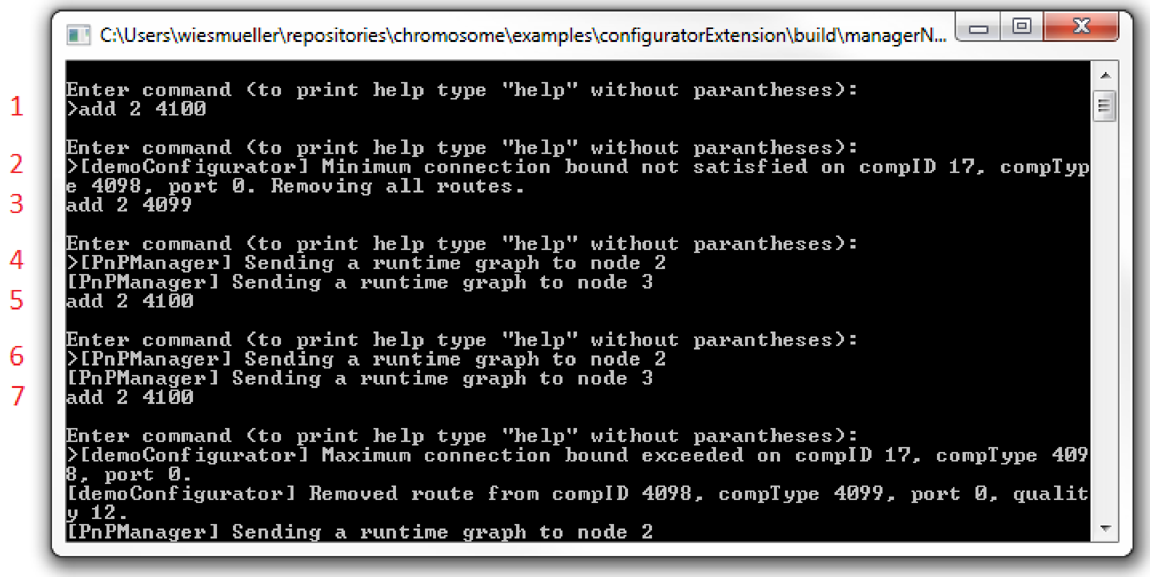
4. Now the lower connection bound is satisfied and the new component is plugged. After waiting some time you will notice that both publishers start sending.

5. Now enter: `add 2 4100`

6. An additional *highQualityPublisher* is added and will immediately be plugged in, as we now have three routes which is in the limits of the connection bounds of the subscriber.[40]

[40] Currently having two instances of the same component type will lead to data loss. That is why you see a warning on the publisher node after instantiating the second *highQualityPublisher*. See also known issue #3964 A.

7. When you subsequently enter: `add 2 4100`
   The configurator will inform you that the maximum connection bound is exceeded and that it is going to remove the route of the *lowQualityPublisher*. As previously noted, established routes cannot be removed yet, so the removal will not actually be executed.


C:\Users\wiesmueller\repositories\chromosome\examples\configuratorExtension\build\managerN...
1
2
3
4
5
6
7

```
Enter command (to print help type "help" without parantheses):
>add 2 4100

Enter command (to print help type "help" without parantheses):
>[demoConfigurator] Minimum connection bound not satisfied on compID 17, compTyp
e 4098, port 0. Removing all routes.
add 2 4099

Enter command (to print help type "help" without parantheses):
>[PnPManager] Sending a runtime graph to node 2
[PnPManager] Sending a runtime graph to node 3
add 2 4100

Enter command (to print help type "help" without parantheses):
>[PnPManager] Sending a runtime graph to node 2
[PnPManager] Sending a runtime graph to node 3
add 2 4100

Enter command (to print help type "help" without parantheses):
>[demoConfigurator] Maximum connection bound exceeded on compID 17, compType 409
8, port 0.
[demoConfigurator] Removed route from compID 4098, compType 4099, port 0, qualit
y 12.
[PnPManager] Sending a runtime graph to node 2
```

Figure 47: Execution of configurator extension example.

A Known Issues

The following known issues exist in this release. They will be fixed in upcoming releases.

A.1 Issue #2968: XMT: Data sinks not scheduled multiple times for multiple incoming data stream

Code generated from XMT has the same issue than described in #2967.

A.2 Issue #3039: Separate the various XME applications in the same network from each other

As soon as multiple XME "master" nodes (nodes with *Login Managers*) run in the same network (e.g., during testing of various applications) at the same time, XME will be come nondeterministic. The reason for this is that a client will connect to any available master and it can not be guaranteed that all nodes of an application will belong to the same XME network.

As a simple solution we could include the same GUID into all nodes belonging to the same network. If the GUID is zero, it will be have like now. A client will connect to the first master available. But it the GUID is not zero, a client will only connect to the master carrying the same GUID.

A.3 Issue #3076: Decide upon action if a function exceeds its timeslice in the schedule

Currently, a function can block the execution of other functions: if a function overshoots its worst case execution time (WCET), currently no action is taken. This means that the other function following the "blocker" in the schedule will be delayed. When the "blocker" finally ends, the other functions will be repeatedly executed in quick succession until the delay is compensated.

A.4 Issue #3183: Monitor losing data

When sensor and empty nodes are running together the monitor node loses data. So we do not see all updates sent by the nodes on Monitor console.

The reason for this is that data gets internally overwritten on the receiver side if more data gets received than can be handled within one cycle of execution. Queues have been recently introduced to cope with this problem, but configuration of the queue size is still a manual task (can be performed in XMT, see *queues* example for illustration). It is planned to automatically calculate the required queue size based on a specification of the expected input behavior.

