Communicating Process Architectures 2016
K. Chalmers, J.B. Pedersen et al. (Eds.)
Open Channel Publishing Ltd., 2016

© 2016 The authors and Open Channel Publishing Ltd. All rights reserved.

Simulation and Visualization Tool Design for Robot Software

Zhou LU [1], Tjalling RAN and Jan F. BROENINK

Robotics and Mechatronics, CTIT Institute, Faculty EEMCS,
University of Twente, The Netherlands

Abstract. Modern embedded systems are designed for multiple and increasingly demanding tasks. Complex concurrent software is required by multi-task automated service robotics for implementing their challenging (control) algorithms. TERRA is a Communicating Sequential Processes (CSP) algebra-based Eclipse graphical modelling tool suite which is capable of C++ code generation. It is designed to ease tedious and error-prone concurrent software development for robotics. However, sufficient simulation and visualization supports are not provided yet in TERRA. A hybrid simulation approach is proposed in this paper to provide simulation capabilities for the TERRA tool suite with respect to the Cyber-Physical Systems (CPS) co-design. Moreover, a visualization for the simulation is designed as well to provide animation facilities which enable users to visually trace simulated execution flows. Finally, we use an example to test the hybrid simulation approach as well as visualization facilities. The simulation approach is shown to be sufficient and the visualization works as intended.

Keywords. simulation, visualization, CSP algebra, animation, CPS, Eclipse

Introduction

Context

Modern embedded systems that are widely used in automated service robotics, are becoming increasingly complex as they are designed to execute multiple and increasingly demanding tasks. Additionally, service robotics are always required to interact with the environment seamlessly, which makes the design even more difficult. This combination of several research and engineering fields is quite challenging for professionals who only focus on their own domains. Following this trend, the term of the Cyber-Physical Systems (CPS) was proposed [1,2,3], which considers the cyberspace and the physical world to be more closely integrated compared to traditional design of embedded systems.

In CPS, *Cyber* represents the information-based computation and network elements, including computing processes, logical communicating processes, and discrete feedback control processes. *Physical* represents the processes, objects, and events in natural or man-made physical systems, operating according to laws of physics in continuous time. Physical processes are combinations of many events occurring at the same time, so they are concurrent by nature. Controlling the dynamics of such processes is one of the main tasks in CPS design. Consequently, concurrency is intrinsic to CPS. To put it another way with respect to the cyberspace, complex concurrent software design is required by multi-tasking automated ser-

[1]Corresponding Author: *Zhou Lu, Robotics and Mechatronics, University of Twente, P.O. Box 217, 7500 AE Enschede, The Netherlands*. Tel.: +31534894419; E-mail: `z.lu@utwente.nl`.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

vice robotics for their challenging control algorithms, unlike traditional embedded systems in which software processes are rooted in sequential steps. Many of the technical challenges in designing and analysing embedded control software come from the need to bridge an inherently sequential semantics with an intrinsically concurrent physical world [4]. The Communicating Sequential Processes (CSP) algebra is a formal language for describing patterns of interaction in concurrent systems [5,6], which is a potential solution for formalization and concurrency challenges in CPS.

Related Work: Model-Driven Design for CPS

Over the previous decade, merging the control, systems, and software engineering built on the principles of Model-Driven Design (MDD), has become one of the key research priorities [7]. Working with models has several advantages. Quality and consistency demands of models can be rigorously checked using formal verification tools [8]. Also, modelled systems can be tested and simulated off-line [9], enabling developers to follow the logic of their application, to check assumptions about its environment, and to gain confidence in end-to-end behaviour [10]. Such activities make system validation straightforward, such that the real implementation can realistically be right-first-time.

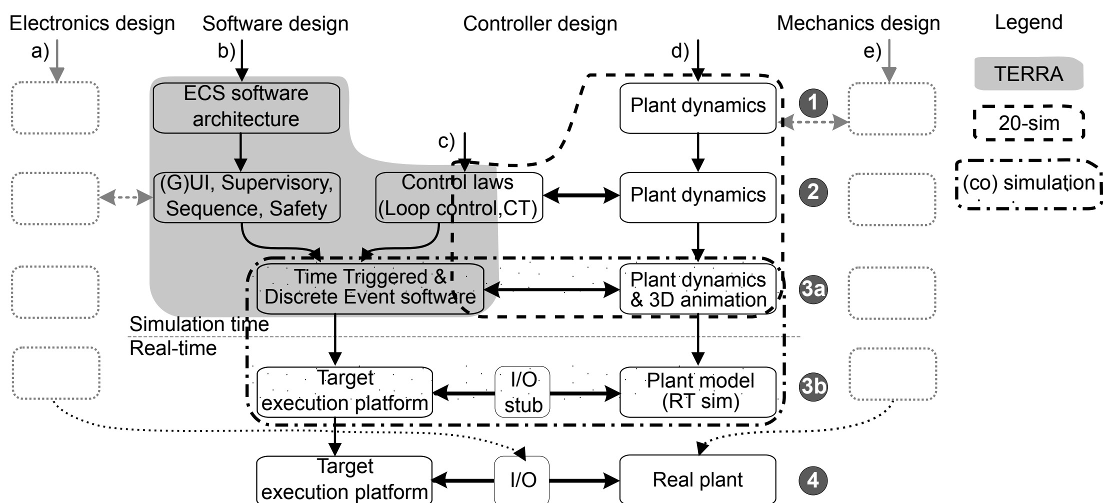


Figure 1. Co-design methodology for CPS using MDD. Modified from (Broenink et al. [11]).

A complete model of a CPS represents the coupling of its environment, physical processes, and embedded computations [10]. A generic concurrent co-design methodology for CPS, proposed by Broenink et al. [11,12] is shown in Figure 1. It explains a way of working, being used in this paper as well, which extensively uses MDD techniques for CPS co-design, whereby the application domain is robotics and mechatronics. TERRA [13,14] is a CSP algebra-based graphical modelling tool suite, which consists of several sets of Eclipse plugins and aims to ease tedious and error-prone concurrent software development for robotics. It supports the design methodology and covers certain scope with respect to MDD for the cyberspace, which includes embedded control software (ECS) architecture modelling, machine-readable CSP [15] transformation and C++ code generation. 20-sim [16] is another graphical modelling tool involved in the design methodology, with respect to MDD for the physical world. It is capable of modelling plant dynamics using bond graphs as well as equation-based control models [17]. Moreover, 20-sim can generate XML files and C++ source code that represent the control model contents and control laws simultaneously, which can be integrated by TERRA models as functional components, e.g. for loop controllers.

Problem Statements and Motivation

The way of working mentioned above asks for precise modelling of *both* the cyber and physical parts. Iterative and incremental design and development is one of the most important features in MDD [18,19], in which sufficient verification and/or validation of models are required from the very early design phase as well as in each design and development iteration.

Using simulation in MDD for CPS, implies that one or more domain models involved, namely Discrete-Event and Continuous-Time domains. Combined simulations of multiple domain models is called *co-simulation*. If different domain models can be co-simulated together efficiently, relevant co-enhancement to certain models can be made through co-simulation results. Consequently, the reliability of software and the confidence in the design will both be increased.

Developing, modifying and integrating models that cover *all* CPS design disciplines is one of the major challenges [20]. Incrementally co-modelling and co-simulation are crucial in CPS co-design. In our way of working, the TERRA tool suite can deal with the MDD for the cyberspace while 20-sim can model plant dynamics in the physical world. However, currently facilities in neither of them is sufficient to satisfy requirements pertaining to CPS co-design. To be more specific, although TERRA and 20-sim have already covered certain areas of cyber-physical modelling, as shown in Figure 1, TERRA does not provide enough simulation facilities to integrate modelled plant dynamics in 20-sim into the design loop. Meanwhile, certain visualization techniques are not sufficient either to fully assess models.

Outline

We propose an MDD approach and implemented facilities for the TERRA suite to simulate the process execution flow, which can be used to gain insight into models. This is summarized in Section 1. However, that approach is not able to simulate functional components that contain certain algorithms which are implemented in C++ code. The scope of this paper focuses on a new approach which is capable of simulating and visualizing the process execution flow whereby functional results of certain algorithms can be obtained simultaneously as well.

In Section 2, design space exploration for obtaining executable models is discussed first. Then a hybrid simulation approach for our way of working is proposed. In Section 3, we introduce a design for visualizing simulation results, followed by results analysis and discussion in Section 4. Conclusions and recommendations are summarized in Section 5.

