COSYLAB
CONTROL SYSTEM LABORATORY

Control Sheet

Cosylab's newsletter

VOLUME 10

21/2011

ISSN: 1855-9255

Timing System complexity

BECAUSE OF THE FPGA/PS MYSTICISM OR THE LACK OF SYSTEMATIC APPROACH?

BY: JOZE DEDIC (COSYLAB)

Timing Systems (TSs) are big deal – at least for machines where many consecutive, precisely timed, and carefully thought-out actions are required to follow and direct the beam from the beginning to the end. Let's explore the big picture of TSs and point out important aspects.

When building a new machine, TS is often addressed with "we need an FPGA expert… & fiber-link expertise". Surely, it is technically challenging - it has to be done in hard real time; if fairly slow, in us, otherwise we're talking ps range. But also, the **timing system is at the heart of the accelerator. It dictates and coordinates the operation of all devices in the whole accelerator chain and helps devices meet machine-physics and equipment requirements.** Having a very good overview of overall system interplay comes before FPGA expertise.

TS is a service to the Control System (CS) device integration. It provides triggering, real-time data distribution and clock synchronization to hardware devices (sensors, actuators and other beam related subsystems, directly responsible for their aspect of desired machine operation). Relying on TS services, CS becomes a bridge between high-level machine-control applications on one side and hardware on the other.

Being the core service provider, it is not hard to imagine that the **TS is tightly coupled with the whole machine**. To do a clean design of the TS we need to understand **complete machine interplay** – we need to:

a) assemble a complete list of TS clients (together with all expectations from TS services),

b) understand the interplay between them, according to the machine's physical needs (i.e. devices' coordinated behavior, as expected by the machine), and

c) define clean interfaces and TS-service functions/routines. When defining the latter, very good collaboration with other system development groups and a lot of iteration is required. For that, one requires a broad set of special skills, as "translation" from TS-clients (physics, electronics, software…) vocabulary of requirements is far from trivial.

You might have noticed I haven't said much about jitter, resolution, accuracy, event rate, delay propagation, etc… I consider this as **(A) core timing-system transfer-layer expertise**. Although this knowledge is readily available in many labs, its entire complexity is still often underestimated. I also haven't said much about **(B) electronic board development**; FPGA, board design, SFP, fiber communication, encoding, drivers, VME, cPCI… etc. Knowledge about this is also found in most labs and many COTS solutions exist… Of course, (B) is built on top of (A). What comes next is **(C) complete machine interplay** as the last ingredient. Seeing the **complexity of ingredient (A) and (B) often blurs the whole picture, causing the ingredient (C) to be neglected**. Looking from this perspective, I claim that TS are so complex that it's even hard to understand what makes them complex in the first place.

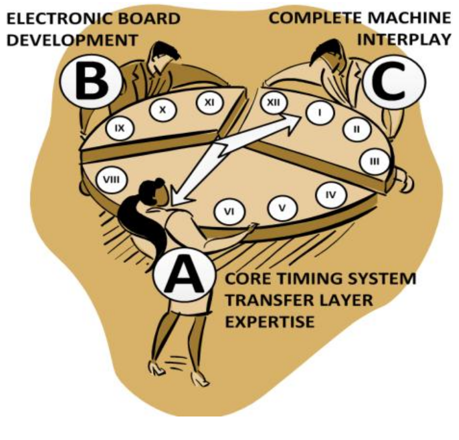
Conclusions

Understanding timing systems is much more than 'simply' being an expert in a certain communication technology. **It's all about understanding what services the rest of the machine requires from the timing system to successfully fulfill the mission of the complete machine.** What we see most of the time is that people playfully jump headlong into implementation (because FPGAs are cheap and everybody "knows how to do it"), but it later turns out that the system does not fit nicely with the rest of the machine. Once additional timing systems are introduced (and other ugly patches...), the system architecture has already gone far down the drain, but everybody sticks with it since they already invested so much effort… We've now been involved in quite a few timing projects and we see this pattern reoccurring.

