Automated Fault Tolerance Augmentation in Model-Driven Engineering for CPS

Tingting Hu[a,*], Ivan Cibrario Bertolotti[b], Nicolas Navet[a], Lionel Havet[c]

[a]*University of Luxembourg – Faculty of Science, Technology and Communication, 6 Avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg*
[b]*CNR – National Research Council of Italy, IEIIT, c.so Duca degli Abruzzi 24, I-10129 Torino, Italy*
[c]*RealTime-at-Work (RTaW), 4 Rue Piroux, 54000 Nancy, France*

Abstract

Cyber-Physical Systems are usually subject to dependability requirements such as safety and reliability constraints. Over the last 50 years, a body of efficient fault-tolerance mechanisms has been devised to handle faults occurring at run-time. However, properly implementing those mechanisms is a time-consuming task that requires a great deal of know-how. In this paper, we propose a general framework which allows system designers to decouple functional and non-functional concerns, and express non-functional properties at design time using domain-specific languages. In the spirit of generative programming, functional models are then automatically "augmented" with dependability mechanisms. Importantly, the real-time behavior of the initial models in terms of sampling times and meeting deadlines is preserved. The practicality of the approach is demonstrated with the automated implementation of one prominent software fault-tolerance pattern, namely N-Version Programming, in the CPAL model-driven engineering workflow.

Keywords: Cognification, Model-driven engineering, Fault-tolerance, Industrial cyber physical systems, Domain-Specific Languages

1. Motivation

Context of the paper. *Cognification* is the introduction of knowledge into a process to boost its efficiency and relevance [1]. Cognification has been at the heart of the ongoing digital revolution with the aim to replace (or for the least complement) intellectual labor, in a similar way as the industrial revolution was about replacing physical labor by machines and processes. Like pointed out in [1], cognification does not imply the use of artificial intelligence. It starts by applying the know-how gathered by human designers.

Model-Based Software Engineering (MBSE) is an enabling technique towards this direction. And it is steadily gaining acceptance in the design of Cyber-Physical Systems (CPS) [2, 3]. Using models as main artifacts in the design possesses key advantages like offering an always up-to-date specification of the system under design, the possibility to perform early-stage validation/verification, and to generate code from the models. All this helps increase both productivity and quality. For a long time, MBSE only dealt with functional concerns, i.e. what the system does. Instead, it overlooked non-functional requirements [4] like real-time, energy, security and dependability objectives, which have become increasingly essential in the design of CPS. This limitation of MBSE has been acknowledged and partially addressed, for instance, in the context of real-time control systems in [5, 6].

CPS such as automotive and Avionics systems need to keep delivering their functionality in the presence of *run-time* faults. Even though some faults can be avoided by following a proper MBSE design flow, or removed by including fault detection and fault removal mechanisms in the system, they can not be completely prevented. The adoption of fault tolerance mechanisms is a proactive quality assurance method, especially needed for safety-critical components of a CPS [7, 8, 9]. Fault tolerance is usually achieved by introducing redundancy in software or hardware. Thanks to the increasing use of code generators, the final code to be deployed on the target platform can often be automatically generated from validated models. This requires taking into account and addressing non-functional properties in the early design phase, at the model level.

Among the non-functional requirements, dependability, and especially safety and reliability, are identified as difficult ones to fulfill with MBSE [4]. Many works such as [10, 11] have been proposed to model and assess the dependability of CPS, in the context of MBSE. But few [12] have been targeted at enhancing the dependability at the design time by incorporating fault tolerance mechanisms, in particular in an automated manner. The Authors of [12] proposed a model-based design framework for fault-tolerant automation systems. But it mainly focused on code generation. Moreover, it didn't take into account software design faults, which need to be handled with

[*]Corresponding author. University of Luxembourg – Faculty of Science, Technology and Communication, 6 Avenue de la Fonte, L-4364 Esch-sur-Alzette, Luxembourg. Tel.: (+352) 46 66 44 5859, Fax: (+352) 46 66 44 35859.

Declarations of interest: none

Email addresses: tingting.hu@uni.lu (Tingting Hu), ivan.cibrario@ieiit.cnr.it (Ivan Cibrario Bertolotti), nicolas.navet@uni.lu (Nicolas Navet), lionel.havet@realtimeatwork.com (Lionel Havet)

more complex fault tolerance methods, such as N-Version Programming (NVP). In this work, we propose a general framework, which automates the "augmentation" of a model into a functionally equivalent model that meets additional dependability objectives.