1. Previous Work

The TERRA tool suite uses an explicit CSP meta-model [13] as its fundamental basis, from which MDD techniques can profit to generate corresponding machine-readable CSP code of a TERRA model. Consequently, required behaviour of the modelled software architecture can be formally specified and verified during the early design phases using the Failures Divergences Refinement (FDR) tool [21]. In the automated service robotics domain, lots of CPS are safety-critical. Their required behaviour goes beyond fundamental properties (e.g. freedom from deadlock and livelock) to include liveness properties (e.g. that the system will react in a certain way, given a certain set of signals received and system state). Many of these can be formally specified and verified [22] but, depending on the complexity, many cannot because of state-space explosion. For the latter, an MDD approach with simulation to follow and observe process execution flow was proposed in [7]. A simulation meta-model was designed to abstract the execution procedure of processes. It consists of hierarchical abstraction levels to represent semantics in TERRA models and is able of generating a simulated execution trace that represents the process execution flow. In addition, rules were defined for

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

model-to-model (M2M) transformation, where the source model and the target model conforms to the CSP meta-model and the TERRA simulation meta-model respectively. Figure 2 is a TERRA example and its simulated execution trace.

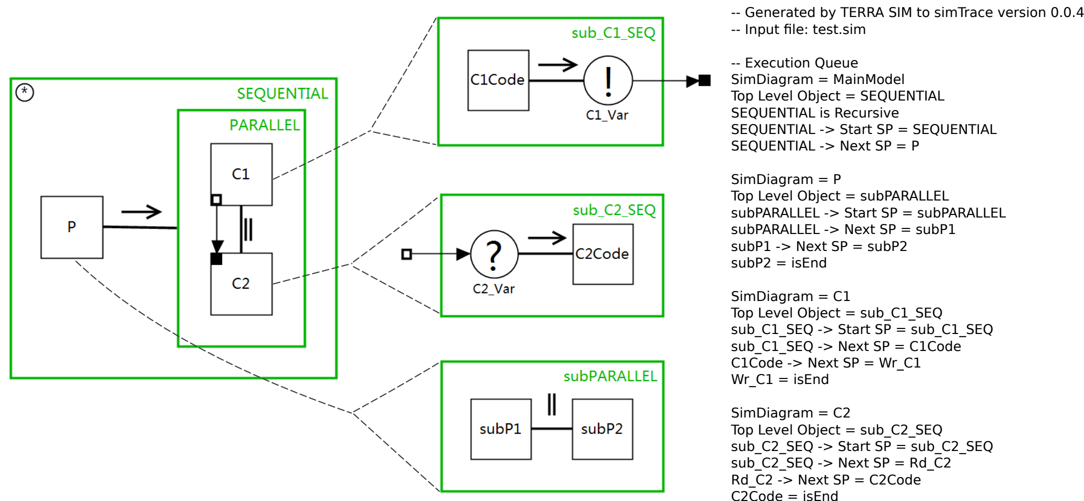


Figure 2. A TERRA model example and its simulated execution trace.

As we can see from the figure, although the simulated execution trace text indicates the process execution flow, neither signals to be varied nor results of algorithms were taken into account. Actually, source code manually added into code blocks cannot be handled. See C1Code to produce data and C2Code to consume data in Figure 2. However, the functionality of the algorithms have to be taken into account within the design loop, otherwise it will be meaningless with respect to a real-world CPS. Moreover, the execution order was visualized in the form of text which is obviously not elegant nor user-friendly. Users have to put a lot of energy into analysing traces, which is error-prone and inefficient.

Here, this paper presents a better approach for simulation and visualization supports to the TERRA tool suite.

2. Design for Simulation

2.1. Design Space Exploration

In the traditional discussion about MDD, the design of a system starts at a high level of abstraction. If a model is defined by a Domain-Specific Language (DSL), in our case the CSP meta-model we use, it always aims to achieve an easier and more formal assessment of problems in the modelled system. Lot of effort has been done to deal with assessment of models, in which one of the most popular and well-known method is about *executable models* [23,24].

2.1.1. Executable Models

An executable model is a model complete enough to be executable, and the 'executability' depends more on the execution tool than on the model itself. Some tools might be able to execute models that partially abstract the system while some others require more complete and precise models. In most situations the form or the type of the execution tool depends on what kind of assessment you need for the modelled system. For instance in our case, if we only want to check the software architecture to verify the absence of deadlock or livelock, then FDR and the generated machine-readable CSP (from TERRA models) are sufficient. Furthermore, if we need to know the execution order of processes then we need to compile

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

the TERRA model to a simulation model, and then generate the corresponding simulated execution trace. However, if not only taking the software architecture but also functional components such as control algorithms and physical dynamics into account, which are crucial and indispensable in CPS co-design, things become different and more complex. The completely designed executable model should represent key characteristics and behaviours of the system, meanwhile *functions* of the system need to be taken into account, where the former contributes to the system verification and the latter contributes to the system validation. Hence, we need to design and develop suitable tools to sufficiently assess the modelled CPS.

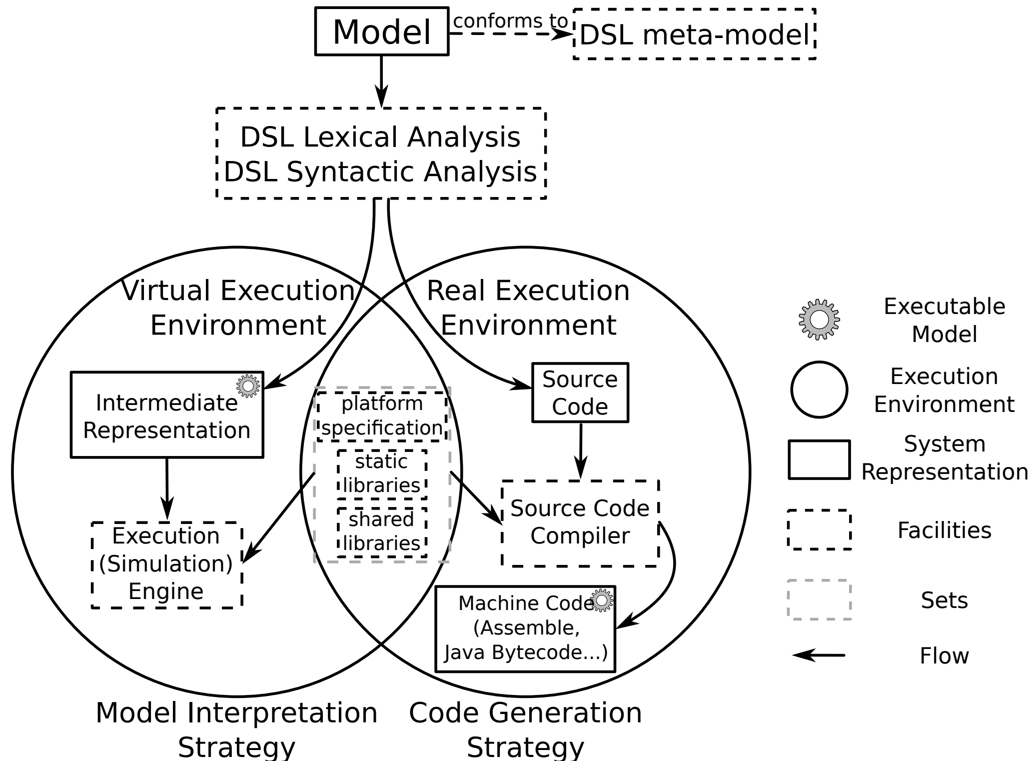


Figure 3. Two strategies to implement execution tools for models.

2.1.2. *Different Strategies: Code Generation*

In MDD, there are several ways to implement execution tools for models. Code generation and model interpretation are the most acceptable and common strategies. Figure 3 is the work flow of the two strategies mentioned above. The code generation strategy involves using a model compiler, which is usually defined as a model-to-text transformation (as what we used in the TERRA tool suite to generate C++ code from models). It aims to generate a lower-level representation of the modelled system using existing programming languages, e.g. C++. In this case, the generated code can be compiled, using a platform-dependent compiler and libraries, to build an executable system to run on an actual target platform. However, although the automatic code generation can ease lots of work for developers, there are still some disadvantages which are innate and hard to eliminate, for instance, the inflexibility. Assume when the execution target platform has been updated with respect to hardware or OS, code generation need to be modified as well to eliminate the discrepancies between different platform specifications. And moreover, most code generation techniques usually only provide skeletons thus produce fragments of code. This implies that developers still must manually add code of certain algorithms for functional purposes. Then there will be an 'asynchronization' between the refined model (intend to re-generate skeletons) and the manually added code.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

2.1.3. Different Strategies: Model Interpretation

Instead, the model interpretation strategy relies on the existence of a Virtual Execution Environment (VEE), also called a simulator or a Virtual Machine (VM), like a Java VM, which is able to directly interpret and execute the model. The model interpretation strategy will be more flexible comparing to the code generation strategy and it brings more opportunities to analyse and refine models. Since a model only needs to be interpreted by the VEE to proceed the execution. Hence, it becomes platform-independent as long as necessary facilities are provided to run the VEE, or it is only dependent on the VEE. Another benefit is that the model interpretation and the execution can be done dynamically, which also means it can enable changes to models at runtime. We have to confirm that internally the model interpretation strategy can be seen as a lightweight code generation, since the model need to be parsed and interpreted by the VEE.