A.5 Issue #3374: Compiler warnings when building under Visual Studio 2012

```
warning C28251: Inconsistent annotation for '...': this instance has no
annotations. See ...
```

A.6 Issue #3514: Scheduling algorithm to address the duration/frequency of functions additional to WCET

With this feature the scheduling principle in the *Plug and Play Client* for the components received in the runtime graph have to be modified. This also may apply to the data structure of runtime graph which is communicated across from *Plug and Play Manager* to *Plug and Play Client*.

A.7 Issue #3548: Support for multiple request/response relationships per component

The *Logical Route Manager* is currently unable to correctly handle multiple request/response communication relationships in a single component. The reason for this is that if multiple request or response ports exist, the LRM is unable to determine which request port belongs to which response port.

Consider the example of a ”server” component with two request handler ports and two response sender ports with different request and response topics. In this case, channel mappings might get generated from both input ports to both output ports, which is not the intended behavior. In order to resolve this problem, we need to:

- Either introduce a relationship between input port and output port (index) for every pair of request and response ports.
- Or derive the information from the dependency of the associated functions, if possible. This would require that functions are uniquely associated with at most one request and one response port. However, this is probably not meaningful, because the execution semantics of request sending functions and response handling functions is different and hence one might want to put them into two separate functions.

A.8 Issue #3683: “Generate Build System” of “Build Application...” in deployment model does not work under Linux

When triggering the action commands are executed, but the build system is not generated. It seems that there are “”-marks missing surrounding the cmake call with path and the `-G` option.

A.9 Issue #3727: Instantiating same component type twice at runtime causes error

Calling `xme_core_pnp_pnpClient_instantiateComponentOnThisNode()` twice on a node for the same component type fails.

A.10 Issue #7344: Input port used by multiple functions leads to nondeterministic behavior

This leads to undeterministic behaviour. Only one of the functins (“randomly” chosen depending on the currentl slot in the Execution Manager) will actually process data on the input port. The first function calls completeReadOperation on that port, so that the other function(s) do(es) not receive the data.

A.11 Issue #3779: pnpInit in `<node>.c` does not work if node contains {pnp, login}{Client, Master}

If the node identifier of `xme_core_pnp_pnpManager_instantiateComponentOnNode()` in `pnpInit` of `<node>.c` is 0 (dynamic case), you receive the following run-time error:

```
xme_core_pnp_pnpManager_instantiateComponentOnNode component failed.
Fatal: Error occurred during initialization of CHROMOSOME pnp components.
Aborting execution.
```

A.12 Issue #3833: Data structures not completely freed after the required components are removed

Consider the following scenario: Node A : running pnpManager (*Plug and Play Manager*) Node B : running pnpClient (*Plug and Play Client*), sensorComponent in that order in the schedule Node C : running pnpClient (*Plug and Play Client*), monitorComponent in that order in the schedule Node B and C have a route between sensorComponent and monitorComponent.

Node B is logged out, Node A sends respective graphs to Node B and Node C. Node B shuts down XME. For Node C, pnpClient deletes the function descriptor of monitorComponent and removes the monitorComponent from schedule. The new schedule gets only activated from the next cycle. Since monitorNode is run after the pnpClient in the current active schedule, if we delete the function descriptor this causes EM to see a fault and it issues a shutdown of the node C. This is not correct. Hence currently XME does not free the data structures. This is a memory leak.

A.13 Issue #3834: Nodes can only be logged out one at a time

Currently we can logout one node in one execution cycle. If multiple nodes are submitted for logout only the last one will be logged out. Request for the rest will be overwritten.

A.14 Issue #3836: Incorrect stop semantics may lead to loss of data

Currently after we receive a logout or stop a component request, XME simply tries to take he required action as soon as possible. This may result in loss of data, if there is any, in the ports of the components. This may be undesirable in certain cases and one must have some policies regarding the same. This becomes even more important when we have queues.

A.15 Issue #3848: `XME_ASSERT_NORVAL` in configureSchedule of `<node>.cpp` raises unexpected exception in C++ mode under MSVC

For the `XME_ASSERT_NORVAL`s in `configureSchedule()` of the node files the warning C4297 is raised if compiling in C++ mode.

A.16 Issue #3964: Schedule output-related waypoints immediately after associated component

Currently component instances of the same type share their ports. The schedule calculation at runtime does not take this into account. This leads to each component instance overwriting the data of the one scheduled before it. On the console this triggers a warning of data loss.

A.17 Issue #4205: Overwriting of master CMakeLists.txt when using more than one deployment model

When using more than one deployment model, regeneration of files leads to overwriting of master `CMakeLists.txt` in application root directory.

A.18 Issue #4262: When calling logout of a node the node shall be stopped

If a node has logged in before, but the manager node vanished in the meantime, a logout request sent by the node will timeout and the node will never shut down.

B Frequently Asked Questions

B.1 Organizational Questions

B.1.1 What is the CHROMOSOME roadmap/release plan?

See Appendix C.

B.1.2 Can I use CHROMOSOME in commercial software?

Yes. Our licensing conditions allow this (see Appendix G). Notice however that the software is provided "as-is".

B.1.3 Can I contribute?

Yes. In case you are interested in contributing the CHROMOSOME code base, let us know. Find contact information on the first page of this document.

B.2 Technical Questions

B.2.1 How can I port CHROMOSOME to my own target platform?