Decoupling functional and non-functional concerns. This work relies on the principle that, especially in the field of critical systems, it is possible and beneficial in terms of system engineering to decouple functional and non-functional concerns. The idea is not novel, for instance interlocking systems currently used in the French railways have been designed in this manner two decades ago. First, a proof of functional correctness is obtained on a model [13]. Then, the execution platform is designed to meet the assumptions of the model, for instance, in terms of temporal behavior and how input/output channels operate. The functional behavior of the component will remain correct on any platforms meeting the model assumptions, thus easing re-use and portability. After functional correctness is validated, the possibility and capability to express non-functional concerns on top of the functional models are essential to integrate them in the design and evaluate their impact on functional correctness. Works strengthening the expressive power of modeling languages in this respect will be discussed in Section 2.

Descriptive versus prescriptive models. Both descriptive and prescriptive models are found in MBSE for CPS. In [14], the author discusses the fundamental difference between descriptive and prescriptive models. *Prescriptive* models are sufficiently precise to define the execution semantics of the models, that is, how exactly they will execute. For instance, a prescriptive model will set the scheduling parameters of a set of software components. *Descriptive* models, on the other hand, are meant to assist in the understanding, design and implementation of a system, in sharing always up-to-date information among stakeholders. But they do not have to state how models should execute. They are usually starting incomplete and are enriched along the development process.

Domain-specific modeling languages. As pointed out in [14], prescriptive models are not a universal solution, one reason being that they tend to impose design choices too early in the design phase and contain too much low-level information. We believe it is however possible to mitigate these limitations with the use of high-level *domain-specific modeling languages* (DSML) that abstract as much as possible implementation details, while preserving a complete semantics of execution. The latter property means that besides the validation of functional correctness, the designer can perform early-stage verification of non-functional properties. For instance, real-time execution properties can be verified by performing schedulability analysis on the model defining the execution semantics.

Cyber-Physical Action Languages. This study adopted a DSML for CPS, namely the Cyber-Physical Action Language (CPAL) [15]. It is the cornerstone of a MBSE design flow developed since 2012. It has been inspired by a variety of successful approaches for critical systems design such as Promela [16, 17] and synchronous languages [18]. And it has been built upon a few important underpinning principles shared with other modern DSLs for CPS like Mbdeddr [19] and MITA[1]. These languages feature high-level constructs (e.g., native support of automata), I/O abstractions and the possibility to rely on additional dedicated internal DSLs, such as the one for dependability mechanisms described in this work. The main use cases of CPAL in the industry have been in the design of critical embedded networks [20, 21], while it has been used in the academia for teaching and research purposes. Regarding the latter, the CPAL project is targeted at exploring the possibilities and addressing the limitations of DSMLs in the context of "cognifying" MBSE [1].

Contributions of the paper. Cyber-Physical Systems are usually subject to dependability requirements such as safety and reliability constraints. Over the last 50 years, a body of efficient fault-tolerance mechanisms has been devised to handle faults occurring at run-time. However, properly implementing those mechanisms is a time-consuming task that requires a great deal of know-how and it is usually error-prone. Our work aims at capturing the expert knowledge and integrating it into the MBSE design flow. In this paper, we propose an automated way to augment prescriptive models specified in DSML with fault-tolerance features, so as to enhance the dependability of the modeled CPS. More generally, this work explores the extent to which supporting non-functional properties can be achieved in an automated manner by means of *model transformation (MT)*. Specific attention is paid that the augmented model remains correct not only in the functional domain, but also with respect to the relevant set of non-functional properties (e.g., sampling times and deadlines). The proposed technique is illustrated with one of the prominent fault-tolerance patterns, namely N-Version Programming (NVP, see [22, 23]).

The plugin-based, model-to-model transformation framework developed for this study is available in open source[2] under the GNU AGPL-3.0 license. It can be freely reused by anyone, for instance to perform design-space exploration, possibly also adapting it to other modeling languages.