One more thing that needs to be determined is whether the VEE is not essentially different from the target platform, then there will be no significant differences between the code generation strategy and the model interpretation strategy. For instance, the JVM provides platform runtime specifications and a bytecode interpreter to execute Java bytecode on any platform that can run the JVM. However, you need to represent your model in Java bytecode first, which means you have to translate your model to an intermediate representation that can be compiled to get a Java bytecode file. The compilation for intermediate code will be a serious amount of work unless there has been a compiler for this. The best choice is to interpret a model to Java source code or to Python source code which enables to directly use the javac compiler or the jython compiler. Then it is basically the same as the code generation strategy.

2.1.4. Obtaining Executable Models for CPS

Only models providing complete information on a system can be used to obtain fully operational and functional executable models, by generating source code from models for a target platform, or by interpreting models and executing intermediate representation in one or more virtual execution environments.

With respect to the model interpretation strategy and the code generation strategy, there are different options to obtain (one or more) executable models which can fully represent operations and functions of a cyber-physical system:

- Using the model interpretation strategy to interpret and execute models:
  - ∗ Design and develop different virtual execution environments (simulators) that contain different interpreters, then coordinate and incorporate multi simulated execution results.
  - ∗ Integrate different models by using model-to-model (M2M) transformations, then design and develop one VEE to interpret and execute the integrated model.
- Using the code generation strategy to obtain (one or more) executable models:
  - ∗ Generate source code (same language for single execution platform) from different models which contain certain interfacing properties to generate APIs for interacting purpose (an execution coordinator is needed as well).
  - ∗ Integrate different models by using M2M transformations then carrying out source code generation.

Design choices listed above can be categorized into two different types with respect to execution strategies. The former item listed under each execution strategy is classified as Loose-Coupling Execution (LCE) while the latter item is classified as Tight-Coupling Execution (TCE). Certain techniques need to be implemented to achieve goals respectively for dif-

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

Table 1. Design choices analysis of obtaining executable models for CPS: advantages (+) and disadvantages (-).

Advantage & Disadvantage / Model Coupling / Execution Strategies		Loose-Coupling Execution LCE		Tight-Coupling Execution TCE
Model Interpretation	+	No M2M transformation	+	Single VEE (and interpreter)
	+	Flexible to update models (separately and might at runtime)	-	M2M transformation is crucial
	-	Multi VEE (and interpreters)	-	Inflexible to update models due to consistency after integration
	-	Manually programming in the end	-	Manually programming in the end
	-	Coordinate and incorporate multi simulated execution results		
Code Generation	+	No M2M transformation	+	Single execution platform
	+	Single execution platform	+	Single code generation engine
	+	Source code will be at hand	+	Source code will be at hand
	-	Multi code generation engines	-	M2M transformation is crucial
	-	'Asynchronization' between added code and models	-	'Asynchronization' between added code and models
	-	Extra model interfacing properties		
	-	Extra execution coordinator		

ferent design choices. Advantages and disadvantages are analysed with respect to obtaining executable models for CPS, as shown in Table 1.

2.2. A Hybrid Approach for the Simulation

We propose a hybrid approach for simulation with respect to our way of working in order to mitigate as much as possible the disadvantages mentioned in Table 1. Figure 4 is the overall structure of the hybrid approach which is separated into three different layers.

As mentioned before, the tool chain we currently use consists of TERRA and 20-sim, which cover certain scope on modelling the cyberspace as well as the physical world. As shown in Figure 4, in the modelling layer (at the top of the figure) the 20-sim tool is capable of modelling a loop control system for a robotic set-up in which components, like signal generators, controllers, A/D converters and plant dynamics, are connected with each other through ports. Both the model-to-model transformation and the code generation are involved as intermediate steps, such that each 20-sim component can be transformed to a TERRA CSP model and generate relevant C++ source code for functional algorithms. Meanwhile, the TERRA tool suite is capable of integrating those TERRA CSP models which can represent the loop control system architecture. Additionally, TERRA CSP models can be transformed to machine-readable CSP models for verification by FDR.

If considering work flows of two strategies to obtain executable models as discussed before but from an opposite way, we should notice that we have already got some C++ code at hand, generated from 20-sim, representing control algorithms. Then, the problem is quite

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

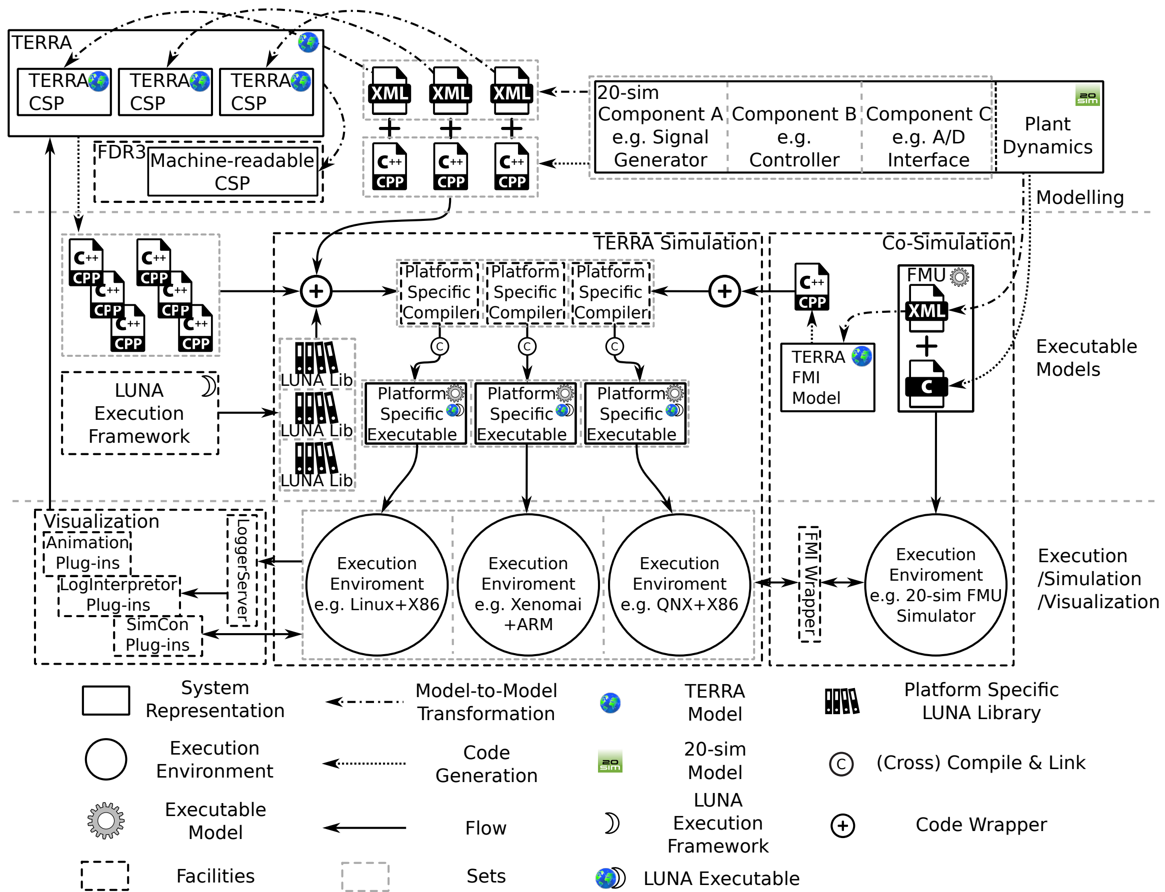


Figure 4. A hybrid approach for simulation.

obvious that we need to determine how to choose from those two strategies to obtain executable models and which one can be a better choice. The first option is using the model interpretation strategy to interpret models into some intermediate representation that contains sufficient semantics to make use of the C++ code generated from 20-sim models. Besides, a certain execution environment is required to support a synchronized execution between interpreted models and C++ code. Although we have designed a simulation meta-model to interpret TERRA models, unfortunately it is not sufficient with respect to requirements mentioned above.