The current release of CHROMOSOME supports only Linux and Windows. However, we are in the process of developing platform support layers for other platforms. These include, for example, medium-sized microcontroller platforms like ARM Cortex M3 (e.g., as central control units). Upcoming releases of CHROMOSOME will ship with the respective functionality. Let us know if you are interested in these topics.

If you favorite platform is not on this list, there are multiple options:

1. You can suggest CHROMOSOME to be ported onto your platform. If your application is of high relevance, we might indeed port CHROMOSOME to that platform.

2. You can contribute to the development of CHROMOSOME and have your platform supported as a target platform.

3. You can take the CHROMOSOME source code and implement the respective platform support without revealing the source code. Our license model permits this (see also Appendix G).

B.3 Conceptual Questions

B.3.1 How do you want to implement semantic aspects?

Current middleware technology lacks semantic integration. We are using a data-centric design, where we describe the data with an ontology. The *topic dictionaries* that are a part of the XMT models allow to clearly separate topics of different domains. If several domains are involved, there might be the necessity to map between the domain-specific ontologies. This is currently done manually, but can be reused (and must therefore be done only once).

B.3.2 How do you specify extra-functional requirements?

We use meta-data to describe concrete data sources. Examples for meta-data are accuracy, confidence level, age. In addition, we want to use application patterns to describe the extra-functional properties of concrete applications.

B.3.3 How do you guarantee extra-functional requirements?

We focus on implementations that can easily guarantee these requirements (e.g., time-triggered execution). If we have more flexible protocols, we have two options: either we cannot guarantee requirements or we have to be more pessimistic.

C Roadmap

This roadmap illustrates the features that are planned for the upcoming releases of CHROMOSOME.

C.1 Version 0.1

Release: internal only

Features:

- Prototypic API

C.2 Version 0.2

Release: 2012-04-02 (Release)

Features:

- Prototypic implementation of data centric communication with dynamic login of nodes.
- Software components state their resource requirements and data dependencies only during runtime.
- As such, the feasibility of the system can not be determined in advance.
- Network communication is based on multicast UDP.
- Routes are dynamically calculated during runtime.
- No own scheduler; OS scheduler is wrapped.
- New components can be dynamically logged into the system during runtime.

Components:

- *Broker* (prototypic)
- *Routing Table* (prototypic)
- *Directory* (prototypic)
- *Login Manager*, *Login Client*, *Login Sever Proxy*, *Login Client Proxy* (all prototypic)
- *Node Manager* (prototypic)
- *Resource Manager* (prototypic)

Platform support: Windows (primary), Linux (inofficial), embedded (inofficial)

Communication protocols: Multicast UDP, no marshaling support

Example applications: Chat example

C.3 Version 0.3

Release: 2013-05-21 (RC1), 2013-05-29 (Release)

Features:

- Completely redesigned API.
- Implementation of data centric communication based on modeling (CHROMOSOME Modeling Tool, XMT) and static generation/configuration of the runtime system by code generation.
- Software components provide an abstract description of their resource requirements via a so-called manifest.
- No dynamic reconfiguration at runtime.
- As such, feasibility of the system could be checked in advance, although this check is not implemented in this version.
- Own scheduler with time-triggered behavior.

Components:

- *Execution Manager*
- *Broker*
- *Data Handler*

Platform support: Linux (primary), Windows (secondary)

Communication protocols: UDP with static addresses, including marshaling

Example applications: Sensor data acquisition from hosts in network

C.4 Version 0.4

Release plan: 2013-07-04 (RC1); 2013-07-17 (Release)

Features:

- Restricted dynamic reconfiguration during runtime: All nodes in the network are known in advance, but software components can be dynamically added (not removed) during runtime.
- Respective automatic calculation of network data paths based on topics (optionally with support of attributes).
- The addresses of all nodes in the network are specified in the modeling tool. No dynamic login or logoff during runtime.
- Plug & Play Manager exists on dedicated node in the network.
- Plug & Play Clients exist on all nodes in the network.
- Plug & Play Manager accepts command from user that indicate changes in the configuration of the application or network (add new node, add new component on node).
- Network topology known in advance (address is known, no dynamic login), no removal of nodes/components.

Components:

- *Plug & Play Manager*
- *Logical Route Manager*
- *Network Topology Calculator*

Platform support: Linux (primary), Windows (secondary)

Communication protocols: UDP with static addresses, including marshaling

Example applications: Sensor data acquisition from hosts in network with dynamic activation of additional data sources

C.5 Version 0.5

Release plan: 2013-08-02 (RC1); 2013-08-27 (RC2); 2013-09-02 (Release)

Features:

- Restricted dynamic reconfiguration during runtime: Nodes can dynamically log into the network during runtime (not removed/logged out).
- Support for data path calculation based on attributes in XMT.
- After login, nodes tell Plug & Play Manager their manifest (which components are already installed on the node).
- "Management routes" built dynamically during runtime.
- Star-topology based communication scheme.

Components:

- *Login Manager*
- *Login Client*

Platform support: Linux (primary), Windows (secondary)

Communication protocols: UDP with static addresses, including marshaling

Example applications: Sensor data acquisition from dynamically "plugged" nodes in network

C.6 Version 0.6

Release plan: 2013-10-01 (RC1); 2013-10-16 (RC2); 2013-10-31 (Release)

Features:

- Support for data path calculation based on attributes in XME runtime system.
- Native support for request/response (client/server based communication pattern).
- Cleanup and refactoring.