Outline of the paper. Section 2 provides an overview of the related work. Then, the fault-tolerance augmentation framework is described in terms of its software architecture and implementation choices in Section 3. The application of the framework to NVP is presented in Section 4, while Section 5 and 6 discuss the run-time behavior of the NVP-augmented model with respect to the original model, by means of two case studies. Finally, conclusions and perspectives are drawn in Section 7.

2. Related Work

Many effort such as [11, 24, 25] have been invested into strengthening the expressiveness of a modeling language regarding dependability, including fault tolerance concepts, with the goal to facilitate dependability analysis. The Authors of [11] defined fault-tolerance patterns with the Architecture

[1] `https://projects.eclipse.org/projects/iot.mita`

[2] `https://github.com/minimap-xl/nhc`

Analysis and Design Language (AADL) to assist dependability analysis at the architecture level. However, when constructing AADL models with dependability features, system designers need to instantiate a selected fault-tolerance pattern and customize it as needed. Instead, this paper proposes a general framework that automates this process by MT for CPAL models. Similarly, the Authors of [24] extend Simulink models to support the specification of common fault-tolerance design patterns, such as voting, comparison, and sparing, so that the extended models can tolerate hardware faults. However, the proposed approach still requires manual effort to extend the models. Work presented in [25] extends UML/Marte with a unified and complete dependability profile, which covers fault-tolerance concepts as well. However, as the others, no automation support is provided.

Directly related to this work is [26] that proposes a MT workflow to automate the verification of dependability properties from the CHESS modeling language that is based on UML/Marte. With respect to [26], this work is concerned with augmenting models with dependability features while retaining the ability to accomplish non-functional analyses on both the original and the transformed model, be it in a simulation environment or on the actual target. For example, both scheduling and code coverage analysis shall still be applicable to the transformed model, in order to properly assess MT suitability, overhead, and performance in a specific application scenario, as will be discussed in Sections 5.2 and 5.4, respectively. To this purpose, it is critical that MT operates within the boundaries of the CPAL language, for which the analysis tools were originally designed and on which they operate.

Sometimes, a specialized form of MT has been adopted to incorporate useful features in the design process, like aspect-oriented programming in a component-based system [27]. In other cases, MT has been seen as a way of translating a model from one language to another, but without augmenting it in any way [28]. Generally, the goal is to exploit the capability of other modeling languages or their toolkits to perform specialized tasks, for instance model checking [29]. Instead, a key point of the work presented here is its ability to *augment* a model with automatically generated code to meet user-specified dependability properties.

MT is indeed a key issue in MBSE that can be performed at different levels of abstraction. Typically MT is done on UML diagrams [30] but then without the executability, or at the opposite end of the spectrum, on the actual code [31] but then without the transformation being captured in the models. Likewise, the tool described in [32] applies MT to a model of a distributed system described by Petri nets to automatically generate the C code of an I^2C-based communication layer for it. However, the dissimilarity of the source and target languages prevents MT from being applied more than once in sequence and binds the tool itself to accomplish a very particular, ad-hoc goal. Instead, the framework proposed in this paper enables users to apply one or more fault-tolerance patterns to the same model multiple times, for instance to perform design space exploration.

Model and tool integration, as well as separation of concerns, are important CPS research topics that have been addressed in very recent related work. Among those, [33] describes an integrated design and simulation platform for evaluating the resilience of a transportation network to cyber-attacks. Similarly, [34] discusses OpenMETA, an experimental design automation tool suite in the automotive domain. Even though both works highlight the significance of integrated platforms, akin to the CPAL project, in rapid and effective CPS development. Neither of them includes model transformation (MT) features. In other words, they support design space exploration mainly through the *composition*, rather than the *transformation* or the *augmentation*, of existing models.

In the past decades, various MBSE frameworks have been proposed to facilitate the modeling, simulation, validation and code generation of CPS. Simulink and SCADE are the most successful ones among them. Simulink models are purely functional. To support the analysis of non-functional properties such as real-time, [35, 36] proposed different frameworks to extend Simulink models and enable the specification of schedulers, tasks, and messages. Instead, those are built-in concepts in CPAL. Even though MT is supported in Simulink via the control system toolbox, it is limited to the conversion among various control model types or between discrete and continuous time representations. Instead, the MT framework proposed in this paper can be applied to dependability augmentation of CPS models and beyond, thanks to its generality. Regarding dependability, work presented in [37] translates Simulink stateflow models into timed automata supported by the Uppaal verifier. It then relies on the verification capability of Uppaal to identify design faults missed by the Simulink Design Verifier and Simulink Polyspace. The verification outputs serve as reference to correct Simulink models. System dependability is improved by eliminating design faults at the *development* phase.