Timing system transport layer—as only the ingredient (B), on top of (A)—has been addressed and solved already multiple times. What's left missing is the system integration layer (C) for your machine. **Reuse the knowledge of the timing system transport layer and focus the core of the work on the system integration layer**.

Even though for its demanding technical implementation the timing-system transfer layer is quite a big chunk to swallow, it is even bigger from the system point of view since the entire machine interplay has to be bound with technical expertise. **Do not fool yourself that work on the timing-system transfer layer can go without timing-system integration work.** I.e., until you see the whole big picture of interplay between all devices and services, don't allow FPGA coding.

In case you are intrigued by the title question, I would like to see your answer, comments… even further questions… sent to joze.dedic [at]cosylab.com


ELECTRONIC BOARD DEVELOPMENT
COMPLETE MACHINE INTERPLAY
B
C
A
CORE TIMING SYSTEM TRANSFER LAYER EXPERTISE
I
II
III
IV
V
VI
VIII
IX
X
XI
XII

A successful Timing System design is composed of and supported by three integral parts.

Packaging of Control System Software for ITER

BY: K.ZAGAR, A.ZAGAR, R.SABJAN, CO-BIK, SOLKAN, SLOVENIA; M.KOBAL, N.SAJE, COSYLAB, LJUBLJANA, SLOVENIA; F.DI MAIO, D.STEPANOV, ITER ORGANIZATION, ST.PAUL LEZ DURANCE, FRANCE

Abstract

Control system software consists of several parts – the core of the control system, drivers for integration of devices, configuration for user interfaces, alarm system, etc. Once the software is developed and configured, it must be installed to computers where it runs. Usually, it is installed on an operating system whose services it needs, and also in some cases dynamically links with the libraries it provides. Operating system can be quite complex itself – for example, a typical Linux distribution consists of several thousand packages. To manage this complexity, we have decided to rely on Red Hat Package Management system (RPM) to package control system software, and also ensure it is properly installed (i.e., that dependencies are also installed, and that scripts are run after installation if any additional actions need to be performed). As dozens of RPM packages need to be prepared, we are reducing the amount of effort and improving consistency between packages through a Maven-based infrastructure that assists in packaging (e.g., automated generation of RPM SPEC files, including automated identification of dependencies). So far, we have used it to package EPICS, Control System Studio (CSS) and several device drivers. We perform extensive testing on Red Hat Enterprise Linux 5.5, but we have also verified that packaging works on CentOS and Scientific Linux. In this article, we describe in greater detail the systematic system of packaging we are using, and its particular application for the ITER CODAC Core System.

Introduction

The principal challenges of today's control systems for large experimental physics facilities are complexity and quality assurance.

By this we mean the fact that a large number of software components – executing either on the same host or in a distributed set-up – need to be integrated into a functioning whole while performing according to performance and reliability expectations. The complexity challenge stems both from the inherently distributed, large-scale nature of a control system, as well as the trends in component-based software engineering and systems engineering, where monolithic systems are giving way for those that are integrated from smaller, more manageable subsystems. With limited development, maintenance and integration resources – in particular skilled staff – it is important that as many tasks as possible are automated, and that common problems have common solutions – i.e., that standardization takes place to the extent that it is economically feasible.

The ITER CODAC [1] control system is also facing these challenges. CODAC integrates software packages that are a product of two decades of work, and which have been developed in diverse environments by different teams. Not surprisingly, each of these packages takes a different approach on how the software is built, and what quality assurance process is in place during its release. We have decided to standardize at least the interface with which the developer (or maintainer) interacts with the build system. To achieve this, we have wrapped the diverse approaches and technologies for building (Makefile [2], Ant [3], shell scripts, Eclipse builder [4], etc.) into one tool. Since many software packages share the same approach (e.g., EPICS base [5] and all of its extensions rely on Makefile, while Control System Studio [6] and all of its plug-ins rely on Eclipse), we were looking for a way to re-use our effort: for example, specify integration with the EPICS Makefile system in a single place, and "invoke" it with a one-line stanza in all software packages where it is needed.