And since we have already got some C++ code at hand, will it be a better choice to take advantage from the code generation strategy to obtain executable models? From a practical point of view but more convenient for development and experimentations, the answer is *Yes*. The TERRA tool suite is capable of generating C++ code from TERRA CSP models. Moreover, the LUNA execution framework [25] can provide execution libraries which support hard real-time, multi platforms and are CSP-capable. Then, with a standard and commonly used C++ compiler for the execution platform (e.g. g++ for Linux or qcc for QNX [26]), we can successfully build an executable binary (i.e a LUNA executable or LUNA application) which is also a representation of the modelled system such that it can be seen as an executable model as well. Although that executable model is built for a specific platform, it will not influence functional criteria which can be presented during execution or simulation (e.g. process execution flow, signals to be varied and functions of algorithms).

Once we obtain executable models through the code generation strategy, both the LCE and the TCE can be used in our hybrid approach regarding to the model coupling criterion in Table 1. From the physical-world perspective, modelled plant dynamics can be transformed to a Functional Mock-up Interface (FMI) [27] model which are commonly used in

co-simulation, and relevant C functions will be generated as well. The FMI model and C code will be wrapped together to obtain an executable model defined as a Functional Mock-up Unit (FMU) which can be used to achieve co-simulation.

The FMI model can also be transformed to a TERRA FMI model (tight-coupling) which provides interfacing definitions like unit and variable references, which have been prototypically implemented and validated with respect to our hybrid approach. However, they are not in the scope of this paper. Combining FMI techniques and our hybrid approach, it is able of interacting with modelled plant dynamics (loose-coupling) simulated by the 20-sim FMU simulator (still work in progress by Controllab Products B.V. [16]).

3. Visualizing the Simulation

The visualization, or to be more specific, the visual representation of simulation results, which should be presented to users for analysing behaviour of the system, is also required in our hybrid approach. Simultaneously, system functionality, such as process execution flow, signals to be varied and results of algorithms, need to be assessed during the simulation. In order to support our hybrid simulation approach which is actually execution-based, numerical and functional, a new visualization design is proposed which implement the animation and other relevant facilities in the tool chain.

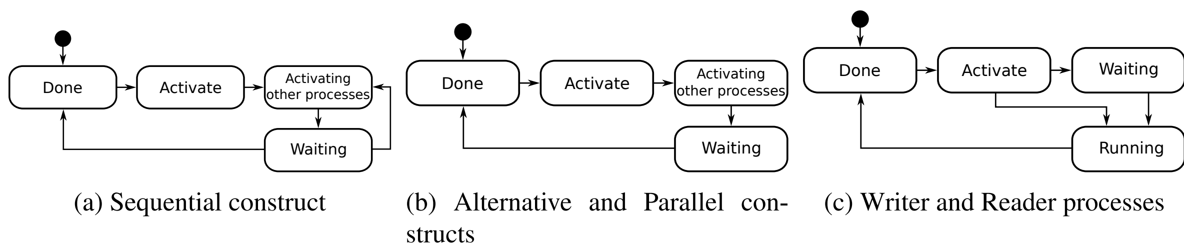
3.1. Tracing the Execution Flow in LUNA

3.1.1. States for CSP Constructs and Processes

In LUNA, a CSP construct emits an event when it is activated. Whilst active, it may activate other processes and wait for them to finish. When this is all done, it has finished and emits another event. A CSP process (e.g. a Reader, a Writer or a 'normal' process) emits an event when activated, when it has to go into a waiting state (for its partner Writer or Reader), when it is running and when it is done. Hence, there are five possible states defined in total for CSP constructs and processes: *Activate*, *Running*, *Done*, *Waiting*, and *Activating other processes*. All state changes generate events and these can be traced during execution to indicate process execution order, which enables better understanding and analysing with respect to CSP specifications.

Figure 5 shows state machine diagrams of certain CSP constructs and processes. The CSP Sequential construct needs to wait for a child to finish before activate the next one, it transitions between *Activating other processes* and *Waiting* until the last child has been activated and has finished, as shown in Figure 5a. Figure 5b is the state machine diagram for the CSP Alternative and the Parallel constructs. For the Parallel construct, it remains in state *Activating other processes* until all children have been activated; in *Waiting*, it waits until all children have finished. As to the Alternative construct, *Activating other processes* applies to activate a single child, one for which the guard expression is met. It might have to wait for this to happen. If more than one guard expression are satisfied, any of them can be chosen, unless the guard order is prioritised, in which case the one with highest priority is chosen. In *Waiting*, the Alternative waits for the activated child to finish. Regarding the rendezvous communication in CSP, it must indicate whether the Writer or the Reader on the other end of the channel is ready for communication or not. Hence, in the state machine diagram for the CSP Writer and the Reader processes, as shown in Figure 5c, after a Writer or a Reader is in the state *Activate*, it transitions to *Running* if and only if the corresponding Reader or Writer is already in *Activate* or is *Waiting* and takes that corresponding Reader or Writer also to *Running*, otherwise, it transitions to *Waiting*. Care must be taken to avoid race hazards so that, for example, they do not both enter their *Waiting* state.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.



(a) Sequential construct (b) Alternative and Parallel constructs (c) Writer and Reader processes

Figure 5. State machine diagrams of CSP constructs and processes, modified from (Ran [28]).

3.1.2. Recording and Transmitting the Execution Flow

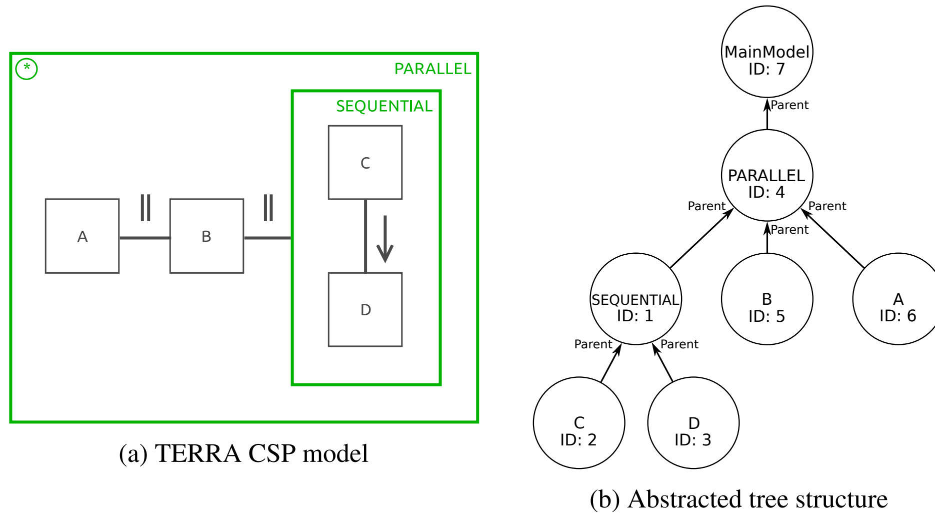
In order to record and transmit execution flow information for a LUNA executable,as shown in Figure 4, which includes discrete state changes that occur for all CSP constructs and processes, as well as their ordering, an original design of LUNA's real-time logging facility was proposed as a proof-of-concept [29]. The logging facility mainly consists of two parts: the logger and the log receiver. The logger is integrated within a developed LUNA executable, or to put it another way, it is part of a LUNA executable. When the logging function is being called by certain objects during execution, it only pays for placing the log data into a large buffer. The logger thread then takes care of transmitting data whenever computational resources are unused such that it will not break the real-time constrains. Anther standalone receiver program named 'loggerServer' will run on the development platform (before the logger starts). The logger server receives and stores log data as files in Comma-Separated Values (CSV) format.

The logger facility for recording state changes can be divided into two phases: the registration phase, and the state recording phase. In the registration phase, the logger intends to register (map) each CSP construct or process to an element (named 'Channel' in Figure 6c) with a specific ID, a name and a type when they are being initialized. The original logger facility has been modified and partially re-factored towards animation facilities (e.g. registering ParentID and updating previous registration), in order to obtain a tree structure model, stored as a CSV file named 'logfile' that represents the hierarchical structure of the executable model. Each CSP construct or process can be defined as a tree node, and each tree node could have one parent and some siblings. A simple tree structure in different presenting forms is shown in Figure 6.

When in the state recording phase, each time when a certain CSP construct or a process transitions from one state to another, the logger will update an element value (state) in a vector whose indices correspond to IDs of all tree nodes, and then places the data into a buffer. Once computational resources are available, the logger thread will transmit the buffered data to the log receiver to store them into a CSV file named 'datafile'. Figure 7 shows the data structure that is used for storing state changes. Each update (state transition) is stored as one line in the CSV file by the log receiver enabling a single entry per line for animation facilities.