SCADE [38] also supports the MDD flow, in which graphical models are used to specify mathematically accurate control models and capture the data flow among various components. The final code to be executed on the target platform is generated from the models with the certified code generator KCG. It ensures that the generated code is compliant with most safety standards. Simulation in SCADE is based on the generated code, either in an emulated environment or on the target platforms. Instead, the CPAL project supports three modes: direct model simulation, model execution on the target platforms, as well as code generation and execution on the target platforms. Direct model simulation offers the opportunity to examine system behavior under diverse scenarios. SCADE is extending its support for multicore architectures [38], using an annotation-based approach, while modeling and timing simulation for multicore systems are already supported in CPAL. Most MT work related to SCADE is to transform models in other DSMLs (e.g. SysML, Simulink) to SCADE, or vice versa. Instead, this work focuses on in-place MT.

The Eclipse Modeling Framework (EMF) [39] is widely used in MBSE, thanks to its large ecosystem and tooling set, including those for model transformation. Models/meta-models are specified with the Ecore modeling language, which is compliant with the essential Meta-Object Facility (eMOF) specification

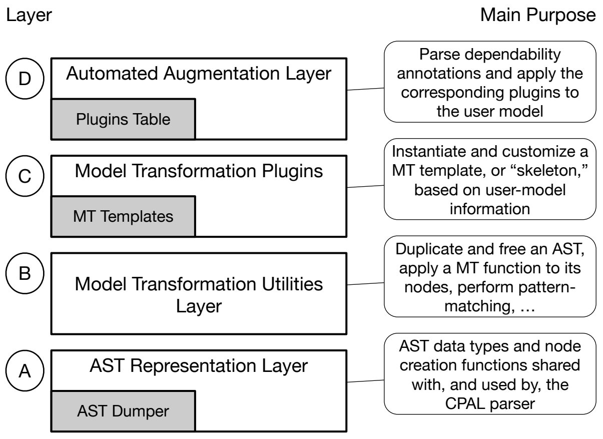

Layer
Main Purpose
D
Automated Augmentation Layer
Plugins Table
Parse dependability annotations and apply the corresponding plugins to the user model
C
Model Transformation Plugins
MT Templates
Instantiate and customize a MT template, or "skeleton," based on user-model information
B
Model Transformation Utilities Layer
Duplicate and free an AST, apply a MT function to its nodes, perform pattern-matching, ...
A
AST Representation Layer
AST Dumper
AST data types and node creation functions shared with, and used by, the CPAL parser

Figure 1: Layered structure of the MT framework.

standardized by OMG. The proposed framework is not built with EMF and MT tools from its ecosystem due to practical considerations. Indeed, otherwise, they need to be included as part of the proposed framework and delivered to end users. This may make the proposed framework unnecessarily fat. More importantly, this introduces extra dependencies of the MT process on one or more specific external tools, which limits the flexibility of the proposed framework and has the potential risk of making it vulnerable to tool updates. Hence, the proposed framework is implemented in C, based on a well-known intermediate representation, namely the abstract syntax tree (AST). On a side note, as discussed in [40, 41], the advantages of using a specialized MT language are becoming less obvious, with respect to general purpose languages.

3. Architecture design and Implementation

The MT framework proposed in this work has been designed according to a modular four-layer architecture depicted in Figure 1 and implemented as about 8000 lines of C code. Individual layers will be discussed in the following, in bottom-up order. Even though the framework will be applied and illustrated taking the NVP dependability pattern as an example in Sections 4–6, its internal structure is general and does not depend on assumptions specific to any particular pattern.

3.1. AST Representation

Model transformation is carried out by means of a CPAL-to-CPAL translator, sometimes called a *transpiler*. As for internal representation, the MT framework is based on a syntax-directed AST representation directly derived from the CPAL grammar, in which AST node types are in direct correspondence with grammar symbols. The AST representation layer (layer A in Figure 1) thus provides functions to create individual AST nodes and bind them together. This choice offers several prominent advantages:

a) it streamlines the development of MT plugins, through the template mechanism described in Section 3.3, because templates can also be written in CPAL;
b) it enables the immediate reuse of the existing `flex`- and `bison`-based CPAL lexer and parser;
c) it is amenable to a compact and efficient C-language implementation without introducing any undue assumption on which specific dependability patterns the tool is going to handle.