Components:

- *(none)*

Platform support: Linux (primary), Windows (secondary)

Communication protocols: UDP with static addresses, including marshaling and broadcast UDP

Example applications: Sensor data acquisition from dynamically “plugged” nodes in network; Request-response calculator example

C.7 Version 0.7

Release plan: 2013-12-13 (RC1); 2013-12-20 (Release)

Features:

- Logout at node level (opposite of node login).
- Interaction with ROS (ROS Gateway).
- Configurator extension ”plugins” for building custom data paths.
- Subscription cardinality specification.
- Manifest interchange format based on XML.
- Cleanup and refactoring.

Platform support: Linux (primary), Windows (secondary)

Communication protocols: UDP with static addresses, including marshaling

Example applications: Sensor data acquisition with exchange to/from ROS

C.8 Version 0.8

Release plan: 2014-03-14 (RC1); 2014-03-28 (Release)

Features:

- “Unplugging” of components.
- Redesigned Data Handler.
- Cleanup and stability improvements.

Platform support: Linux (primary), Windows (secondary)

Communication protocols: UDP with static addresses, including marshaling

Example applications: Sensor data acquisition with dynamic loading of sensor and monitor software to previously ”blank” nodes

C.9 Version 0.9

Release plan: June 2014 (RC1); June 2014 (Release)

Features:

- Microcontroller support (ARM Cortex M3).
- Binary loading for non-microcontroller platforms (based on shared objects).
- XMT: Limited support for Models@Runtime (live feedback of current application state to XMT).

Components:

- *Binary Manager*

Platform support: Linux (primary), Windows (secondary), ARM (tertiary)

Communication protocols: UDP with static addresses, including marshaling

Example applications: Sensor data acquisition with dynamic loading of sensor and monitor software to previously "blank" nodes; microcontroller example application

C.10 More

The following items are planned, but not yet assigned to a milestone on the roadmap:

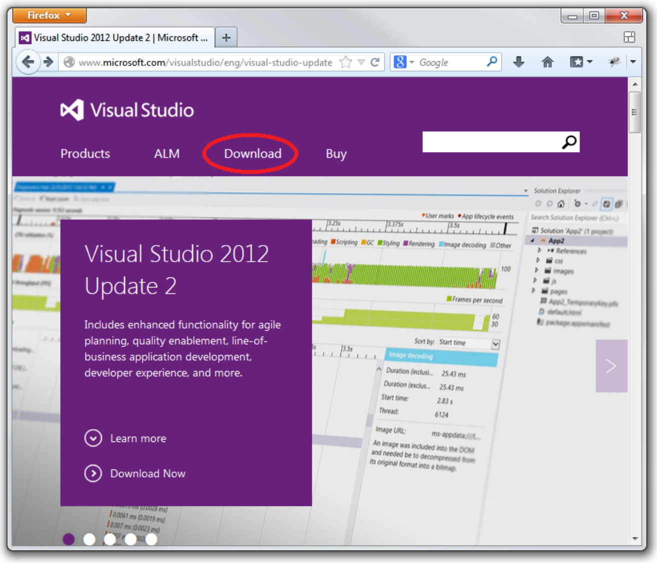
- Reliable communication (reliable UDP?).
- Native event driven scheduling and execution.
- Health monitoring (including runtime self-tests).

D Installing Visual C++ 2010 Express

Follow these steps to install Visual C++ 2010 Express:

1. Point your favorite browser to http://go.microsoft.com/?linkid=9709949. This should download the web installer for Microsoft Visual C++ 2010 Express. If the link does not work, manually download the software:

   (a) Navigate to http://www.microsoft.com/visualstudio/ (compare Figure 48). Click on *Download* at the top (not *Download Now*!). In the popup menu, choose "2012 Downloads."

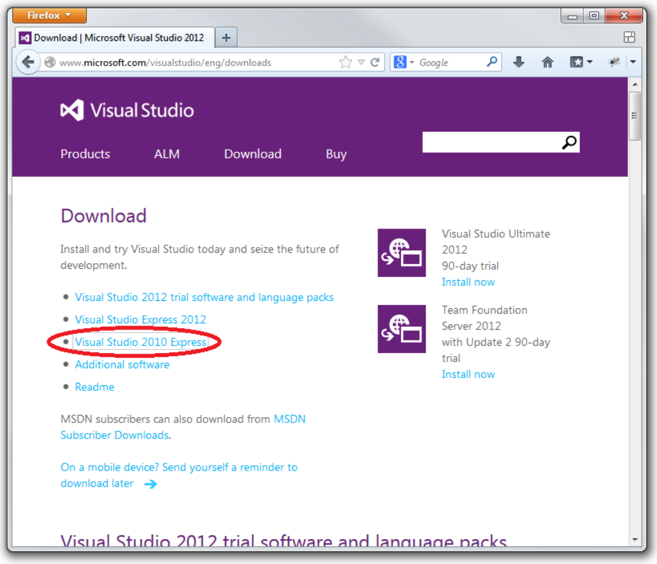

Firefox
Visual Studio 2012 Update 2 | Microsoft ...
www.microsoft.com/visualstudio/eng/visual-studio-update
Google
Visual Studio
Products
ALM
Download
Buy
Learn more
Download Now

Visual Studio 2012 Update 2

Includes enhanced functionality for agile planning, quality enablement, line-of-business application development, developer experience, and more.

Figure 48: Manually downloading Visual C++ 2010 Express (step 1).

   (b) On the page shown in Figure 49, click on *Visual Studio 2010 Express*.


Firefox
Download | Microsoft Visual Studio 2012
www.microsoft.com/visualstudio/eng/downloads
Google
Visual Studio
Products
ALM
Download
Buy

Download

Install and try Visual Studio today and seize the future of development.

