Communicating Process Architectures 2012
P.H. Welch et al. (Eds.)
Open Channel Publishing Ltd., 2012

Design and Use of CSP Meta-Model for Embedded Control Software Development

Maarten M. BEZEMER [1], Robert J.W. WILTERDINK and Jan F. BROENINK

Robotics and Mechatronics (formerly CE), Faculty EEMCS,
University of Twente, The Netherlands.

{M.M.Bezemer , R.J.W.Wilterdink , J.F.Broenink} @utwente.nl

Abstract. Software that is used to control machines and robots must be predictable and reliable. Model-Driven Design (MDD) techniques are used to comply with both the technical and business needs. This paper introduces a CSP meta-model that is suitable for these MDD techniques. The meta-model describes the structure of CSP models that are designed; using this meta-model it is possible to use all regular CSP constructs when constructing a CSP model. The paper also presents a new tool suite, called TERRA, based on Eclipse and its frameworks. TERRA contains a graphical CSP model editor (using the new CSP meta-model), model validation tools and code generation tools. The model validation tools check whether the model conforms to the meta-model definition as well as to additional rules. Models without any validation problems result in proper code generation, otherwise the developer needs to address the found problems to be sure code generation will succeed. The code generation tools are able to generate CSPm code that is readable by FDR and to generate C++/LUNA code that is executable on embedded targets. The meta-model and the TERRA tool suite are tested by designing CSP models for several of our laboratory setups. The generated C++/LUNA code for the laboratory setups is able to control them as expected. Additionally, the paper contains an example model containing all supported CSP constructs to show the CSPm code generation results. So it can be concluded that the meta-model and TERRA are usable for these kind of tasks.

Keywords. co-simulation, code generation, CSP, eclipse, EMF, LUNA, meta-model, model transformations, model verification, TERRA

Introduction

Context

Software that is used to control machines and robots must be really predictable and reliable, as these *Cyber-Physical Systems* are *Safety-Critical*. Model-Driven Design (MDD) and Model-Driven Engineering (MDE) are nowadays inevitable to comply with both the technical and business needs. Designers in engineering domains usually have graphical design tools, in which the designs are formulated as labelled and directed graphs. Quality control and automatic consistency checking are crucial here, to support an effective design process. When these design graphs/models can be formulated in a formal language, the quality and consistency issues can be rigorously checked. The CSP process algebra [1] can be used for this, provided CSP models can be specified, shown and edited.

[1]Corresponding Author: *Maarten Bezemer, Robotics and Mechatronics, Faculty EEMCS, University of Twente, P.O. Box 217 7500 AE Enschede, The Netherlands*. E-mail: M.M.Bezemer@utwente.nl.

For sound reasoning about and working with domain models, the collection of domain concepts must be known. Meta-models, as abstraction to models, are able to highlight the properties of the model itself. However, the CSP process algebra is not defined by a meta-model but rather by using a context-free grammar (cf. [1]). Creating a formal CSP meta-model will elucidate CSP semantics and open-up many more possibilities, like model-to-model transformations from UML diagrams to CSP.

The construction and editing of CSP models needs to be supported by a domain specific editor for CSP. This will eliminate syntax errors and trivial contextual flaws, like unconnected rendezvous channels. Unfortunately, the existing gCSP tool [2] has not the quality we need and is not based on an explicit meta-model. The entire model definition lives mostly in the developers minds. An added benefit of having a meta-model is, that it is relatively easy to construct a domain specific (CSP) editor in an editor-generator framework like the Eclipse Model Framework (EMF) [3].

To alleviate the problems mentioned above, an explicit CSP meta-model has been developed. The new CSP tool suite presented in this paper, called Twente Embedded Real-time Robotic Application (TERRA), contains among others a CSP editor, which uses the CSP meta-model.

The ultimate goal for MDE is to create designs that are first-time right, i.e. have the model(s) verified at multiple stages in the design process, implementation (code) correct by (model) specification and satisfying all requirements targeted by the design. Based on the meta-model, shown in Figure 1, interfaces between tools can be designed, e.g. for (co-)simulation. Model-to-model (M2M) transformations can be used to optimise CSP models. From the CSP model, via model-to-text (M2T) transformations, TERRA is able to generate C++ code for the LUNA framework [4]. This is a hard real-time, multi-threaded, CSP-capable execution framework designed for embedded control software. Although the LUNA CSP execution engine was created before the CSP meta-model was created, LUNA does conform to the CSP meta-model. Thus, CSP models can be executed on embedded computers and control actual hardware, as was specified in the design.

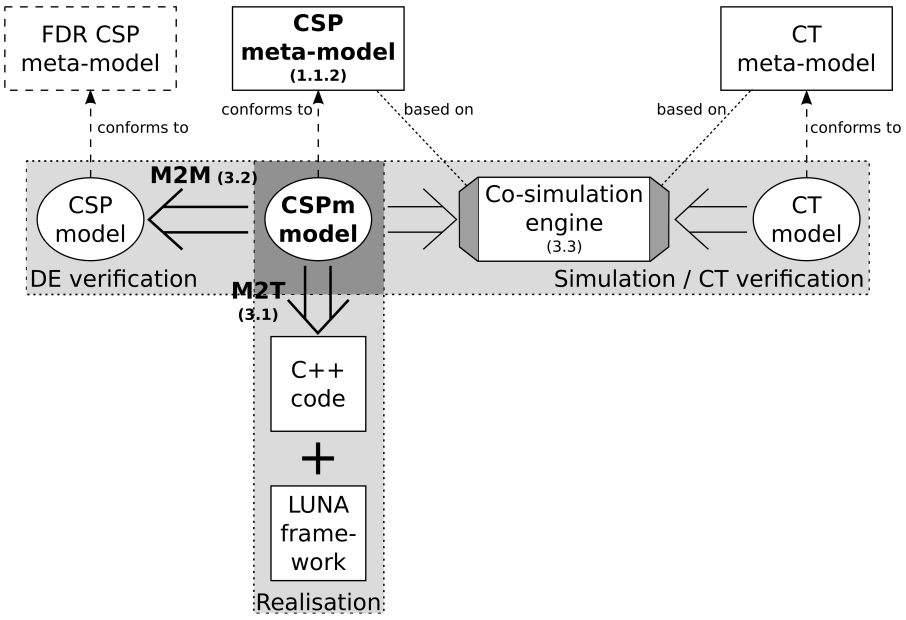

Figure 1. Meta-model usage. The bold items are newly designed and presented in this paper. (The numbers between the parenthesis refer to the corresponding sections).