As we are not the first to have come across this challenge, a market survey revealed that build tool frameworks already exist (for example, Maven [7] and Gradle [8]). After our evaluation, performed in late 2009, we have settled to use Maven 2 as the platform, and we have chosen to solve our challenges by implementing a plug-in for this tool.

Another challenge is managing deployment across the many hosts that will eventually constitute the control system. However, this challenge is not uncommon in the IT industry, where large corporations also have thousands of computers that need to be managed with the limited IT staff. To leverage existing solutions, ITER had decided to take an off-the-shelf approach: using Red Hat Enterprise Linux and its automated installation and update capabilities enabled with the Red Hat Satellite software [9].

Managing installation, un-installation and updating of software packages on an individual host is a rather complex task in itself. The most trivial step of it is to place the files constituting a software package (executables, scripts, configuration files and data files) to the right places in the file system. As un-installation and update need to be able to clean-up those files, meta-data must be associated with each file to specify which software package had installed it. Installation/un-installation might require that some actions are taken (e.g., adaptation of configuration files of other software components, creation of database schemas, population of databases, etc.). And finally, software package might have dependencies, and other software might depend on: installing a package might thus have a precondition that other packages are installed beforehand, and uninstalling it may have a consequence that those depending on it should be uninstalled as well.

```
# Spec file for package rf-ich
-sample-MCioc
# Generated by the codac-
packager Maven plugin.
# Date: Fri Sep 30 13:32:55
CEST 2011 ...

%define unit_version_full %
{codac_version_full}.v1.0a1

Name:%{codac_rpm_prefix}-rf-ich
-sample-MCioc
Version:%
{codac_version_full}.v1.0a1
...

Requires:%get_current codac-core-3.0-
epics-autosave

%description
Input (a snippet from Maven pom.xml):

%install
sed -r -i 's#epicsEn-
vSet\("TOP"\s*\,\s*".*?"\)
#epicsEnvSet\("TOP","/opt/codac-3.0/apps/
rf-ich-sample"\)#g' %{buildroot}/opt/
codac-3.0/apps/rf-ich-sample/iocBoot/
iocMC-ICHCore/envPaths
...

install -d %{buildroot}/etc/opt/codac-
3.0/alt.d/
echo --slave \"%{_bindir}/MC-ICHCore-
ioc\" \"codac-sudo-MC-ICHCore-ioc\" \"/
opt/codac-3.0/bin/services/sudo-service\"
>> "%{buildroot}/etc/opt/codac-3.0/alt.d/
rf-ich-sample-MCioc"
echo --slave \"%{_initrddir}/MC-ICHCore-
ioc\" \"codac-srv-MC-ICHCore-ioc\" \"/
opt/codac-3.0/bin/services/MC-ICHCore-
ioc\" >> "%{buildroot}/etc/opt/co
```

Figure 1: Example of a SPEC file that provides meta-information and build instructions for an RPM package.

This problem, too, has already been solved by the IT community. On Linux platforms, the Debian package management (APT/DEB) and Red Hat Package management (YUM/RPM) are commonplace. As ITER has chosen Red Hat Enterprise Linux as the operating system, we have opted for the YUM/RPM technology.

To provide a RPM, the developer must provide a so-called SPEC file. The SPEC file contains (see Figure 1): meta-information about the package (name, version, description, etc.), instructions in form of an executable script on how to build the package (unpack the sources, run configure/make or other tools, etc.) which files to package, and what default permissions to assign to them the scripts to execute before and after installation, and before and after uninstallation.

Constituents of a Control System

In case of ITER CODAC, the control system consists of the following kinds of software packages: EPICS IOC applications. Configuration files for the BEAST alarm server. Configuration files for the BEAUTY archiving system. Kernel modules, e.g., for implementation of kernel-mode device drivers. User-mode device drivers and libraries.

The EPICS IOC applications are standard EPICS applications, built with the EPICS' Makefile system. They consist of the binary compiled for the target platform, the st.cmd start-up script, and EPICS database files.

These files are packaged in an RPM package, and an init.d script is automatically generated that allows for starting up and shutting down of the IOC as a system service. The init.d script also provides a console through which developers and maintainers can access the EPICS shell of the IOC process (via the screen tool).