Not less importantly, the AST is a language-independent representation of the corresponding source model. For instance, an AST node may represent an abstract variable declaration independently from any concrete grammatical details of the declaration itself. In a similar way, the AST enables the MT framework code to manipulate relatively complex entities, like process definitions, regardless of the underlying modeling language.

Therefore, as long as one stays in the domain of procedural languages, whose execution semantics are close to that of CPAL, it is foreseen that the MT framework could be retargeted towards a different modeling language by modifying this layer only, while leaving the AST representation unaltered. As a consequence, the AST representation layer also shields the layers to be discussed next against modeling language dependencies.

Another important part of the AST representation layer is the *dumper*, a software module that emits the CPAL source corresponding to an AST as the last stage of MT. As a side benefit of working at the CPAL level, it was possible to enhance the human-readability of the code emitted by the dumper and simplify source code comparisons, without additional effort, by means of the standard CPAL code beautifier.

3.2. Model Transformation Utilities

The MT utilities layer (B in Figure 1) offers a set of generic AST manipulation functions that upper layers can use as building blocks, regardless of the specific dependability patterns they realize. Its design has been inspired by the AST optimizations of `gcc` and includes a compact subset of the functions available there. All functions in this layer are inherently recursive, as they have to traverse the AST structure. One of their goals is to shield the upper layers from the complexity and pitfalls of recursion. Their typical level of abstraction is best illustrated by an example. Among others, the layer provides the function:

```
int MT_Apply(AST *tree, enum ast_node_code type,
             char *name, mt_apply_f f, void *hl_cookie);
```

It recursively applies a functor `f` to all nodes of a given AST `tree` that match a pattern specified by a node `type` and `name`, as per a breadth-first traversal. The argument `hl_cookie` points to caller-defined state information, which is held for the whole application and passed to each invocation of `f`. The advantage of using `MT_Apply` becomes evident when considering that `f` is always invoked on one AST node at a time, regardless of AST complexity. Additional functions allow `f` to escape from the recursive descent and return a status indication to the top-level caller of `MT_Apply`, without delving into any details of how non-local jumps work.

Also within this layer, `MT_Apply` is used as a basis for more complex MT functions, for instance:

```
int MT_Make_Unique(AST *tree, char *suffix);
```

This function makes all global objects in an AST `tree` unique, by appending a unique `suffix` to their names, and returns the number of AST nodes it has modified. This function is useful to distinguish global objects in a dependability template when the same dependability pattern is applied to diverse components of a user model based on the same template, as better illustrated in Section 4.2.

Its implementation is straightforward and basically consists of a sequence of calls to `MT_Apply`. Each call scans the AST for a certain `type` of global object, without any filtering on the `name`. The functor `f` passed to `MT_Apply` synthesizes the new, unique name and updates the AST node by invoking `MT_Rename`, another utility function in this layer. The `hl_cookie`, which persists across all calls to `MT_Apply`, is used to propagate the `suffix` downwards (from `MT_Make_Unique` to `f`) and the count of modified AST nodes upwards.

In turn, the `MT_Rename` function:

```
void MT_Rename(AST *tree, char *old, char *new);
```

replaces an `old` top-level identifier with a `new` one in the AST `tree`. Its implementation is based on `MT_Transform`, a lower-level variant of `MT_Apply` on which `MT_Apply` itself is based. With respect to `MT_Apply`, `MT_Transform` supports finer control of recursion rules, which determine on which AST nodes the functor `f` will be invoked. This is necessary, in this case, to properly implement language scoping rules. More specifically, since the `MT_Rename` function shall only operate on a top-level identifier, its recursive descent along the AST must stop when it would enter a portion of AST where the top-level identifier is shadowed. This happens, for instance, when there is a parameter or a local variable with the same name.