Related Work

Meta-Models

For describing software designs, UML diagrams [5] are the de-facto standard. However, the standards are purely declarative and do not provide formal semantics. Formalising UML diagrams with CSP (i.e. adding formal semantics in general) is desired, such that livelock and termination checks can be done, to guarantee the quality of service of the components. In [6] a case study on model-to-model transformation from UML activity diagrams to CSP is discussed, whereby multiple transformation solutions and tools are evaluated. The authors divide the solutions in three categories: pure graph transformations, solutions with control structures and solutions based on a host framework / language. The presented (target) CSP meta-model resembles an Abstract Syntax Tree (AST) [7] for CSP grammar, where programming language concepts (e.g. *ProcessAssignment*) are mixed with CSP concepts. Since the meta-model is more concerned with storing a CSP document than modelling the process composition and communication, this meta-model is considered not suitable for the use cases presented in this paper. In [8] a rule-based model transformation solution with control is discussed, whereby the transformation from UML state charts to CSP is taken as a case study. However, no explicit CSP meta-model is given.

In the BRICS project work is ongoing to define a new generalised component meta-model, called the BCM [9]. However, in the BCM concurrency cannot be expressed as explicitly as in CSP (e.g. sequential execution cannot be specified). At the basis the meta-model uses the well-known Component-Port-Connector (CPC) meta-model (among others described in [10]), whereby components exchange information through ports over connectors (i.e. channels). The CSP meta-model, introduced in this paper, also has a CPC meta-model at its basis, as will be shown in Section 1.1.1.

In the DESTECS project, work is being done to define a Structural Operational Semantics (SOS) of co-simulation of discrete-event (DE) controller schemes and continuous-time (CT) behaviour of the machine to be controlled [11]. An SOS description consists of type definitions describing the static structure and transition relations for the behaviour of the model [12]. The DESTECS SOS description is not a meta-model, but it serves more or less as a meta-model, because a precise definition of the model elements comprising a co-simulation is given. Models here also use the CPC approach, mentioned above.

Tools

Many (supporting) tools are available to design and execute CSP models. The Failure Divergence Refinement (FDR) tool [13] can be used to check CSP models, among others on livelock and deadlock conditions. The gCSP tool [2] can be used to design CSP models. The C++CSP2 [14] and the LUNA [4] software frameworks implement CSP model execution in C++. However, none of these tools and frameworks provide an explicit meta-model, which is required for model validations and model transformations. Furthermore, since the meta-models are not available, the exact implementation of the tools lives mostly in mind's of the developers and thereby limits their extensibility and transparency.

Ptolemy II [15] is a heterogeneous modelling and simulation tool that allows to create multi-domain models using different models of computation (e.g. Finite State Machines or CSP), consisting of actors and directors. Actors are comparable to submodels or processes. The director determines the domain and the model of computation that is used by the simulator for executing an actor. The interaction between actors with different models of computation is based on well defined interfaces and flow of control rather than model-to-model transformations.

Outline

The first section discusses the design and implementation of the CSP meta-model. Next, the construction process for models that conform to the meta-model is described in Section 2. Followed by use cases for these models, showing situations where the (meta-)models provide additional advantages. Section 4 presents a concrete example of a hypothetical application created with TERRA. Conclusions are given in Section 5. The paper finishes with future work.

1. CSP Meta-Model

A meta-model in general consists of elements, as shown in Figure 2, to capture domain concepts (model objects). Each element might have one or more attributes and/or operations. Attributes hold data associated with the element, like a string to store the name of the object, or references to store a relation between this object and the referenced object. Element operations provide additional functionalities or ways to use the element. Each element, attribute and operation has a set of properties to refine the element, attribute or operation. These properties include a name, type, default value, and so on.

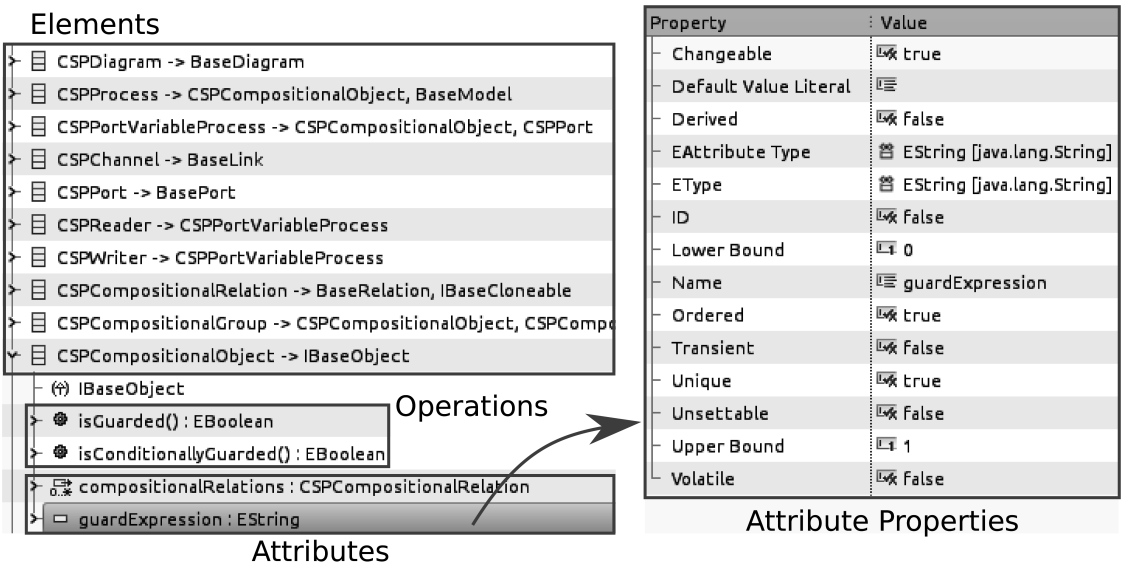

Figure 2. Meta-model terminology (applied on the CSP meta-model).