- Visual Studio 2012 trial software and language packs
- Visual Studio Express 2012
- Visual Studio 2010 Express
- Additional software
- Readme

MSDN subscribers can also download from MSDN Subscriber Downloads.

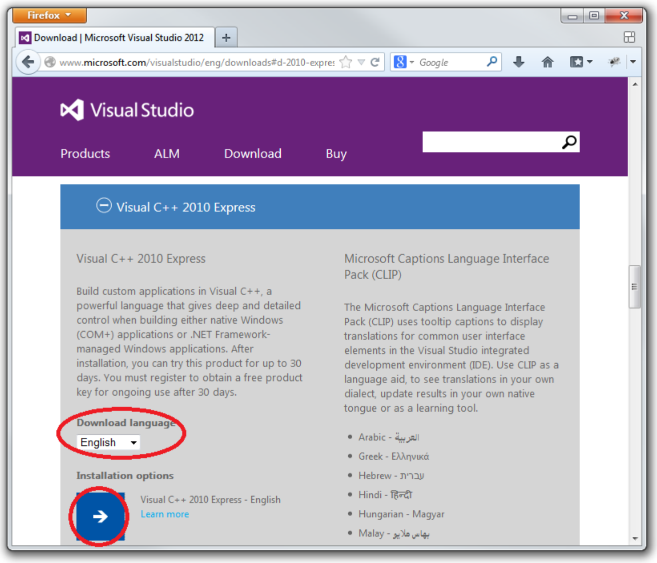
On a mobile device? Send yourself a reminder to download later →

Visual Studio Ultimate 2012 90-day trial
Install now

Team Foundation Server 2012 with Update 2 90-day trial
Install now

Visual Studio 2012 trial software and language packs

Figure 49: Manually downloading Visual C++ 2010 Express (step 2).


Firefox
Download | Microsoft Visual Studio 2012
www.microsoft.com/visualstudio/eng/downloads#d-2010-expre
Google
Visual Studio
Products
ALM
Download
Buy
English
Learn more

Visual C++ 2010 Express

Visual C++ 2010 Express

Build custom applications in Visual C++, a powerful language that gives deep and detailed control when building either native Windows (COM+) applications or .NET Framework-managed Windows applications. After installation, you can try this product for up to 30 days. You must register to obtain a free product key for ongoing use after 30 days.

Download language

Installation options

Visual C++ 2010 Express - English

Microsoft Captions Language Interface Pack (CLIP)

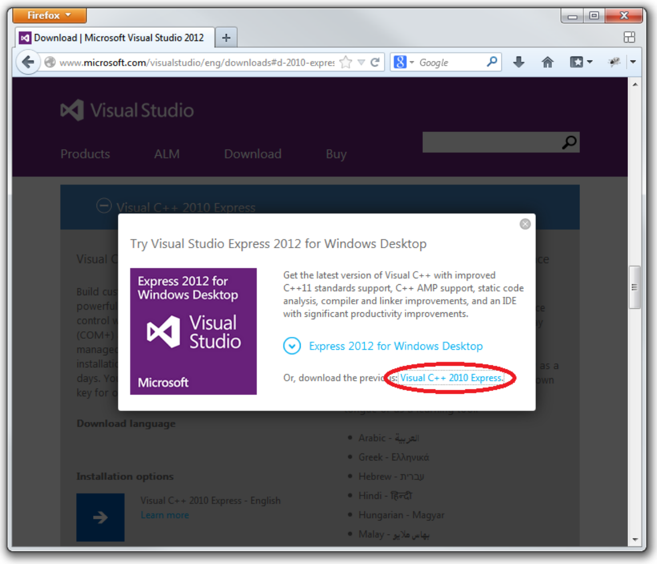
The Microsoft Captions Language Interface Pack (CLIP) uses tooltip captions to display translations for common user interface elements in the Visual Studio integrated development environment (IDE). Use CLIP as a language aid, to see translations in your own dialect, update results in your own native tongue or as a learning tool.

- Arabic - العربية
- Greek - Ελληνικά
- Hebrew - עברית
- Hindi - हिन्दी
- Hungarian - Magyar
- Malay - بهاس ملايو

Figure 50: Manually downloading Visual C++ 2010 Express (step 3).

(c) Open the *Visual C++ 2010 Express* tab, choose your language and click on the arrow (compare Figure 50).

(d) If you are asked whether you want to try Visual Studio Express 2012 for Windows Desktop instead, choose *Visual C++ 2010 Express* (compare Figure 51).


Firefox
Download | Microsoft Visual Studio 2012
www.microsoft.com/visualstudio/eng/downloads#d-2010-expre
Google
Visual Studio
Products
ALM
Download
Buy
Express 2012 for Windows Desktop
Visual Studio
Microsoft

Try Visual Studio Express 2012 for Windows Desktop

Get the latest version of Visual C++ with improved C++11 standards support, C++ AMP support, static code analysis, compiler and linker improvements, and an IDE with significant productivity improvements.