For security reasons, it is not advisable to run services as the root user. Therefore, a system user called "codac" is provided, and all services run under that account. This raises some issues with permissions (e.g., the codac user by default doesn't have permissions to interact with kernel-mode device drivers, nor does it have permissions to set its real-time attributes such as scheduling priority and CPU affinity). The packaging ensures also that these permissions issues are properly addressed.

Packaging of configuration files for BEAST and BEAUTY involves putting the configuration files in the RPM package, and running the database import tools upon installation of the RPM to populate the BEAST and BEAUTY configuration databases with their content. In CODAC, the content of these configuration files is automatically generated by the SDD tools [1], thus they are in-sync with the contents of the EPICS configuration database.

Kernel modules are built with standard tools for kernel modules. Currently, kernel modules can be built for two targets: A regular kernel.

A real-time kernel. For each build, a separate RPM package is provided, which then has its dependency set as required (either to the kernel or to the kernel-rt package).

THE Maven Plugin

The packaging of control system's constituents described in the previous section into RPMs is performed by the a Maven plugin we have developed.

The Maven plugin is designed to execute the packaging task during the "package" phase of its lifecycle – i.e., after compilation and testing, but before installation and deployment.

The description of the packaging is very concise. Figure 2 shows an example of the Maven configuration that results in RPM SPEC file shown in Figure 1.

The RPMs thus produced allow installation of several versions of the ITER CODAC system simultaneously. E.g., version 2.1 and 3.0 can be installed at the same time, with the first residing in /opt/codac-2.1 and the latter in /opt/codac-3.0. The developer can then switch between the two versions with a command codac-version, which redirects all the softlinks from the system's paths accordingly (via the alternatives system) and reconfigures environment variables.

In addition, the Maven plugin provides command-line options that facilitate generation of Maven's project definition file, the pom.xml. Thus, even developers not familiar with syntax of this file can configure their projects to be packaged.

```
<package name="MCioc">
  <include name="MC-ICHCore" type="ioc"/>
  <include type="boy" file="*"/>
  <include type="databrowser" file="power.plt"/>
</package>
```

Figure 2: Excerpt from Maven's POM XML file responsible for generating RPM SPEC file from Figure 1.

The developer might use the Maven-based tools as follows. Firstly, the developer would create a *project* (a unit in ITER's CODAC terminology) by executing:

```
mvn iter:newunit -Dunit=my-unit
```

This results in a subdirectory *m-my-unit* (the m-prefix is prepended due to ITER CODAC's naming convention for units). The directory structure conformant to CODAC's standards, and Maven's *pom.xml* file, are also created by this command.

The following sequence of commands creates an EPICS application, and then configures an IOC process to run that application. The type of the application here is "psh", referring to ITER CODAC's *Plant System Host*:

```
cd m-my-unit
mvn iter:newapp \
   -Dapp=PlantSystemHost \
   -Dtype=psh
mvn iter:newioc \
   -Dioc=PlantSystemHost \
   -Dapp=PlantSystemHost \
   -Dtype=psh
```

Now, the RPM can already be prepared by executing:

```
mvn package
```

It is also possible to conveniently start the resulting IOC:

```
mvn iter:run
```

Conclusion

The Maven-based tools that have been developed to facilitate the development process greatly simplify development of software for the control system – from input/output controller processes to extensions of the operator's graphical user interfaces. The interface provided to developers is simplified so that even those who had no prior exposure to EPICS, Maven or RPM are able to create new EPICS-based, CODAC-compliant projects, build them, and ensure that results are packaged in an installable RPM. While the tools have been developed for ITER, they can be adapted to other projects as well. For example, currently an effort to adapt ITER CODAC for the needs of the European Spallation Source's control system.

References

[1] F. Di Maio et al., "The CODAC Software Distribution for the ITER Plant Systems", ICALEPCS'11, Grenoble, France.

[2] GNU Make; http://www.gnu.org/s/make/.

[3] Apache Ant; http://ant.apache.org/.

[4] Eclipse; http://www.eclipse.org/.

[5] Experimental Physics and Industrial Control System; http://www.aps.anl.gov/epics/.