The meta-model presentation in Eclipse/EMF is similar to the UML class diagram presentation. Both methods can be used to create an object oriented ontology to capture the domain concepts. Concepts like the described elements, attributes and operations can be found in the UML class diagrams, however they are named classes, variables and methods respectively.

1.1. Design Choices

The CSP meta-model employs a modular design, as different use cases require different details. The meta-model can be extended to make it suitable for a variety of different use cases, ranging from designs of practical robotic setups to mathematical models with process algebra. For example, a design for a robotic setup requires information of the target hardware, while a mathematical model does not require such information. The following sections describe the modular parts of the CSP meta-model.

1.1.1. Base Meta-Model

The base meta-model captures the common factors used in component based designs. It provides means to design CPC (data-flow) like models, as it provides elements like component (BaseObject), port (BasePort) and connection (BaseRelation) objects, as shown in Figure 3.

The base meta-model objects are also shown on the left side in Figure 4, making it clear how the CSP meta-model is extending the base meta-model. All meta-models use the (absolute) base meta-model, directly or indirectly via other meta-models and more domain specific features to the base elements. In BRICS the CPC (base) meta-model is presented as another layer of abstraction, i.e. the meta-meta-model [9]. The inheritance relation between the meta-models, as proposed in this paper, is stronger than the regular 'conforms to' relation between meta-model and its meta-meta-model.

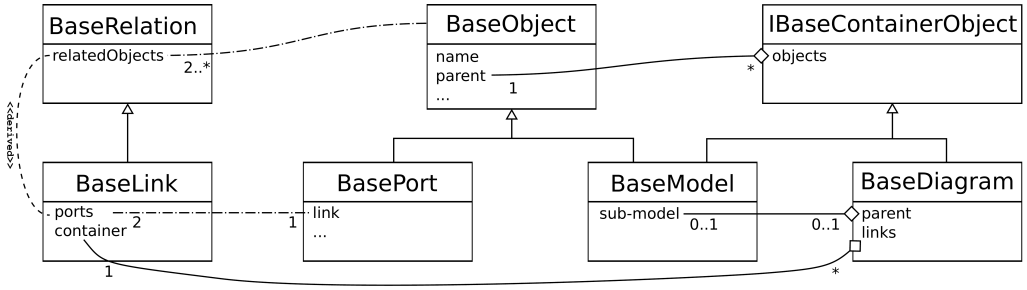

Figure 3. Partial base meta-model, showing the interesting parts of the meta-model.

The BaseObject element stores very basic details, like its name. Most other objects extend this BaseObject element and add more specific details. The same goes for relations, the BaseRelation defines that two or more objects are related. The actual type of relation that is shared between the objects, is defined by elements that extend the relation element. For example, the BaseLink element defines a data exchanging relation between objects.

For creating hierarchical models, multiple nested levels of objects need to be supported, i.e. each sub-process contains its own processes. For this purpose the meta-model provides a specialised container element called IBaseContainerObject. This element is used by other elements to provide support for containing other objects. For example, the BaseDiagram element uses the IBaseContainerObject element to contain all of its model objects, ports and links.

1.1.2. CSP Meta-Model

The CSP meta-model extends the base meta-model and provides CSP domain specific elements, like readers, writers, channels and object compositions, see Figure 4. These elements detail an explicit meta-model for Hoare's [1] context-free grammar CSP definition.

A CSPChannel is a BaseLink defining a rendezvous or buffered channel. The additional attribute provides the means to optionally specify the channel buffer size, used when the channel is a buffered channel. The same can be applied to readers and writers: they require a port attribute, so it is possible to connect a CSP channel to these objects. An intermediate element is added for shared functionality of the reader and writer elements, because both require a port and are also a BaseObject.

The main addition of the CSP meta-model to the base meta-model is the facility to express CSP algebra. Therefore, the CSP meta-model provides compositional meta-model elements, which are divided into a CompositionalObject, a CompositionalRelation and a CompositionalGroup element. The CompositionalObject (extended BaseObject) is used to specify that the object has a compositional relation to another CompositionalObject. CompositionalRelations are extending BaseRelations and are used to add details of this compositional