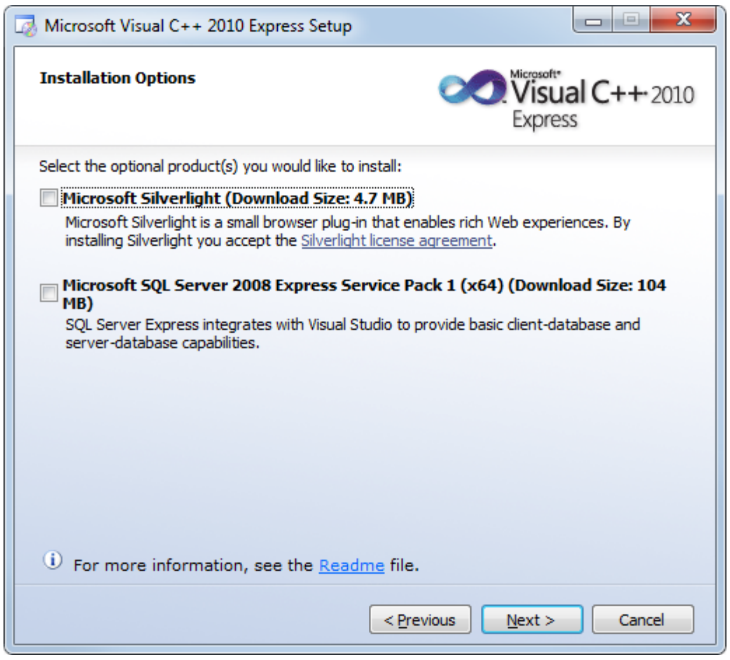
Express 2012 for Windows Desktop

Or, download the previous: Visual C++ 2010 Express.

Figure 51: Manually downloading Visual C++ 2010 Express (step 4).

2. After downloading `vc_web.exe`, launch it and follow the instructions on the screen. On the *Installation Options* page, you may deselect *Silverlight*, *SQL Server 2008* and any related service packs; they are not needed for CHROMOSOME (Figure 52).

3. After a few minutes, *Visual C++* setup will report that the installation has finished. In some cases, a reboot may be required to use *Visual C++*.


Microsoft Visual C++ 2010 Express Setup

Installation Options

Select the optional product(s) you would like to install:

☐ **Microsoft Silverlight (Download Size: 4.7 MB)**
Microsoft Silverlight is a small browser plug-in that enables rich Web experiences. By installing Silverlight you accept the Silverlight license agreement.

☐ **Microsoft SQL Server 2008 Express Service Pack 1 (x64) (Download Size: 104 MB)**
SQL Server Express integrates with Visual Studio to provide basic client-database and server-database capabilities.

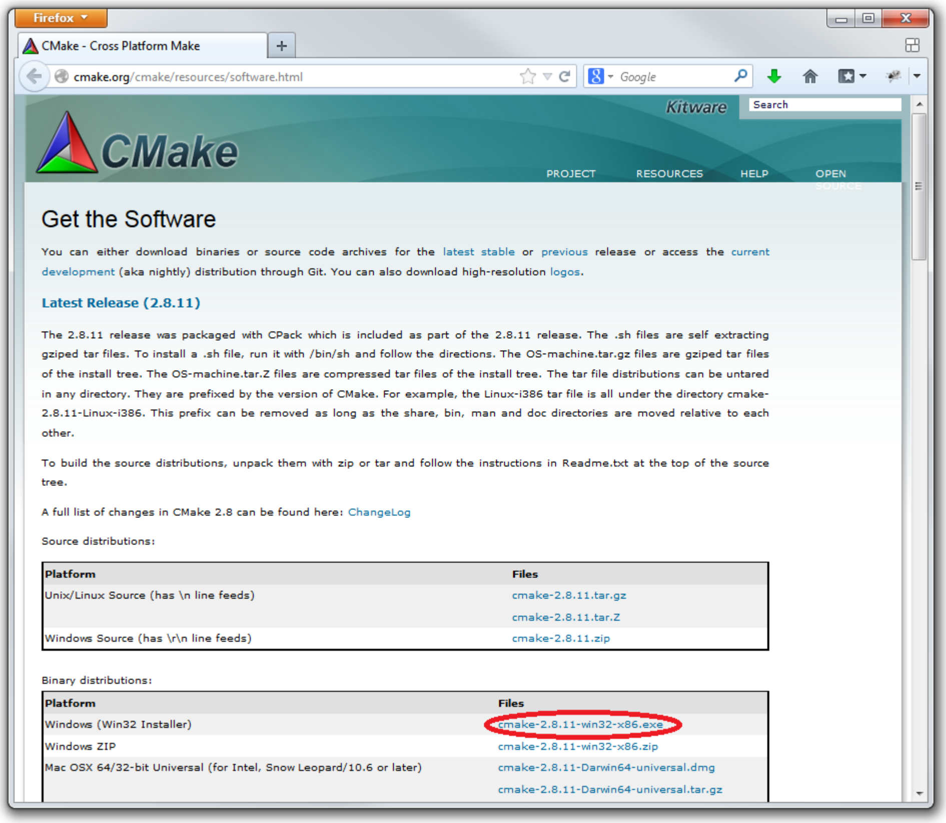
For more information, see the Readme file.

Figure 52: Visual C++ 2010 Express installation options.

E Installing CMake on Windows

Follow these steps to install CMake on Windows (on Linux, your distribution should include a package named `cmake`; for more information see Section 2.1).

1. Point your favorite browser to `http://cmake.org/cmake/resources/software.html` and click on the link corresponding to the Windows installer (compare Figure 53).


Firefox
CMake - Cross Platform Make
cmake.org/cmake/resources/software.html
Google
Kitware
Search
CMake
PROJECT
RESOURCES
HELP
OPEN SOURCE

Get the Software

You can either download binaries or source code archives for the latest stable or previous release or access the current development (aka nightly) distribution through Git. You can also download high-resolution logos.