[6] Control System Studio; http://cs-studio.sourceforge.net/.

[7] Apache Maven; http://maven.apache.org/.

[8] Gradle; http://www.gradle.org/.

[9] Red Hat: Red Hat Network Satellite; http://www.redhat.com/red_hat_network/.

---

* The Centre of Excellence for Biosensors, Instrumentation and Process Control - COBIK, Velika pot 22, SI-5250 Solkan, Slovenia COBIK is an operation financed by the European Union, European Regional Development Fund and Republic of Slovenia, Ministry of Higher Education, Science and Technology."

** This article will also be published at the JACoW site.

VHDL fashion police

A JAVA-BASED VHDL SOURCE CODE VERIFIER

BY: ROK TAVČAR

Here at Cosylab, we take pride in our VHDL. Along with great architectural design, we always strive to assure high quality code in terms of readability, uniformity and clarity.

Having an ever growing force of VHDL developers, we need systematic code review and a good set of coding style rules.

None of that is new. But we've noticed that a big part of the code review effort is spent on finding boring mistakes like inaccordance with naming conventions, indent style or »VHDL programming NO-NOs« like assigning values to signals upon declaration (this is not C!).

The developer has a blind spot for own mistakes, rendering him/her useless for their removal. The reviewer is in turn annoyed by repetitively fixing simple errors which steal attention from real issues in the code, e.g. neglecting to oversample a signal which crosses clock domains. That's just not fair to anyone, the least of all to the code.

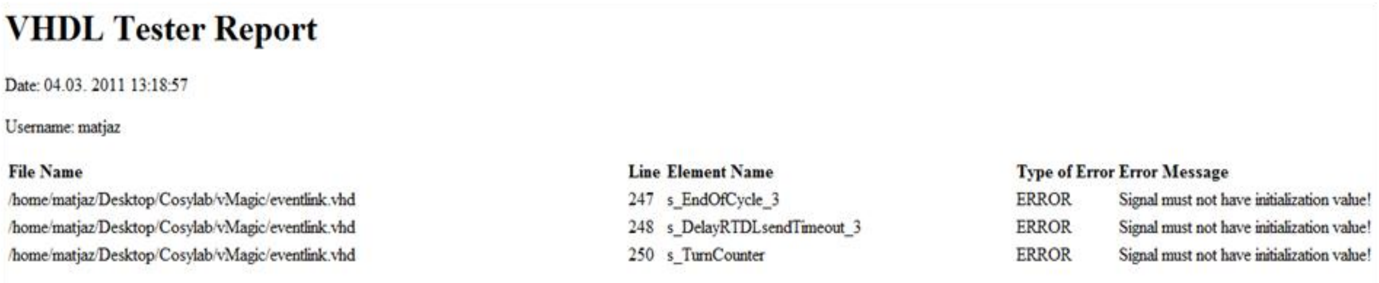
This calls for automatization! We refrained from commercial tools as we found them expensive and awkward to use. We also want total freedom with rule definitions. As our set of conventions is custom to Cosylab, we decided to develop a software tool of our own. The main requirements were that the tool is easy to use and adding a new rule is a simple programming task. It should be able to »spider-through« a folder tree to find all VHDL files, check them against defined rules and generate a report.

The Cosylab VHDL code verifier was developed as part of a Cosylab Java Academy project. It is inteded for periodic use by the developer to keep the code clean at all times. Upon code review, the reviewer doesn't touch the code if the automatic report isn't void of errors. With the tool employed, the reviewer can focus on conceptual errors and only produces real feed for a learning

experience!

It's also great to know that if you need to look at someone else's code, it will look like your own. That makes us feel more like a family.

* The Centre of Excellence for Biosensors, Instrumentation and Process Control - COBIK, Velika pot 22, SI-5250 Solkan, Slovenia COBIK is an operation financed by the European Union, European Regional Development Fund and Republic of Slovenia, Ministry of Higher Education, Science and Technology."