3.3. Model Transformation Plugins

Above the MT utilities there is a layer composed of multiple MT modules, or *plugins* (C in Figure 1). Each MT plugin is responsible for a specific kind of MT on the input user model. All of them can accept arguments coming from the dependability annotations (specified in the user model), which trigger the execution of MT plugins and direct them to operate on specific entities of the user model. For instance, an argument may indicate which user process instance the MT plugin must transform. More details about dependability annotations are given in Section 3.4. In turn, a plugin is made up of two components: a template, or *skeleton*, written in CPAL, and the MT code written in C.

The skeleton contains the bulk of the transformed code, in which the parts that depend on user model contents and that must be specialized according to it are represented by placeholders. On its part, the MT code scans the user model AST and identifies the entities it shall operate on, based on its arguments. Then, it turns to the AST of the skeleton, locates the placeholders, and replaces them with the information extracted from the user model AST. At the end of this process, which resembles template instantiation in object-oriented languages, the specialized skeleton becomes the MT plugin output.

The dual-language approach to MT plugin implementation has several distinct advantages. Firstly, it greatly reduces the number of AST nodes the plugin must synthesize from scratch because most of the output AST can be copied from the AST of the skeleton. Synthesizing AST nodes within a plugin is possible by means of the layer presented in Section 3.1. But it must be done carefully to ensure the AST is still grammatically correct. Instead, the template AST comes from parsing a CPAL module and its conformance is therefore guaranteed by construction. Moreover, since skeleton placeholders are themselves legal language entities, a skeleton can be probed with the full suite of CPAL analysis and simulation tools while it is being developed. As a result, this reduces MT plugins development time and improves their quality.

This plugin-based architecture has the advantage of enabling the reuse of a large fraction of the MT framework. This is because the code specific to a certain MT method resides just in the corresponding plugin, whereas the other framework layers only contain general-purpose code.

Even more importantly, this approach fits well in the MDE paradigm outlined in Section 1 and adheres to sound software engineering techniques because:

a) a skeleton that corresponds to a certain fault tolerance (FT) technique represents a formally specified, but still human-readable, *design pattern* [42] for that technique;
b) it drives the automatic MT process, thus ensuring the *consistency* between the design pattern and the MT output;
c) it enables a clear separation between “what” the MT plugin should provide (stated in the dependability annotation) and “how” this goal is accomplished (realized by the plugin code based on the skeleton).

For instance, the same MT plugin code can operate on different skeletons depending on the execution target. In this way, the implementation details of a FT mechanism can be tailored to the target itself, while the plugin code is still target-independent.

This approach has the advantage of shielding novice users from the low-level implementation complexity of the design pattern because they will work only at the abstract annotation level. At the same time, this does not prevent advanced users from adapting design patterns according to their specific needs by working at the skeleton level, but still without ever delving below CPAL language abstraction. Indeed, the expressive power of CPAL ensures that plugins can have a broad range of functionalities, without being limited in principle to special, restricted application scenarios.

Lastly, the automatic application of design patterns also enhances model quality because both the pattern itself and its specialization (embodied by the MT plugin skeleton and code, respectively) can be developed, thoroughly analyzed and tested once, and then reused multiple times. Further information about the dual-language approach will be given in Section 4 by means of a case study involving the NVP dependability pattern.

3.4. Automated Augmentation Process

The augmentation process is implemented by the top layer of the MT framework (D in Figure 1). Each MT plugin is reg-

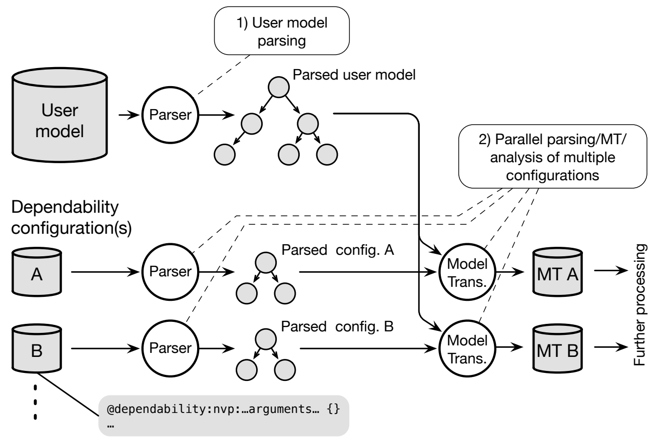


Figure 2: Parallel MT workflow.

istered in a configuration table that contains its name, the file name of its skeleton, and three pointers to its MT code. Of these, one points to the main MT method (also called *driver*) while the other two indicate pre- and post-processing methods.