Moreover, as discussed before, our hybrid simulation approach is execution-based which should be capable of providing numerical and functional assessment for an executable model (or a modelled system). Hence, during the execution, various signal or variable values need to be logged as well. In principle, it is the same as the logging procedure for state changes. Each interested signal or variable needs to be registered with a specific ID and then can be logged when its value varies. In our current progress an easier alternative implementation is done for prototyping, which directly outputs varied values of a specified signal or a variable through the logger. The output messages are handled automatically by the logger and the log receiver, to save them into a file, also in CSV format, and it does not require registration beforehand.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.



(a) TERRA CSP model

(b) Abstracted tree structure

```
0.000: Date: 15-6-2016
0.000: Measurement start is: 13:9:52:620037722
0.000: TicksPerMilliSecond: 1899992
0.000: Registered channel - NAME: Sequential ID: 1 TYPE: 2
0.000: Channel names: names = {'Sequential' }
0.000: Registered channel - NAME: C ID: 2 TYPE: 2 PARENT_ID: 1
0.000: Channel names: names = {'Sequential' 'C' }
0.000: Registered channel - NAME: D ID: 3 TYPE: 2 PARENT_ID: 1
0.000: Channel names: names = {'Sequential' 'C' 'D' }
0.000: Updated channel - ID: 1 NEWNAME: SEQUENTIAL
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' }
0.000: Registered channel - NAME: Parallel ID: 4 TYPE: 2
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'Parallel' }
0.000: Registered channel - NAME: B ID: 5 TYPE: 2 PARENT_ID: 4
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'Parallel' 'B' }
0.000: Updated channel - ID: 1 NEWNAME: SEQUENTIAL NEWPARENT_ID: 4
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'Parallel' 'B' }
0.000: Registered channel - NAME: A ID: 6 TYPE: 2 PARENT_ID: 4
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'Parallel' 'B' 'A' }
0.000: Updated channel - ID: 4 NEWNAME: PARALLEL
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'PARALLEL' 'B' 'A' }
0.000: Registered channel - NAME: MainModel ID: 7 TYPE: 2
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'PARALLEL' 'B' 'A' 'MainModel' }
0.000: Updated channel - ID: 4 NEWNAME: PARALLEL NEWPARENT_ID: 7
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'PARALLEL' 'B' 'A' 'MainModel' }
0.000: Registered channel - NAME: Terminator ID: 8 TYPE: 2
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'PARALLEL' 'B' 'A' 'MainModel' 'Terminator' }
0.000: Updated channel - ID: 7 NEWNAME: MainModel NEWPARENT_ID: 8
0.000: Channel names: names = {'SEQUENTIAL' 'C' 'D' 'PARALLEL' 'B' 'A' 'MainModel' 'Terminator' }
```

(c) Tree structure logged from a LUNA executable

Figure 6. Tree structure representation of a TERRA CSP model.

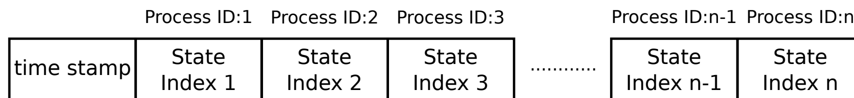


Figure 7. Data structure for storing state changes.

3.2. *Visualizing Traced Log Data*

As discussed in the previous section, the LUNA logging facility is capable of tracing the execution flow of an executable model by recording its log information. The tree structure of the modelled system contains registration information (mapped with IDs, names, etc) for each CSP construct and process, as well as state changes during execution are recorded and transmitted. However, if state changes that represent the execution flow cannot be shown in the form of visualization to designers, it will hinder designers to gain more insight into models, since logged data are many lines of varied values which stand for different states of each registered CSP construct and process. That is obviously inefficient and unnecessary to analyse by designers 'manually'.

Regarding to our hybrid approach, when a LUNA executable model runs on an execu-

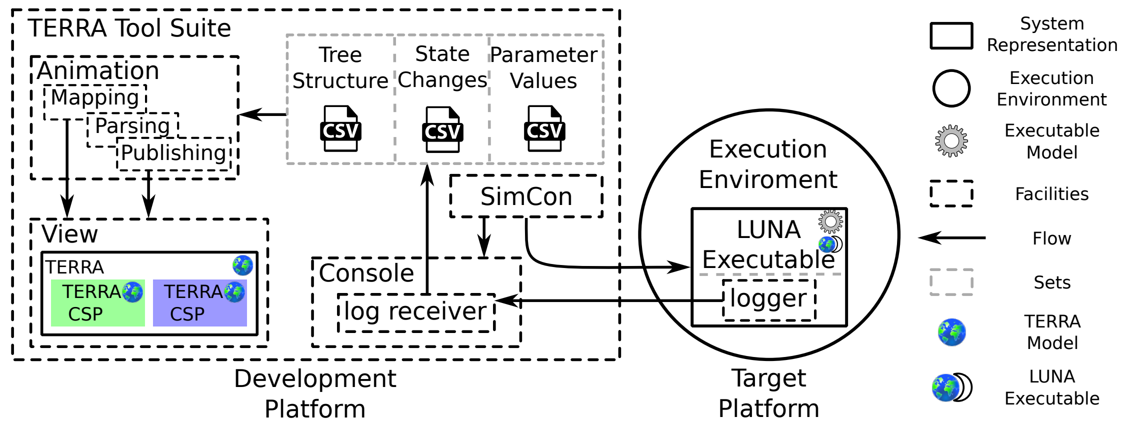


Figure 8. Overall structure of the visualization.

tion environment, its logged data will be transmitted to the development platform at runtime by the logger and be received by the log receiver. Since the LUNA executable model is compiled from certain C++ code that generated from TERRA models, meanwhile those TERRA models are graphically designed and presented, thus logged data can be reused to depict state changes of all CSP model elements as well as the execution order of processes by the form of animation in the TERRA tool suite. Hence, certain facilities are designed and developed in order to visualize traced log data in order to meet all requirements as discussed before. In Figure 8, the overall structure of our visualization design is presented. All visualization related facilities within the scope of the TERRA tool suite are implemented as sets of Eclipse plugins, which includes the SimCon facility, the Animation facility and other relevant facilities like a Console for the log receiver and user interface.

The SimCon facility mainly consists of execution parameters configuration for both the simulation (like simulation step size, simulation end time, etc) and the logging facility (file names, port number, etc), as well as execution control (like start/stop/pause/resume) for the simulation. It can also help to mimic a 'slow' simulated execution in order to provide users a better visual experience by setting a parameter which aims to configure time delay between steps during the simulation. Otherwise, since logged data will be treated as snapshots during the simulation, if it flows too fast then it will become like a flash which is impossible for designers to observe. In our current prototype this is implemented as a simple mechanism that either adds a timer to the LUNA executable or, manually, provides a pause/resume facility on the Console with which the user can intervene during execution.

The Animation facility is the core part of the visualization design. Once the simulation starts, the logging facility will log certain data and store them into corresponding CSV files. The first file is the tree structure file that logged and stored during the initialization phase for different objects from code perspective, which register each CSP construct or process to an element with a specific ID, as shown in Figure 6c. Then, it is the state changes file that logged and stored during the execution phase which represent different states for registered elements. The last one is the signal values CSV file which stores varied values for one or more specified signals. The animation facility will first map each TERRA model component with a corresponding element in the tree structure file. Then it will read and parse the state changes data line by line, since each line is defined as a single entry for the animation, or put it another way, each line represents a snapshot of the simulation. After reading and parsing one log entry, a database object will be updated and then a new snapshot will be published to subscribers (the graphical view and the textual view). Human observers are good at spotting differences, especially from a graphical view where colour changes instantly command attention. Additionally, the database object can also be used to enable backward stepping capability, which will bring advantage to system validation. In our current implementation, we

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

use different colours to graphically identify states for each TERRA model component, and a textual view is provided as well. Although varied signal values are stored in a file, they are not visualized in any kinds of views yet which is also part of our future work.

Summarizing, continuous cycles of *logging* → *reading* → *parsing* → *updating snapshots* → *publishing snapshots to subscribers* → *show difference between snapshots* form an animation for the simulated execution.