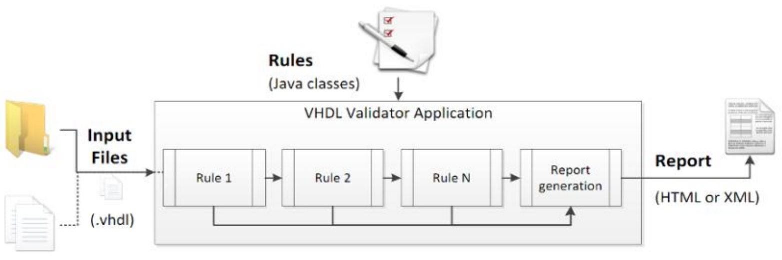


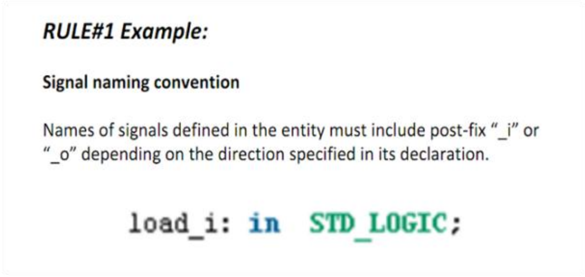
VHDL Tester Report

Date: 04.03. 2011 13:18:57

Username: matjaz

File Name	Line	Element Name	Type of Error	Error Message
/home/matjaz/Desktop/Cosylab/vMagic/eventlink.vhd	247	s_EndOfCycle_3	ERROR	Signal must not have initialization value!
/home/matjaz/Desktop/Cosylab/vMagic/eventlink.vhd	248	s_DelayRTDLsendTimeout_3	ERROR	Signal must not have initialization value!
/home/matjaz/Desktop/Cosylab/vMagic/eventlink.vhd	250	s_TurnCounter	ERROR	Signal must not have initialization value!





RULE#1 Example:

Signal naming convention

Names of signals defined in the entity must include post-fix "_i" or "_o" depending on the direction specified in its declaration.

```
load_i: in  STD_LOGIC;
```

Figure 1: Program flow

Figure 2: Example report

Reminder!

Visit us on our **freshly renewed** web page: www.cosylab.com

COSYLAB d.d.

Teslova ulica 30

SI-1000 Ljubljana

Slovenia

Phone: +38614776676

Fax: +38614776610

E-mail: controlsheet@cosylab.com

Web: www.cosylab.com

Visit us on : http://cosylab.com/resources/our_newsletter_control_sheet/

The power of Cosylab...

Cosylab has its own secret agent!

A secret photo of the notice board of the PSI control group, taken by our secret agent in february this year. We're proud that they considered our control sheet article relevant enough to put in on the board. This gives us motivation to continue.

Cosylab T-Shirts, priceless. For young and old(er)!

Even though you can't really see it very clearly—Mark Boland's cute baby boy Santiago is wearing a lovely Cosylab jumpsuit! The expression of Mark's baby shows that there might be a great potential there. There definitely are some very inspirational thoughts in that cuties head, who is a year older since this picture was taken.
Of course we cannot miss the Cosylab t-shirt on the beach. Since its winter, it just might send us into a daydream underneath the palmtrees and cosy beach life near the ocean.

Congratulations Mark!

Mark Boland and his baby Santiago - proud owners of Cosylab jumpsuit and Cosylab T-Shirt