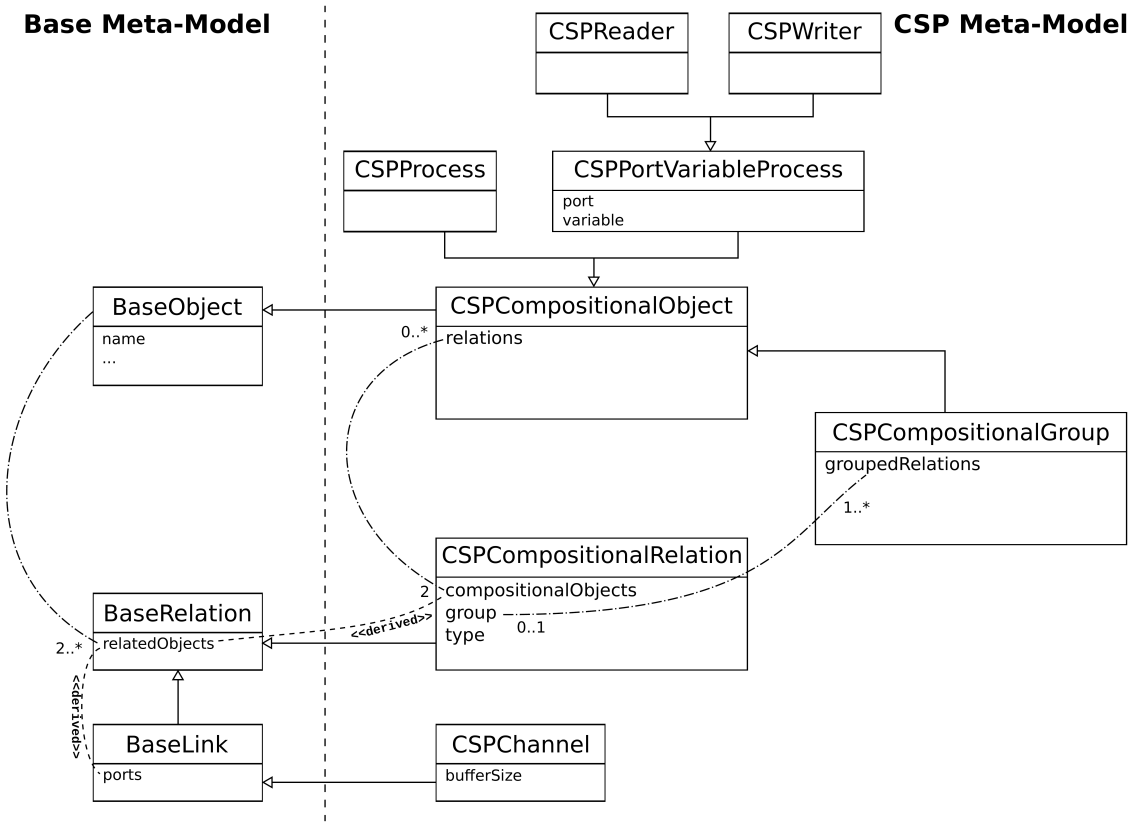

Figure 4. Partial CSP meta-model, showing the relation with the base meta-model objects and the CSP compositional objects.

relation, for example a type attribute defines whether it is a parallel, sequential, alternative or another relational type. The CompositionalGroups are used to group the relation objects, with the same type, to make the actual groups. They are extending CompositionalObject, so it is possible to hierarchically define compositional relations between compositional groups and objects as well.

Table 1. Representation of CSP constructs with the meta-model definition.

CSPm	meta-model representation	additional attribute(s)
`p = ...`	CSPProcess	
	CSPCompositionalGroup	*groupedRelations*: relations that are grouped
`channel c`	CSPChannel	*ports*: two connected ports/processes
`datatype <type>`	CSPVariableDescription	*name*: name of the variable
`= <name>`		*type*: boolean for `Bool`, integer for `Int`, ...
`c !<variable>`	CSPWriter	*variable* contains data to write on the channel
		link: channel to write to
`c ?<variable>`	CSPReader	*variable* contains data read from the channel
		link: channel to read from
`p ; q`	CSPCompositionalRelation	*type* = SEQUENTIAL
		compositionalObjects: the two related objects
`p || q`	CSPCompositionalRelation	*type* = PARALLEL
		compositionalObjects: the two related objects
`p [] q`	CSPCompositionalRelation	*type* = ALTERNATIVE
		compositionalObjects: the two related objects
`if`-statement	CSPRecursionProperty	*expression*: true when another loop is required

Table 1 shows the relation between CSP definitions and their representations using the CSP meta-model. Most of the CSP meta-model representations are shown in Figure 4 as well.

CSPProcess and CSPCompositionalGroup are both used to define a process. Their difference is that CSPProcess is used to define a process from a sub-model and CSPCompositionalGroup is used to define a process from a series of processes that have the same compositional relation type.

Processes that are part of an alternative compositional relation, require guard constructs. These are provided by a *guardExpression* that is contained in the process object, this expression must have a boolean result. The *guardExpression* is optional for the CSPReader or CSPWriter objects, if it is not provided a channel guard implementation is used.

The meta-model supports recursions called CSPRecursionProperty. Recursions are implemented as properties, so they can be attached to other model objects. Depending on the `<expression>` it loops over the `<process>`. The CSP representation would be similar to this:

```
p = if (<expression>) then <process> ; p else SKIP
```

The CSP meta-model does support all regular CSP constructs required to define processes and the communication between them. More modern CSP extensions, like mobile channels [16], are currently not supported by the meta-model. If the need rises, the meta-model can be extended easily to support such an additional feature.

1.1.3. Other Meta-Models

The CSP meta-model, described in the previous section, is designed according to Hoare's CSP definition. If an application requires additional elements the CSP meta-model can be extended. For example, to add notion of mobile channels, the port and channel elements should be extended to store additional information to support this. Extending the CSP meta-model can be done either directly or by designing a new meta-model that inherits the original CSP meta-model. Depending on the additional requirements and the nature of changes one of these possibilities can be chosen.

A pure CSP model is only able to communicate data between processes, the robotic system example requires, among others, mathematical calculations and interfacing with hardware. The required executable code is implemented in a programming language, which can be modelled as another meta-model element. By extending the compositional object element of the CSP meta-model, an additional attribute could be added containing the code that is executed when it is activated.

1.2. Model validation

A model is restricted by the meta-model definition, basically alien concepts are not representable in the meta-model. Furthermore, additional checks are needed to verify that the modelled reciprocal elements are correct and valid attribute values are used. For example all objects require a valid name to identify themselves. These names need to be unique, otherwise it is ambiguous which object is meant by a name.

Depending on the purpose of the (meta-)model, additional validation rules are required. These additional validation rules are not (strictly) defined by the meta-model, as they need additional knowledge of the modelled system and its use cases. Therefore, model constructing tools require additional validation rules, especially if a user is involved in the construction process, as the user might not be aware of all these additional rules.

The earlier example indicates that the names, besides needing to be unique, also need to conform to additional rules in order to be valid. The target language definition, aimed

at by the code generation, defines naming requirements from which validation rules can be inferred.

1.3. Implementation

The meta-model is implemented using the Eclipse Model Framework (EMF)[3]. This is a framework that provides means to design a custom meta-model. It uses a meta-model itself, called Ecore, to model that custom meta-model. The actual (Java) implementation is obtained through round-trip code generation from the modelled custom meta-model. The framework also provides additional services that are required for the implemented meta-models, like notification on model changes or model (de-)serialisation to load/store models.

Model validation and code generation are using the Eclipse Epsilon framework [17]. It provides a new language called Epsilon Object Language (EOL) which use used to specify the Epsilon Validation Language (EVL) and the Epsilon Generation Language (EGL) and several more. The Epsilon framework provides means to used these languages for validation and code generation purposes.

2. Constructing Models

There are multiple methods of constructing models using the meta-model information. The most obvious method is by an editor. The user is able to (graphically) design a model, using the meta-model elements provided by the editor. TERRA is such an editor for the CSP meta-model, the rest of this section gives more details of its design.