4. Results

To illustrate our hybrid simulation approach and visualization facilities, a simple but classic model structure is used to mimic a traditional loop control system, as shown in Figure 9. At the top, it is a 20-sim model that represents the system, which consists of a step signal generator, a P-controller and the plant dynamics modelled as a liner system, in state space form. In the middle, it is a TERRA model which contains three top-level CSP processes (in Parallel), namely the Step, the Controller and the Plant blocks, whose sub-level models are shown at the bottom and are transformed from the corresponding 20-sim model. Simultaneously, algorithms (C++ code) for corresponding code blocks (the XXStepModel, the XXControllerModel and the XXLinearSystemModel) are generated by 20-sim.

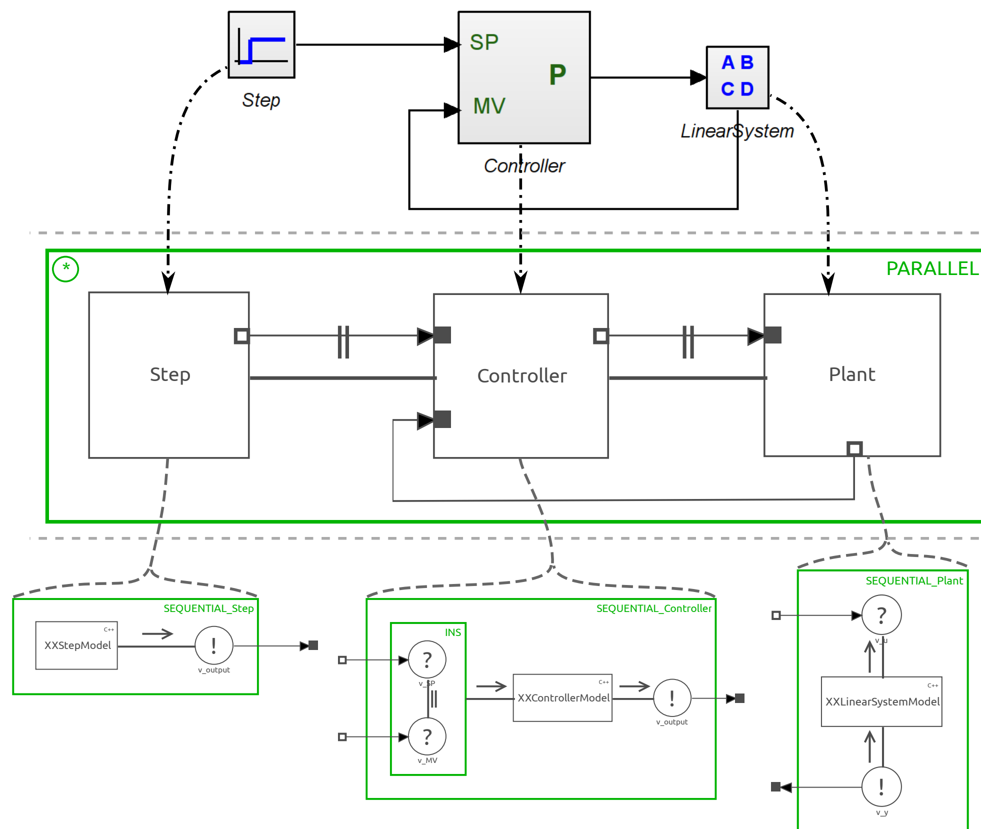


Figure 9. Example: models for a loop control system.

4.1. Generated C++ Code

C++ code is generated from our testing TERRA models. Figure 10a shows fragments of code in the constructor of the top-level model which is shown in the middle part of Figure 9. It mainly consists of the initialization for top-level objects, such as *channels*, *model objects* and *groups*, which are corresponding to CSP channels, processes and constructs. At the top-

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

```
MainModel::MainModel() :
    Recursion<CSProcess>()
{
  SETNAME(this, "MainModel");

  // Initialize channels
  myControlleroutput_to_PlantuChannel =
           new UnbufferedChannel<double, One2In, Out2One>();
  myPlanty_to_ControllerMVChannel =
           new UnbufferedChannel<double, One2In, Out2One>();
  myStepFunctionoutput_to_ControllerSPChannel =
           new UnbufferedChannel<double, One2In, Out2One>();

  // Initialize model objects
  myController = new Controller::Controller(myPlanty_to_ControllerMVChannel,
           myStepFunctionoutput_to_ControllerSPChannel,
           myControlleroutput_to_PlantuChannel);
  SETNAME(myController, "Controller");
  myPlant = new LinearSystem::LinearSystem(myControlleroutput_to_PlantuChannel,
           myPlanty_to_ControllerMVChannel);
  SETNAME(myPlant, "Plant");
  myStepFunction = new Step::Step(myStepFunctionoutput_to_ControllerSPChannel);
  SETNAME(myStepFunction, "StepFunction");

  // Create PARALLEL group
  myPARALLEL = new Parallel(
    (CSPConstruct *) myController,
    (CSPConstruct *) myPlant,
    (CSPConstruct *) myStepFunction,
    NULL
  );
  SETNAME(myPARALLEL, "PARALLEL");

  // Register PARALLEL as top-level recursive object
  setToActivate(myPARALLEL);
  setEvaluateCondition(true);
}
```

(a) Code generated from a top-level model

```
Controller::Controller(ChannelOut<double> *MV,
            ChannelOut<double> *SP, ChannelIn<double> *output) :
     Sequential(NULL)
{......
  myr_MV = new Reader<double>(&v_MV, MV);
  SETNAME(myr_MV, "r_MV");
  myr_SP = new Reader<double>(&v_SP, SP);
  SETNAME(myr_SP, "r_SP");
  myw_output = new Writer<double>(&v_output, output);
  SETNAME(myw_output, "w_output");
......
  // Register model objects
  this->append_child(myINS);
  this->append_child(myXXControllerModel);
  this->append_child(myw_output);
}
```

(b) Code generated from a sub-level model

Figure 10. Example: C++ code generated from TERRA CSP models (Figure 9).

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

level of our testing model there are three channels, three CSP processes and one CSP Parallel construct. Meanwhile, code is generated from sub-level models. Figure 10b shows part of the constructor code, which is generated from the bottom-middle part of Figure 9, namely the sub-level model of the Controller process, for which it also registers child model objects.

4.2. Simulation Comparison

After generating C++ code from the integrated TERRA model, we compiled and linked all C++ code together with a platform-specific LUNA library. The LUNA library used in our test was built for QNX real-time OS (32-bit, on X86). An executable model (binary for QNX on X86) that represents a whole CPS was obtained. Then, on a QNX virtual machine, the executable model was simulated as Tight-Coupling Execution (TCE), the execution flow and signal values were traced and recorded by the logging facility. Figure 11 is the plot diagram of recorded signal values during simulation. 20-sim simulation executed for 10 s (simulated), while our hybrid simulation executed for 16 s. 20-sim simulation results are treated as the ground truth in our case. Although we can see there are minor differences between our simulation results and 20-sim, they are quite limited and are within numerical tolerance.

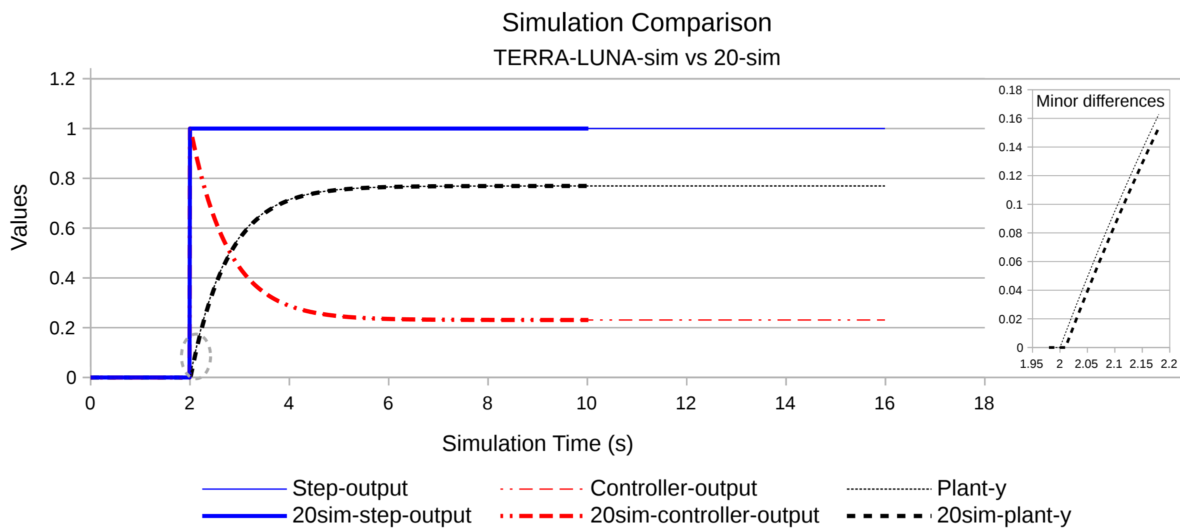


Figure 11. Comparison between different simulation results, the Minor differences box is a blow-up of the dashed oval area towards the bottom-left.