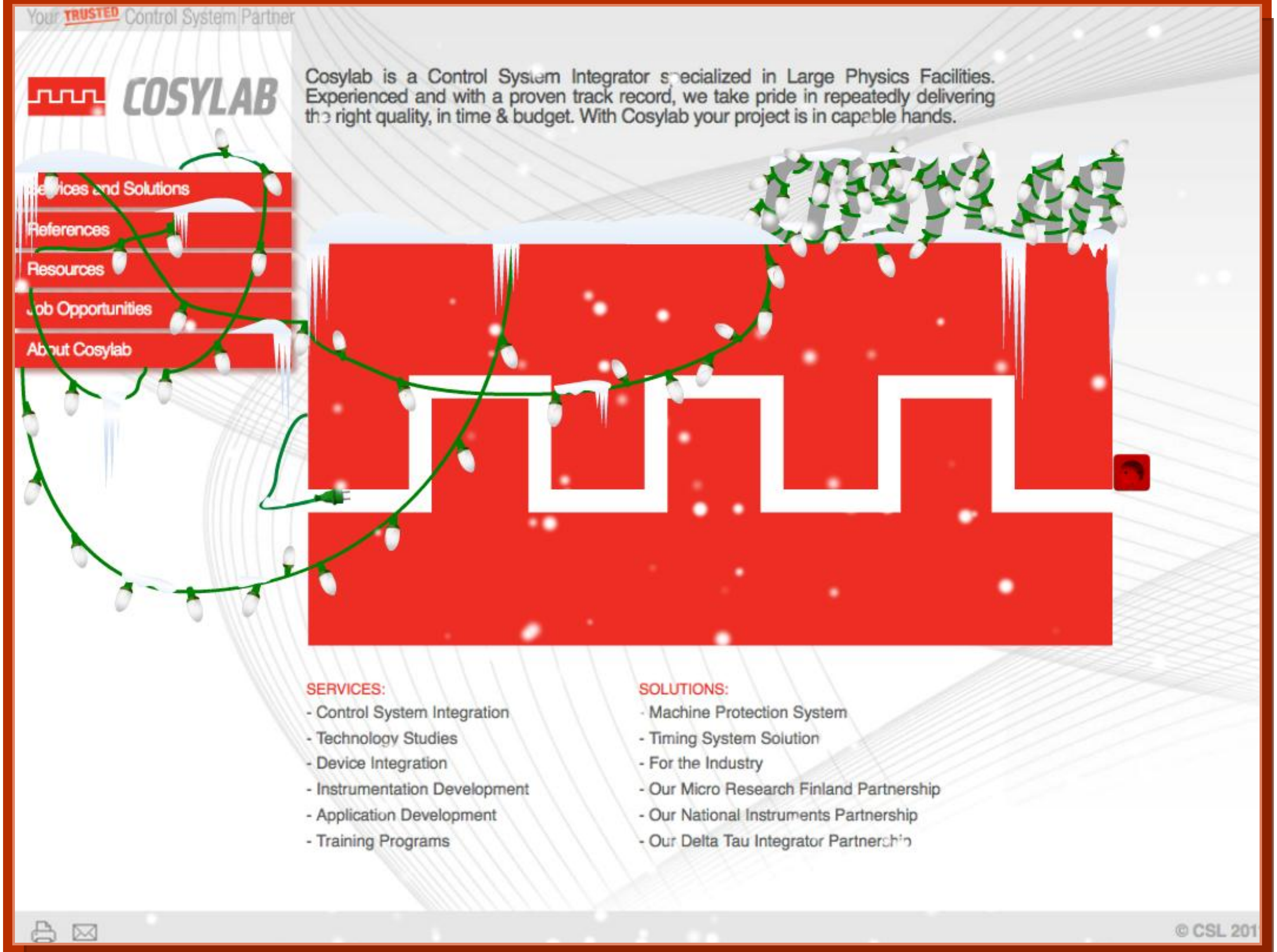
Happy and Cosy New Year 2012!

Dear all,

may prosperity and success be the two sides of the coin for this year and for all the years ahead. By clicking on the picture below, we wish for a happy and joyful year, as fun as our surprise new year's card's intention.

Best wishes and happy 2012,

Your Cosylab team



Your TRUSTED Control System Partner

COSYLAB

Cosylab is a Control System Integrator specialized in Large Physics Facilities. Experienced and with a proven track record, we take pride in repeatedly delivering the right quality, in time & budget. With Cosylab your project is in capable hands.

SERVICES:
- Control System Integration
- Technology Studies
- Device Integration
- Instrumentation Development
- Application Development
- Training Programs

SOLUTIONS:
- Machine Protection System
- Timing System Solution
- For the Industry
- Our Micro Research Finland Partnership
- Our National Instruments Partnership
- Our Delta Tau Integrator Partnership

© CSL 201

Visit us on : http://www.cosylab.com/ny2012/