Latest Release (2.8.11)

The 2.8.11 release was packaged with CPack which is included as part of the 2.8.11 release. The .sh files are self extracting gziped tar files. To install a .sh file, run it with /bin/sh and follow the directions. The OS-machine.tar.gz files are gziped tar files of the install tree. The OS-machine.tar.Z files are compressed tar files of the install tree. The tar file distributions can be untared in any directory. They are prefixed by the version of CMake. For example, the Linux-i386 tar file is all under the directory cmake-2.8.11-Linux-i386. This prefix can be removed as long as the share, bin, man and doc directories are moved relative to each other.

To build the source distributions, unpack them with zip or tar and follow the instructions in Readme.txt at the top of the source tree.

A full list of changes in CMake 2.8 can be found here: ChangeLog

Source distributions:

Platform	Files
Unix/Linux Source (has \n line feeds)	cmake-2.8.11.tar.gz
	cmake-2.8.11.tar.Z
Windows Source (has \r\n line feeds)	cmake-2.8.11.zip

Binary distributions:

Platform	Files
Windows (Win32 Installer)	cmake-2.8.11-win32-x86.exe
Windows ZIP	cmake-2.8.11-win32-x86.zip
Mac OSX 64/32-bit Universal (for Intel, Snow Leopard/10.6 or later)	cmake-2.8.11-Darwin64-universal.dmg
	cmake-2.8.11-Darwin64-universal.tar.gz

Figure 53: Selecting CMake for download, highlighted in red the download link.

2. After downloading the setup, launch it. Follow the instructions on the screen. When you get prompted whether to add CMake to the system PATH, you may choose to *not* add it. CHROMOSOME does not require CMake to be on the system search path.

F Installing the Modeling Tool

F.1 Installation

The CHROMOSOME Modeling Tool (XMT) is provided as an Eclipse plugin. fortiss provides an Eclipse update-site at `http://download.fortiss.org/public/xme/xmt/update-site/` from where you can install it. This section will first describe how to install Eclipse and then describe how to install the XMT plugin from the update-site.[41] Let us start with the installation of Eclipse:

1. The XMT requires Eclipse Indigo. More recent versions are not supported and may not be fully compatible. To download the correct version of Eclipse, point your favorite browser to the following web page:
   `http://www.eclipse.org/downloads/packages/eclipse-modeling-tools/indigosr2`
   and click on the link corresponding to your operating system to download Eclipse. The linked version is the Eclipse Modeling Tools package and already contains several of the plugins required by the XMT.

2. After downloading you simply extract the Eclipse archive into a directory of your choice.

3. If you do not already have a Java Runtime Environment installed, then go to the following web page and do this now (install the product named *Java SE Development Kit* found in the *Java Platform (JDK)* category):
   `http://www.oracle.com/technetwork/java/javase/downloads/`
   Notice that you need at least Java version 1.7!
   Also make sure that the versions of Java and Eclipse match (32-bit/64-bit).

4. Go into the Eclipse directory and start the Eclipse executable. When asked for a workspace directory, either accept the proposed location or enter a directory of your choice. Eclipse will store your settings and projects in that directory.

Now we can install the XMT plugin into Eclipse.

1. In Eclipse select *Help* → *Install New Software...*. This will show the install dialog.

2. Click on the `Add...` button and enter as location:
   `http://download.fortiss.org/public/xme/xmt/update-site/`
   and press `OK`. Loading may take some time.

3. You will see a long list of categories. Select the category 'CHROMOSOME Modeling Tool (XMT)'.[42]

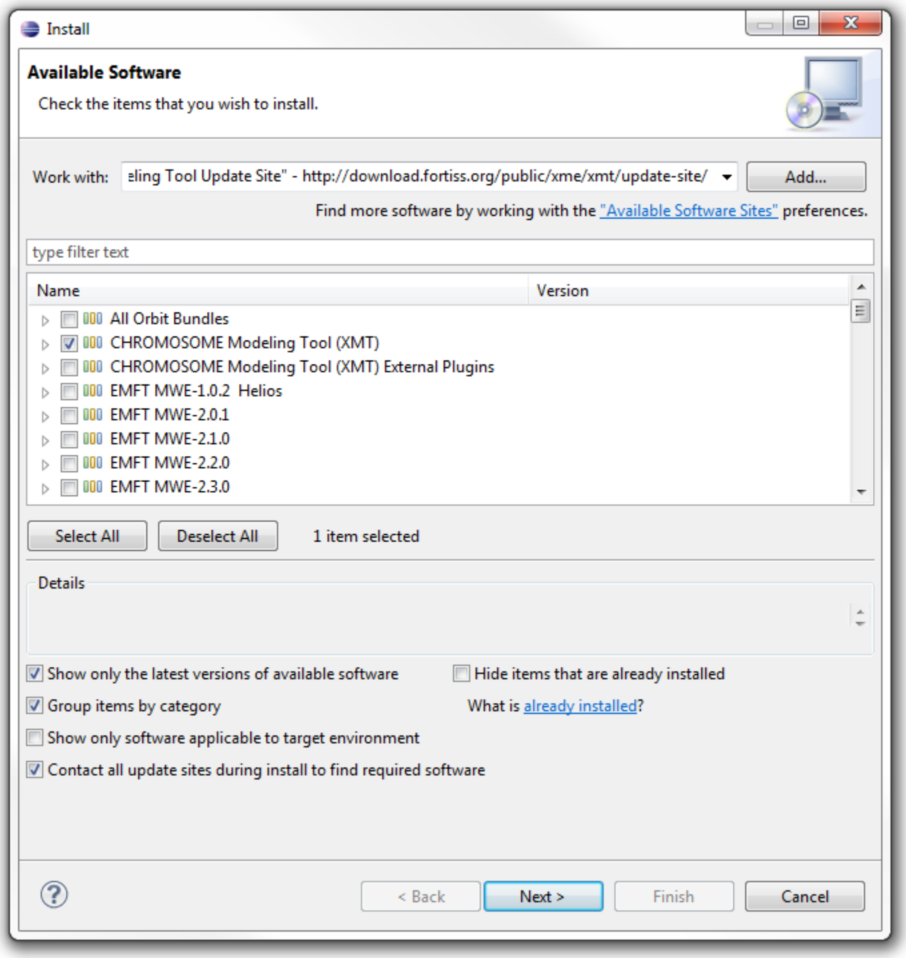
4. Make sure that you have the same options selected as in Figure 54.