Another model construction method is model-to-model transformation: A source model is used to construct the resulting model. Whether the result conforms to the same meta-model as the source model, depends on the type and reason of the transformation. Section 3.2 provides more details on model-to-model transformation.

Text-to-model transformation is another way to construct models, it could be used to reverse-engineer a model to make its structure more clear. For example, C++ code could be converted to a CSP model to understand concurrent architecture of the code. As code files do not contain all required information that (graphical) models require, like coordinates and object dimension, this should be added by the transformation algorithm or the user. This form of model construction is not seen often, probably due to complexity issues of missing model data and the lack of functional use.

2.1. TERRA

Currently, TERRA is the only tool (or editor) to construct and design CSP models, that conform to the described CSP meta-model. TERRA is basically a collection of Eclipse plugins, shown by Figure 5.

The figure is a snap-shot of the current version of TERRA, new plugins are added when additional functionalities/services are added. The dashed lines separate grouped collections of plugins, the bold text indicates the specific functionality of each group. Most plugins are dependent on one of more other plugins, shown by the arrows in the figure. Dependencies on non-TERRA plugins are left-out to prevent the figure getting overly complex.

Each plugin within a group contributes to the group's overall functionality, like a meta-model, an editor, a specific type of model transformation/code generation, model validation and so on. All plugins together result in a complete CSP model development environment, providing support from constructing the model to using it for various purposes.

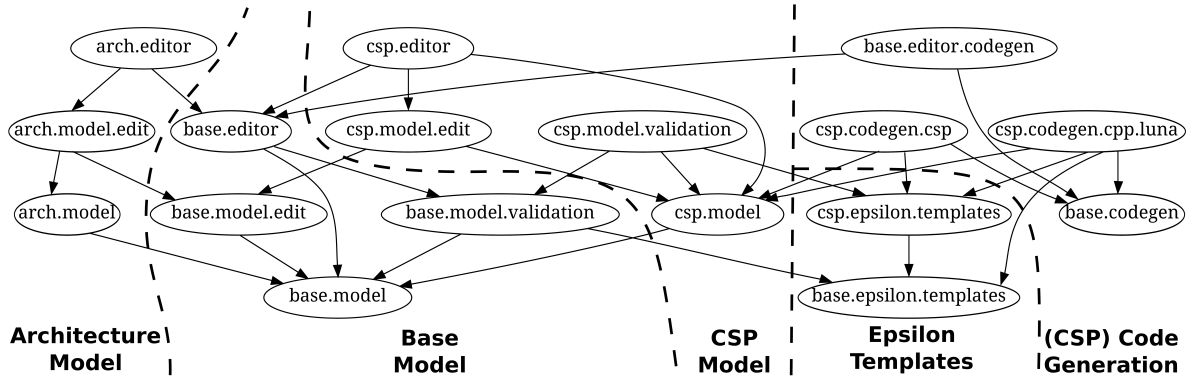

Figure 5. Overview of all TERRA plugins and their dependencies (the figure is generated using the source code of TERRA, as it was on the moment of writing).

2.1.1. The Graphical Eclipse Framework

A graphical user interface is preferred to be able to efficiently develop CSP models, as a graphical representation of a model result in a clearer overview of the design. The Graphical Eclipse Framework (GEF) [18] is used for this purpose. GEF provides means to add a graphical user interface that is nicely integrated with EMF. As described in Section 1.3, EMF is used to handle the meta-model details. The CSP editor exactly uses this EMF functionality to access and update the model data.

Each EMF model element has a GEF counter-part. These so called edit-parts provide the editor with a figure that graphically represents the model element. The edit-part also provides commands to modify the corresponding model element, and handles these commands.

2.1.2. Modular Tools and Editors

As mentioned earlier and shown in Figure 5, TERRA is a modular collection of tools and editors. Besides the CSP meta-model, TERRA includes also an architecture meta-model, which is also based on the base meta-model. Together with its accompanying editor, it provides means to model the system on a higher, architecture level.

From an architecture point of view, complete system parts, like a rotation belt or a robotic feeder arm, can be represented by a single component in the architecture model. These components have generic ports (CPC) that can be connected to each other using channels to communicate data from one component to another. For example the belt might need to communicate with the feeder arm to check whether it ready to picket a new object from the belt.

The details of the components can be hierarchically modelled using sub-models. These sub-models can be of any (supported) meta-model type, therefore the CSP models can be used for this. It is also possible to use another architecture sub-model to further specify the architecture of a component. For example, a feeder arm consists of a motor and several sensors that all could be considered as complete system parts. The level of detail is up to the designer.

Integration of TERRA with other tools extends the capabilities of TERRA with the advantages of these external tools. For example, in 20-sim [19] it is possible to model (parts of) physical systems and design their loop controllers. Since the model of computation used by 20-sim is data-flow, its concepts match with the CPC concepts of the base meta-model. Therefore a 20-sim interface meta-model could extend the base meta-model and thereby seamlessly integrating 20-sim models into TERRA.

Due to the modular design of TERRA, is it also possible to add (new) ways of model-to-model or model-to-text transformations. Each transformation type can be tailored for a specific target system or use case. Currently, it is possible to perform model-to-text transfor-

mations for FDR and C++/LUNA. More information on these transformations is provided in Section 3.1.

3. Use Cases

The previous section described ways to construct models conforming to the meta-model definition. This section explains ways to make use of these constructed models, first model-to-text transformation is described, also known as code generation. Next, model-to-model transformations are discussed. They differ from model-to-text transformations, as their result can still be used by the modelling tools. Simulation and co-simulation are discussed last.

3.1. Model-to-Text Transformation

In MDE it is common to use model-to-text transformations for generating the implementation of the model. It transforms the model into a (plain) text file, for example source code that can be compiled into an executable application. After a model is correctly constructed, the model-to-text transformation results in executable code that behaves as specified by the model. Using model-to-text transformations, TERRA is able to generate C++ code that is compatible with LUNA [4]. This (compiled) code is used to execute the model on an embedded target.