4.3. Visualization Test

Besides numerical signal values, state changes during the execution were recorded as well to support visualization facilities. Figure 12 shows the tree structure mapped by the logger during test, which is a kind of in-order traversal. Each node with a PAR or SEQ footnote represents a CSP construct, and the one without a footnote is a CSP process. Meanwhile, each model element was registered with a specific ID except three SEQ constructs, namely the SEQUENTIAL_Controller, the SEQUENTIAL_Plant and the SEQUENTIAL_Step. It is a kind of optimization during the registration phase, since the parental node of each construct mentioned above is a CSP process that only contains a single child, of which the processing will be directly after its parental node. Each time when a state transition happened on a specified process, a new state value was updated in a mapped vector to refresh the previous process state. State changes were stored in a CSV file where each line stands for a states snapshot for all processes that being simulated. Since TERRA models are graphically presented, state changes, which are dynamically stored during simulation will be published to subscribers, being the graphical view and the textual view simultaneously. Various states (e.g.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

Activate, Activating other processes, Waiting, Running and Done) are presented by different colours, as shown in Figure 13a. Figure 13b is the states snapshot stored in the log data file, which was published to the graphical view during our test, at which the Step (ID: 11) and the Controller (ID: 4) were in state *Waiting (1)* while the Plant (ID: 7) was just turning into state *Activate (3)*. Snapshots that are dynamically published to the graphical view form the so-called animation, which can help users to analyse how the loop control flow works.

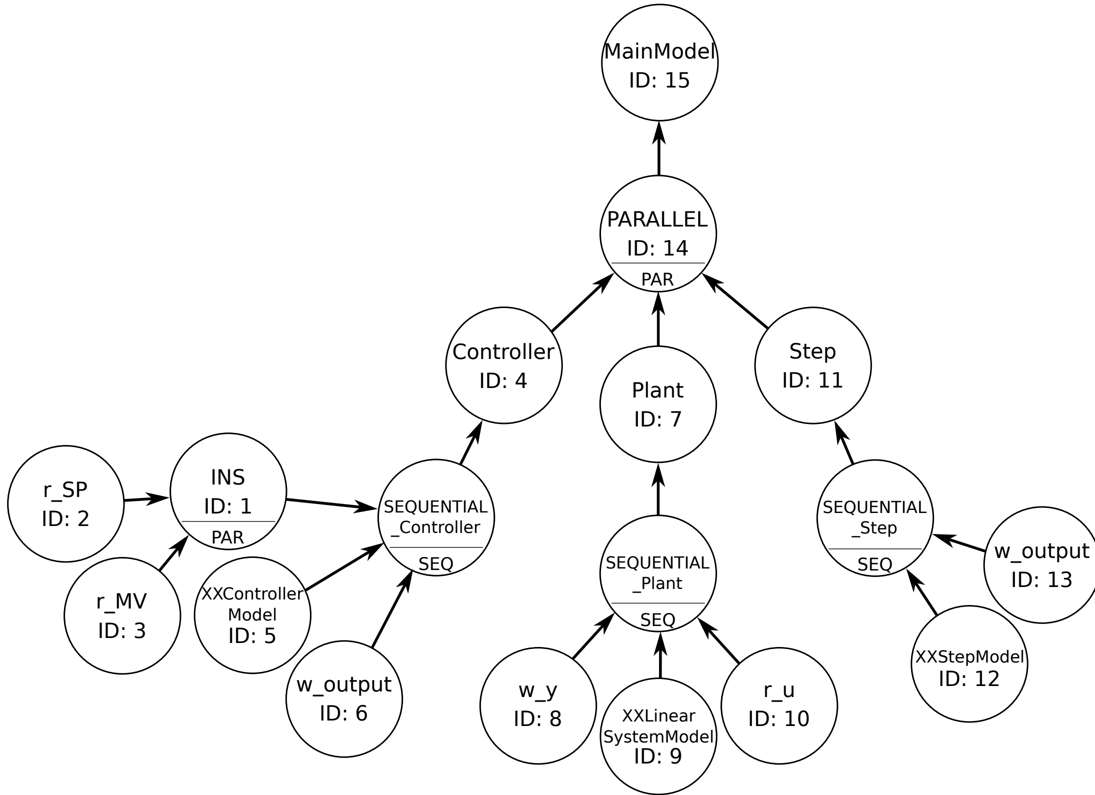
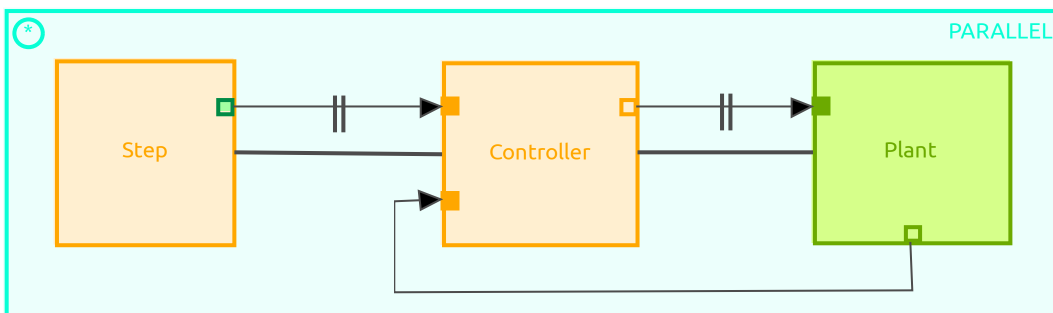


Figure 12. Example: the tree structure of the loop control model (Figure 9), the arrow-line of each node points to its parent.



(a) Animation view

Process ID	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
time stamp 0.230	1	1	1	1	0	0	3	0	0	0	1	0	2	4	4

(b) States snapshot in the log data file

Figure 13. Example: one snapshot during simulation for the loop control model.

4.4. Repeatable Execution Flow

Despite the fact that our test model is a simple loop control system, it contains quite some states and paths through those states. Figure 14 shows only part of the state transition diagram for our test model. For the simulation of more complex systems, a state space explosion may (and, usually, will) occur, which quickly gets beyond our capability to analyse without making mistakes. This problem is considerably aggravated without visualization tools to aid the presentation of the results from the simulation.

Our simulation follows an execution of the model, where state changes information is logged for each process in the model during execution. Only that one execution sequence is traced. For any non-trivial model, many execution orderings are possible because of the freedom allowed by the CSP Parallel and Alternative constructs. If a simulation run shows up a problem that manifests itself only for the execution ordering that happened during that run, we may need to repeat that run ordering many times in order to spot the circumstances that caused the bug. Happily, we can do this since the activation mechanism in LUNA for Parallel and Alternative is resolved using pseudo-random numbers [7]. All that is needed is to record the initial random number seed used by each run and, then, force its reuse if the run needs to be repeated with the same execution ordering.

Figure 14 shows a particular execution flow (black) that we may want to repeat (because the behaviour presented is interesting and that presentation is sensitive to the ordering within that flow). Numerous other flows are shown in grey.

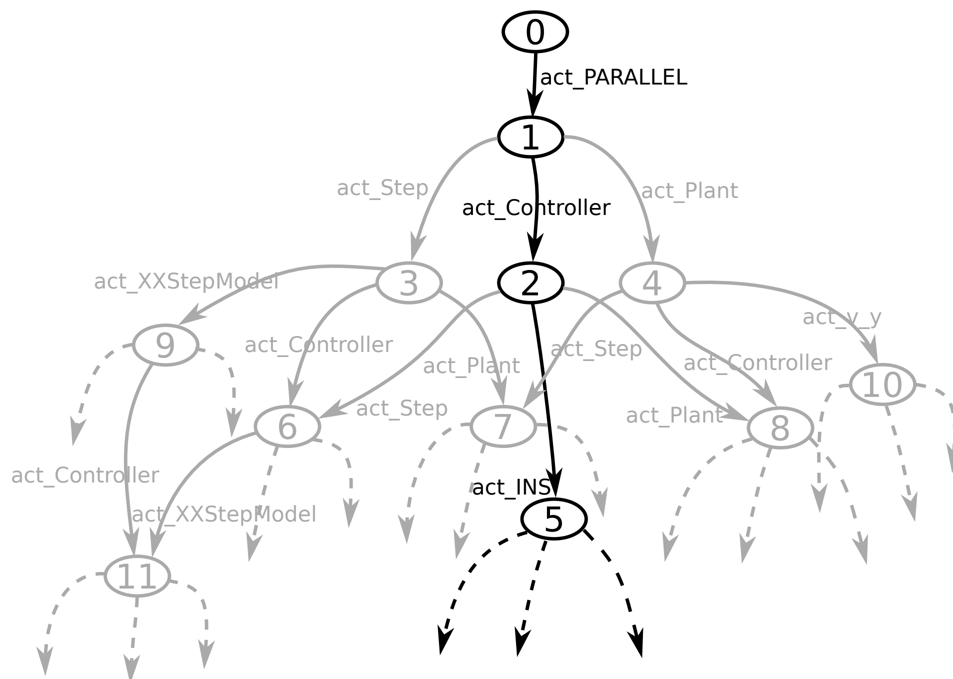


Figure 14. Simplified and partial state transition diagram ('act' stands for 'activate').