5. Press `Next` to continue. Eclipse will show that it is about to install the XMT plugin. There shouldn't be any errors.

6. Press `Next` again. The licenses of the XMT plugin will be displayed. If you want to continue you must accept it and press `Finish`. This will start the installation.

7. During installation Eclipse will warn about unsigned content. Press `OK` to continue.

8. After the installation you will be prompted to restart Eclipse. Do this and you are done.



Work with: eling Tool Update Site" - http://download.fortiss.org/public/xme/xmt/update-site/

Find more software by working with the "Available Software Sites" preferences.

type filter text

Name	Version
☐ All Orbit Bundles	
☑ CHROMOSOME Modeling Tool (XMT)	
☐ CHROMOSOME Modeling Tool (XMT) External Plugins	
☐ EMFT MWE-1.0.2 Helios	
☐ EMFT MWE-2.0.1	
☐ EMFT MWE-2.1.0	
☐ EMFT MWE-2.2.0	
☐ EMFT MWE-2.3.0	

1 item selected

☑ Show only the latest versions of available software
☐ Hide items that are already installed
☑ Group items by category
What is already installed?
☐ Show only software applicable to target environment
☑ Contact all update sites during install to find required software

Figure 54: Install dialog of XMT update-site.

To verify if the tool has been installed correctly go to *Window* → *Open Perspective* → *Other...*. You should see an entry 'XMT'.

F.2 Troubleshooting

- The XMT perspective is not visible after installation of the XMT plugin.
  - Check if you have at least Java Runtime 1.7 installed. If you have several java versions installed, then also check if eclipse is run with the correct one. To force eclipse to use a certain version you can add the following two lines to the `eclipse.ini`[43] file:

    ```
    -vm
    C:/path/to/java/bin/javaw.exe
    ```

    Replace the second line with your actual path to the java executable. Note that you *must* enter it in two separate lines like above. You must also put it before the `-vmargs` option.

[41] When you visit he update-site with your browser a web page will appear with a link to the Eclipse help site for installing from update-sites.

[42] All other plugins that are needed by the tool are included in this update-site and will be installed automatically.

[43] You find this file directly in your eclipse installation directory.

G CHROMOSOME License

The following licensing conditions apply to both the CHROMOSOME runtime system (XME) and the CHROMOSOME Modeling Tool modeling tool (XMT).

Licensor for the purpose of discussing and improving the Work, but excluding communication that is conspicuously marked or otherwise designated in writing by the copyright owner as "Not a Contribution."

"Contributor" shall mean Licensor and any individual or Legal Entity on behalf of whom a Contribution has been received by Licensor and subsequently incorporated within the Work.

2. Grant of Copyright License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable copyright license to reproduce, prepare Derivative Works of, publicly display, publicly perform, sublicense, and distribute the Work and such Derivative Works in Source or Object form.

3. Grant of Patent License. Subject to the terms and conditions of this License, each Contributor hereby grants to You a perpetual, worldwide, non-exclusive, no-charge, royalty-free, irrevocable (except as stated in this section) patent license to make, have made, use, offer to sell, sell, import, and otherwise transfer the Work, where such license applies only to those patent claims licensable by such Contributor that are necessarily infringed by their Contribution(s) alone or by combination of their Contribution(s) with the Work to which such Contribution(s) was submitted. If You institute patent litigation against any entity (including a cross-claim or counterclaim in a lawsuit) alleging that the Work or a Contribution incorporated within the Work constitutes direct or contributory patent infringement, then any patent licenses granted to You under this License for that Work shall terminate as of the date such litigation is filed.

4. Redistribution. You may reproduce and distribute copies of the Work or Derivative Works thereof in any medium, with or without modifications, and in Source or Object form, provided that You meet the following conditions:

   (a) You must give any other recipients of the Work or Derivative Works a copy of this License; and

   (b) You must cause any modified files to carry prominent notices stating that You changed the files; and

   (c) You must retain, in the Source form of any Derivative Works that You distribute, all copyright, patent, trademark, and attribution notices from the Source form of the Work, excluding those notices that do not pertain to any part of the Derivative Works; and

   (d) If the Work includes a "NOTICE" text file as part of its distribution, then any Derivative Works that You distribute must include a readable copy of the attribution notices contained within such NOTICE file, excluding those notices that do not pertain to any part of the Derivative Works, in at least one of the following places: within a NOTICE text file distributed as part of the Derivative Works; within the Source form or documentation, if provided along with the Derivative Works; or, within a display generated by the Derivative Works, if and wherever such third-party notices normally appear. The contents of the NOTICE file are for informational purposes only and do not modify the License. You may add Your own attribution notices within Derivative Works that You distribute, alongside or as an addendum to the NOTICE text from the Work, provided that such additional attribution notices cannot be construed