Shorter development cycles can be obtained by using model-to-text transformations making it easier to fix a problem in the application. Updating the model and just regenerate the code is much faster than diving into the source code and finding the problem. Each time when the model is modified, other development processes, e.g. model validation, can be (automatically) repeated. Using validation in an automatic way is called live-validation and helps the developer greatly in preventing problems with the model.

Formal verification of the model might be required to guarantee the quality of the executable application. When the model is formally verified, the resulting application code is also formally correct by specification. Formally checking the model could be done in the construction tool or editor, but it is also possible to use an external tool for this. For example, FDR [13] is able to formally verify CSP models. Because FDR cannot read the TERRA CSP models directly, model-to-text transformations are used to convert the TERRA CSP model into an FDR readable file.

In an ideal situation code generation of a formal verified model results in a first-time right implementation. Unfortunately this is not always the case. The CSP meta-model provides means of adding user C++ code, in order to be able to do actually something with the generated code, besides transporting data around. The custom C++ code cannot be checked by FDR and thus the code generation result might still be formally incorrect even though FDR did not find any problems with the model.

3.2. Model-to-Model Transformation

In contrast to model-to-text transformation, model-to-model transformation results in an usable model afterwards. The resulting model should conform to the same or a different meta-model, depending on the purpose of the transformation. A transformation conforming to a different meta-model can be used to export the model from one tool to another.

Transformations to the same meta-model are interesting to change the model from one point of view to another. For example, the designer, while constructing a model according to his own way of working, includes lots of sub-models to create a clear hierarchical model. When model-to-text transformation is used to generated an executable application, this hierarchical construction is not required. The target will not benefit from a nicely formatted hierarchical view, on the contrary it probably results in additional overhead and resource usage.

Model-to-model transformation can be used to convert the hierarchical model into a flattened model. Both models are still equal from a CSP point of view. More information on this topic and possible techniques are described in [20].

Furthermore, model-to-model transformations are also interesting for removing unnecessary model parts based on their configuration. For example, take a composition of multiple Generic Architecture Components (GACs). The GAC is a template object/model for robotic control applications and are typically used on an architectural model level. Since each GAC is designed to handle a broad spectrum of configurations, in order to make them re-usable, they can benefit greatly from optimisations based on their used configuration. Besides other features, a GAC could contain error handling support on a local and global level. When a GACs only used the local error handling, the model-to-model optimisation is able to remove the global error handling support and vice versa.

Both optimisation cases can serve as input to a model-to-text transformation to generate the optimised code without limiting the way of working of the designer.

3.3. (Co-)Simulation

To gain insight in models, simulations are commonly used. These simulations can be categorised into simulations of one or more different domain models. For DE simulations of CSP models, the order of active (and blocked) processes is determined from the CSP algebra (included in the meta-model). Running a CSP model has helped new model designers (students) to understand the CSP semantics of their designed model.

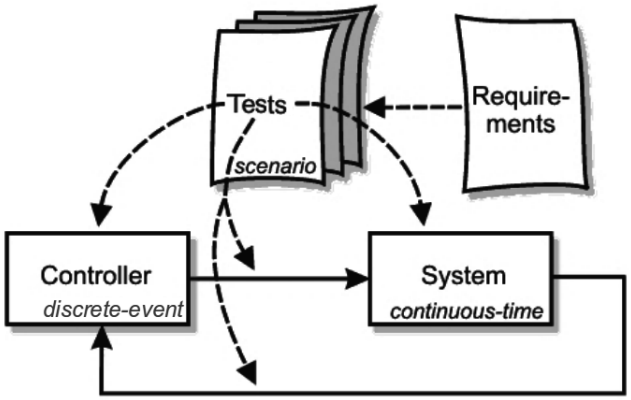

Figure 6. Co-simulation with scenario based testing [21].

The simulation of multiple different types of domain models is commonly known as co-simulation. For instance, a CT dynamic model of the proposed hardware is simulated together with the DE software simulation, as shown in Figure 6. This will increase confidence in a CSP model, since the dynamic model of the system will create external stimuli based on the modelled physical behaviour of the proposed setup. Extending the simulations with scenario-based tests, based on system requirements, will create more reliable software, as was experienced in the DESTECS project [21,22].

The design of both types of simulation engines should be based on their respective meta-models, as depicted in Figure 1. For co-simulation, the external stimuli can be generated by using external models provided external tools (Section 2.1.2) that are capable of running stepwise simulations. When an object is activated by the TERRA simulation engine, that contains an external models capable of this, ports can be used to first communicate required data to the external simulation tool and after a simulation step, grab the results and send it back to the TERRA simulation engine. Depending on its current use, a top-level port in TERRA can easily be (re)connected to either actual hardware (drivers) or a simulation engine.

4. Example

This section shows an example application to demonstrate that TERRA is fully functional. It uses all CSP constructs of Table 1, designed using the TERRA CSP editor. Real applications for some for our laboratory setups, like JIWY [4] and the ProductionCell [23], have also been developed using TERRA. Some of our students are also successfully using TERRA for their assignments.

Most of the symbols of Figure 7 are similar compared with the ones used by gCSP [2]. Basically, the example application does some calculations in the 'Calculations1' process. It defines following variables: *pickFirst*, *firstValue*, *secondValue* and *readValue*. Depending on the calculations the value of the variable *firstValue* or *secondValue* is communicated to 'THREAD2' were it is used to perform some other calculations. The boolean *pickFirst* variable is set by 'Calculations1' to determine which reader from the 'READERS' group should be used to read the correct value. The dashed line of the readers shows that they are conditional/expression guarded readers. Note that in real-life applications the required values for the calculations and their results are communicated with the (embedded) set-up, however this is left-out to keep the example simple.