5. Conclusions and Recommendations

The simulation comparison showed that our hybrid simulation approach is working as intended. It provide comparable results as the ground truth simulated in 20-sim. Although there are minor differences in values, they are within numerical tolerance from control perspective. The visualization test showed that the animation in TERRA is sufficient to indicate simulated execution flow and states of processes animated are consistent with traced execution data.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

With the visualization, it is easier for designers to observe simulation results, which also means it is easier to analyse state space and helps to gain more insight into models.

Moreover, following our hybrid simulation approach and using the visualization facilities, it can provide opportunity to implement a rapid prototyping of a system for validation, which can reduce developing costs in both time and money. Furthermore, since model-driven development is the fundamental basis in our hybrid approach which towards obtaining an executable model for a target platform, it also brings opportunity to obtain an executable and deployable binary which can be right-first-time after refinements through visualized simulation.

In our current progress, although the varied signal values during simulation can be stored in a file, they cannot be automatically visualized as state changes. This is crucial for numerical assessment of a modelled CPS. Moreover, the log receiver is not integrated into the TERRA tool suite which brings extra overhead. Additionally, the number of logged state changes is quite high which make orientation difficult. On the other hand, such a level of detail may be needed for analysing and debugging. Therefore, there should be options to include or exclude states from animations. Lastly, timing analysis need to be implemented as well, since real-time performance is one important criterion in CPS especially for modern service robotics which toward seamless interaction with environments.

References

[1] E. A. Lee. Cyber Physical Systems: Design Challenges. In *Proceedings of the 11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC)*, pages 363–369. IEEE, May 2008.

[2] L. Sha, S. Gopalakrishnan, X. Liu, and Q. Wang. Cyber-Physical Systems: A New Frontier. In *Proceedings of the IEEE International Conference on Sensor Networks, Ubiquitous and Trustworthy Computing (SUTC '08)*, pages 1–9, June 2008.

[3] R. R. Rajkumar, I. Lee, L. Sha, and J. Stankovic. Cyber-Physical Systems: the Next Computing Revolution. In *Proceedings of the 47th Design Automation Conference (DAC)*, pages 731–736. ACM, 2010.

[4] P. Derler, E. A. Lee, and A. S. Vincentelli. Modeling Cyber-Physical Systems. *Proceedings of the IEEE*, 100(1):13–28, 2012.

[5] C. A. R. Hoare. Communicating Sequential Processes. *Communications of the ACM*, 21(8):666–677, 1978.

[6] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A Theory of Communicating Sequential Processes. *Journal of the ACM (JACM)*, 31(3):560–599, 1984.

[7] Z. Lu, M. M. Bezemer, and J. F. Broenink. Model-Driven Design of Simulation Support for the TERRA Robot Software Tool Suite. In *37th WoTUG Technical Meeting - Communicating Process Architectures 2015*, pages 257–272, Canterbury, UK, August 2015. Open Channel Publishing Ltd.

[8] M. M. Bezemer, M. A. Groothuis, and J. F. Broenink. Way of Working for Embedded Control Software using Model-Driven Development Techniques. In *Proceedings of the IEEE ICRA Workshop on Software Development and Integration in Robotics (SDIR VI)*, pages 6–11, May 2011.

[9] J. Eker, J. W. Janneck, E. A. Lee, J. Liu, X. Liu, J. Ludvig, S. Neuendorffer, S. Sachs, and Y. Xiong. Taming Heterogeneity - the Ptolemy Approach. *Proceedings of the IEEE*, 91(1):127–144, 2003.

[10] J. C. Jensen, D. H. Chang, and E. A. Lee. A Model-Based Design Methodology for Cyber-Physical Systems. In *Proceedings of the 7th International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 1666–1671. IEEE, 2011.

[11] J. F. Broenink, Y. Ni, and M. A. Groothuis. On Model-Driven Design of Robot Software Using Co-Simulation. In *Proceedings of the International Conference on Simulation, Modeling, and Programming for Autonomous Robots (SIMPAR), workshop on Simulation Technologies in the Robot Development Process*, pages 659–668. TU Darmstadt, November 2010.

[12] J. F. Broenink, M. A. Groothuis, P. M. Visser, and M. M. Bezemer. Model-Driven Robot-Software Design Using Template-Based Target Descriptions. In *Proceedings of the ICRA 2010, workshop on Innovative Robot Control Architectures for Demanding (Research) Applications*, pages 73–77. IEEE, May 2010.

[13] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. Design and Use of CSP Meta-Model for Embedded Control Software Development. In *Proceedings of the Communicating Process Architectures 2012*,

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

pages 185–199. Open Channel Publishing Ltd., August 2012.
[14] M. M. Bezemer. *Cyber-Physical Systems Software Development: Way of Working and Tool Suite*. PhD thesis, University of Twente, November 2013.
[15] B. Scattergood. The Semantics and Implementation of Machine-Readable CSP, 1998.
[16] Controllab Products B.V. 20-sim website. http://www.20sim.com/, visited on 2016-06-01.
[17] J. F. Broenink. Modelling, Simulation and Analysis with 20-sim. *Journal A*, 38(3):22–25, September 1997.
[18] C. Larman and V. R. Basili. Iterative and Incremental Developments: A Brief History. *Computer*, 36(6):47–56, 2003.
[19] D. C. Schmidt. Model-Driven Engineering. *Computer*, 39(2):25–31, 2006.
[20] J. Sztipanovits. Composition of Cyber-Physical Systems. In *Proceedings of the 14th Annual IEEE International Conference and Workshops on the Engineering of Computer-Based Systems (ECBS '07)*, pages 3–6, March 2007.
[21] T. Gibson-Robinson, P. Armstrong, A. Boulgakov, and A. W. Roscoe. FDR3 - A Modern Refinement Checker for CSP. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 8413 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 2014.
[22] P. H. Welch and J. B. Pedersen. Santa Claus: Formal Analysis of a Process-oriented Solution. *ACM Trans. Program. Lang. Syst.*, 32(4):14:1–14:37, April 2010.
[23] L. Bassi, C. Secchi, M. Bonfe, and C. Fantuzzi. A SysML-Based Methodology for Manufacturing Machinery Modeling and Design. *IEEE/ASME Transactions on Mechatronics*, 16(6):1049–1062, Dec 2011.
[24] G. D. Kapos, V. Dalakas, A. Tsadimas, M. Nikolaidou, and D. Anagnostopoulos. Model-Based System Engineering Using SysML: Deriving Executable Simulation Models with QVT. In *Systems Conference (SysCon), 2014 8th Annual IEEE*, pages 531–538, March 2014.
[25] M. M. Bezemer, R. J. W. Wilterdink, and J. F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In *Proceedings of the Communicating Process Architectures 2011*, pages 157–175. IOS Press BV, June 2011.
[26] QNX Software Systems Limited. QNX website. http://www.qnx.com/, visited on 2016-06-20.
[27] T. Blochwitz, M. Otter, J. Akesson, et al. Functional Mockup Interface 2.0: The Standard for Tool Independent Exchange of Simulation Models. In *Proceedings of the 9th International MODELICA Conference; September 3-5; 2012; Munich; Germany*, number 076, pages 173–184. Linköping University Electronic Press, 2012.
[28] T. C. Ran. Design of Animation Facilities for Analysing Cyber-Physical System Software Architectures. Master's thesis, EEMCS, University of Twente, September 2015. `https://www.ram.ewi.utwente.nl/aigaion/attachments/single/1302`.
[29] R. Wilterdink. Design of a Hard Real-time, Multi-threaded and CSP-capable Execution Framework. Master's thesis, EEMCS, University of Twente, June 2011. `http://essay.utwente.nl/61066/`.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.

CPA 2016 preprint – the proceedings version may have other page numbers and may have minor differences.