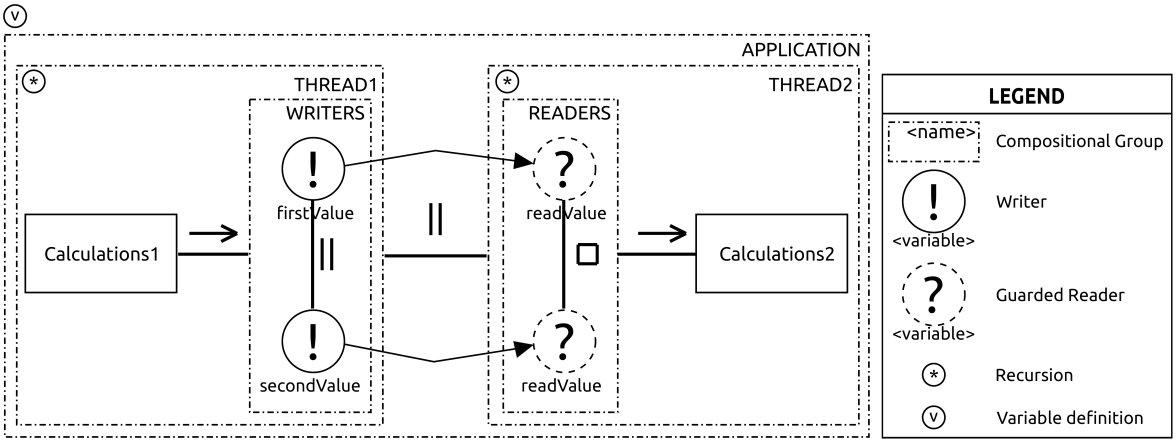

Figure 7. Model of example application designed with TERRA.

A CSP implementation of this application is shown by Listing 1. At the moment of writing, both the channel guards and conditional guards are not yet supported by the 'CSPm to CSP code generator', so the listing shows the result as if the guards would have been supported.

The recursion property of 'THREAD1' and 'THREAD2' is expressed by

```
if(<boolean expression>) (<statement>) ; <original process> else SKIP
```

CSPm code generation could be improved in the future to optimise this piece of code. The `boolean expression` is always `true` in this case, so the `if`-statement is redundant and should be left out in such cases.

After the model is formally checked for correctness, the 'LUNA/C++ code generator' can be used to generate C++ code using the LUNA framework. This code can be compiled in order to let the (control) application run on an embedded target. The calculation blocks can be filled in using C++ code blobs that perform the required (control) software calculations. For robotic targets the control software could be generated by external tools, like 20-sim. In future versions of TERRA, models from these external tools can be added to the TERRA CSP model directly, so the manual integration is not required anymore.

The generated C++ code is fully object oriented. The (full) resulting C++ code listings are not provided here as they would be extensive and take too much space. Basically, the

```
datatype Bool = pickFirst
datatype Int = firstValue, secondValue, readValue

channel FW_to_FR   -- FirstWriter  to FirstReader  channel
channel SW_to_SR   -- SecondWriter to SecondReader channel

APPLICATION = THREAD1 [| {| FW_to_FR , SW_to_SR |} |] THREAD2
THREAD1 = if (true) then (Calculations1 ; WRITERS ) ; THREAD1
                                                       else SKIP
THREAD2 = if (true) then (READERS ; Calculations2 ) ; THREAD2
                                                       else SKIP
WRITERS = FirstWriter ||| SecondWriter
READERS = FirstReader [] SecondReader
FirstWriter  = FW_to_FR!firstValue -> SKIP
SecondWriter = SW_to_SR!secondValue -> SKIP
FirstReader  = pickFirst & FW_to_FR!readValue -> SKIP
SecondReader = not pickFirst & SW_to_SR!readValue -> SKIP
```

Listing 1. Partial CSP of the design of Figure 7 (mainly) generated by the 'CSPm to CSP code generator'

generated C++ code first initialises all channels that are defined by the object that is being generated. Next all sub-objects that are contained by an object are generated. The generated channels are attached to reader, writers and sub-models that have ports. After that all generated objects are grouped by their compositional groups. These created groups are grouped again if required, for example the 'THREAD1' and 'THEAD2' groups are grouped together into the 'APPLICATION' group. The top-most diagram is instantiated and started up from the generated `main` function to make the model executable.

The actual code generation requires a logical sequence of steps based on (construction) dependencies introduced by LUNA. However, these details are considered out of the scope of this text.

This example section showed that TERRA is suitable to design models, its CSPm code generation is usable to formal check TERRA models and the LUNA/C++ code generation allows the model to be executed on an embedded target. When the required inclusion of the generation of the guards is finished, even more complex models can be properly checked, without manually adding the missing code.

5. Conclusions

The presented CSP meta-model is suitable to design CSP models that conform to Hoare's CSP definition. Section 1 describes how the CSP meta-model is derived using a modular approach by extending the base meta-model. The CSP meta-model has all kinds of use cases, as described in Section 3. For example, model-to-text transformations are used to formally verify the CSP model with FDR or to generate code that can be executed on an embedded target.

TERRA is an integrated collection of tools to support the MDD/MDE way of working. The user is able to graphically construct a CSP model that conforms to the CSP meta-model. Model checking on livelocks and deadlocks conditions is supported by using FDR. When satisfied, the CSP model can be transformed into LUNA based code using model-to-text transformations.

The modular nature of the base and CSP meta-models makes it possible to support additional requirements. Therefore, the CSP meta-model is suitable as a standard for all kinds of CSP modelling related work. It is recommended to make use of this meta-model to stan-

dardise modelling within the CSP community. Hopefully a standard meta-model will emerge that is suitable for the needs of the community and helping to improve interaction between multiple disciplinaries within the community.

6. Future work

TERRA needs to be extended to be able to make use of external models (see also Section 2.1.2). TERRA focuses on process (communication) flow modelling, where other tools/models focus on other specific areas. Being able to make use of their expertise is better than re-inventing the wheel. Therefore to support an external tool, a meta-model needs to be added to TERRA to create model interfaces that can be used by the existing TERRA editors.

Robotic systems, one of the target modelling uses of TERRA, consist often of similar components, for example to drive motors or read sensors. Having a library of building blocks (Section 3.2) and patterns decreases development time and makes the software more reliable. For efficient re-use of these building blocks and patterns should be parametrisable. TERRA needs to be extended to present the developer with these parameters when such a building block or pattern is used in a model.

For example, a composition of GACs will most likely not result in optimal executable code. This problem can be tackled to add model-to-model transformations to TERRA for optimisations [20]. This optimised model can be used for the model-to-text transformation.

For educational purposes and testing, a CSP simulator is going to be added to TERRA, as described in Section 3.3. Integration with external tools will also be included for co-simulation purposes and thus adding the possibility to combine DE and CT simulations. This will make the simulations more realistic and thereby helping the developer to create a first-time right application.

The simulator can also be used to playback a real-life situation using the application logs. Instead of the CSP algebra to determine the order of active processes, the log data combined with sensory readings will be used.

Acknowledgements

The research leading to these results has received funding from the European Community's Seventh Framework Programme (FP7/2007-2013) under grant agreement no. 231940 (project BRICS) and no. 248134 (project DESTECS). This research has been carried out as part of the Dutch PIDON project TeleFLEX.

References

[1] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice-Hall, London, 1985.

[2] D.S. Jovanović, B. Orlic, G.K. Liet, and J.F. Broenink. gCSP: A Graphical Tool for Designing CSP Systems. In I. East, J. Martin, P.H. Welch, D. Duce, and M. Green, editors, *Communicating Process Architectures 2004*, volume 62, pages 233–252, Amsterdam, September 2004. IOS press.

[3] D. Steinberg, F. Budinsky, M. Paternostro, and E. Merks. *EMF: Eclipse Modeling Framework 2.0*. The Eclipse Series. Addison-Wesley Professional, 2nd edition, 2009.

[4] M.M. Bezemer, R.J.W. Wilterdink, and J.F. Broenink. LUNA: Hard Real-Time, Multi-Threaded, CSP-Capable Execution Framework. In P.H. Welch, A.T. Sampson, J.S. Pedersen, J.M. Kerridge, J.F. Broenink, and F.R.M. Barnes, editors, *Communicating Process Architectures 2011, Limmerick*, volume 68 of *Concurrent System Engineering Series*, pages 157–175, Amsterdam, November 2011. IOS Press BV.

[5] Object Management Group (OMG). Unified Modeling Language (UML), 2006. `http://www.omg.org/spec/UML`.

[6] D. Varró, M. Asztalos, D. Bisztray, A. Boronat, D. Dang, R. Geiß, J. Greenyer, P. Van Gorp, O. Kniemeyer, A. Narayanan, E. Rencis, and E. Weinell. Transformation of UML Models to CSP: A Case Study for Graph Transformation Tools. In Andy Schrr, Manfred Nagl, and Albert Zndorf, editors, *Applications of Graph Transformations with Industrial Relevance*, volume 5088 of *Lecture Notes in Computer Science*, pages 540–565. Springer Berlin / Heidelberg, 2008.

[7] D. Watt, D. Brown, and R.W. Sebesta. *Programming Language Processors in Java: Compilers and Interpreters AND Concepts of Programming Languages*. Prentice Hall Press, Upper Saddle River, NJ, USA, 2007.

[8] J. Küster. Definition and validation of model transformations. *Software and Systems Modeling*, 5:233–259, 2006.

[9] M. Klotzbücher, P. Soetens, and H. Bruyninckx. Bcm: A minimal robotic component model for multitarget system and component generation. Technical report, Best Practice in Robotics, EU FP7 project, March 2010.

[10] Object Management Group (OMG). *Lightweight Corba Component Model (LCCM)*, chapter 13. 2006. `http://www.omg.org/spec/CCM`.

[11] K.G. Lausdahl, A. Ribeiro, P.M. Visser, F.N.J. Groen, Y. Ni, J.F. Broenink, A.H. Mader, J.W. Coleman, and P.G. Larsen. D3.3b — Co-simulation Foundations. Technical report, The DESTECS Project (INFSO-ICT-248134), December 2011.

[12] G.D. Plotkin. The origins of structural operational semantics. *Journal of Logic and Algebraic Programming*, 60–61:3–15, July–December 2004.

[13] Formal Systems (Europe) Limited. FDR2, 2008. `http://www.fsel.com/software.html`.

[14] N.C.C. Brown. C++CSP2: A Many-to-Many Threading Model for Multicore Architectures. In A.A. McEwan, W. Ifill, and P.H. Welch, editors, *Communicating Process Architectures 2007*, pages 183–205, July 2007.

[15] Ptolemy. Ptolemy II website, June 2012. `http://ptolemy.eecs.berkeley.edu/ptolemyII/index.htm`.

[16] P.H. Welch and F.R.M. Barnes. A CSP Model for Mobile Channels. In P.H. Welch, S. Stepney, F. Polack, F.R.M. Barnes, A.A. McEwan, G.S. Stiles, J.F. Broenink, and A.T. Sampson, editors, *Communicating Process Architectures 2008, York*, volume 66 of *Concurrent Systems Engineering Series*, pages 17–33. IOS Press BV, September 2008.

[17] D. Kolovos, L. Rose, and R. Paige. *The Epsilon Book*. 2012.

[18] D. Rubel, J. Wren, and E. Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. The Eclipse Series. Addison-Wesley Professional, 2011.

[19] Controllab Products. 20-sim - Graphical modeling and simulation tool, June 2012. `http://www.20-sim.com/`.

[20] M.M. Bezemer, M.A. Groothuis, and J.F. Broenink. Analysing gCSP Models Using Runtime and Model Analysis Algorithms. In P.H. Welch, H.W. Roebbers, J.F. Broenink, F.R.M. Barnes, C.G. Ritson, A.T. Sampson, D. Stiles, and B. Vinter, editors, *Communicating Process Architectures 2009, Eindhoven*, volume 67 of *Concurrent System Engineering Series*, pages 67–88, Amsterdam, November 2009. IOS Press BV.

[21] DESTECS. Design support and tooling for embedded control software. Website, 2010.

[22] K.G. Pierce, C.J. Gamble, Y. Ni, and J.F. Broenink. Collaborative modelling and co-simulation with destecs: A pilot study. In *3rd IEEE track on Collaborative Modelling and Simulation, in WETICE 2012*, pages 1–6. IEEE-CS, June 2012.

[23] M.A. Groothuis and J.F. Broenink. HW/SW Design Space Exploration on the Production Cell Setup. In P.H. Welch, H.W. Roebbers, J.F. Broenink, and F.R.M. Barnes, editors, *Communicating Process Architectures 2009, Eindhoven, The Netherlands*, volume 67 of *Concurrent Systems Engineering Series*, pages 387–402, Amsterdam, November 2009. IOS Press.