As a whole, the MT process is driven by dependability annotations. To ease the specification of various kinds of non-functional aspects, the CPAL language incorporates a general-purpose annotation system. For consistency, dependability annotations adhere to the same syntax. They can therefore be regarded as embodying an *internal DSL* for dependability, which complies with its *host DSL* (CPAL itself in this case) and possesses adequate expressive power for the particular concern of interest. More specifically, a dependability annotation is written as:

```
@dependability:<MT_plugin>:<Arguments> {<Snippet>}
```

where `<MT_plugin>` is the name of a MT plugin, `<Arguments>` is a colon-separated list of plugin arguments, and `<Snippet>` is an optional code snippet. `<Snippet>` is currently unused but available to further extend the internal DSL's expressive power. An example of annotation is given at the bottom left of Figure 2. As outlined in the previous section, being declarative, the internal DSL hides the details related to dependability mechanisms and the MT process (namely the "how") from the designers, letting them focus only on specifying the "what".

Depending on the specific dependability pattern being called for, the use of a dependability annotation may or may not require further information or additional effort from the user. For instance, the application of triple modular redundancy (TMR) to a process does not. Indeed, in the latter case, MT will simply instantiate the target process, which is already present in the original model, in multiple identical copies. On the contrary, using NVP requires the user to provide a total of N diverse versions of the target process, whereas only one version had likely been defined in the original model.

Processing an annotation provokes the execution of a MT plugin. Multiple annotations are honored in sequence using the AST of the user model as initial input, and taking the output of the previous annotation as input to the next. The plugin name `<MT_plugin>` mentioned in the annotation is first looked up in the configuration table. Then, the plugin driver is called, with the AST of the corresponding skeleton passed to it, together with the input and output ASTs and any additional `<Arguments>` found in the annotation, also in AST form. If the driver is about to be called for the first time, the call is preceded by the invocation of the plugin pre-processor. Similarly, the plugin post-processor is invoked after processing all dependability annotations, if the plugin has been used at least once.

For efficiency, a MT cache stores the AST of the plugin skeleton for possible reuse. In addition, although the automated augmentation layer handles dependability annotations in sequence, it supports multiple threads of control that operate concurrently on different sets of annotations, possibly starting from the same user model.

Although end users are ultimately responsible of identifying the portion of their models to be augmented, pinpointing suitable MT modules, and expressing their needs by means of appropriate dependability annotations, the MT tool carries out augmentation in a completely automatic way, without further user intervention.

This approach is very favorable for productivity because users can solely focus on the high-level dependability properties their system must implement, without necessarily grasping all the lower-level details of how they are practically realized. At the same time, it also retains considerable flexibility. Indeed, as described in Section 3.3, nothing prevents more advanced users from adapting the standard design patterns provided by MT plugins to their own specific requirements, by working on their skeletons.

3.5. Model Transformation Workflow

The main program of the MT tool implements and coordinates the whole MT workflow. As shown in Figure 2, it takes a user model and a set of dependability configuration files as input. Each configuration file contains a set of dependability annotations, as specified in Section 3.4, and represents a dependability augmentation scenario the user would like to apply to the input model and investigate. Both the user model and the configuration files are first parsed to acquire their contents in AST form. Then, the tool activates the automated augmentation layer to apply each configuration to the user model. Finally, all the output ASTs are converted back to CPAL by means of the dumper described in Section 3.1, optionally beautified, and saved for further processing.

Since the MT framework as a whole is thread-safe and supports concurrent execution, the tool may fork into multiple threads, one for each scenario, in order to speed up the analysis. The same reasoning also applies to any further processing to be performed on the transformed models, because the standard CPAL analysis tools support concurrent execution, too. This is important from the user experience point of view, especially if design-space exploration is being performed by means of automatic tools rather than by hand as the number of distinct dependability scenarios those tools generate may be very large. A multi-threaded approach enables multiple scenarios to be analyzed efficiently in parallel, and then compared. Note also that, as a baseline case, when there is only one dependability configuration to be evaluated, dependability annotations can be embedded directly in the user model for convenience of use.

4. Model Augmentation with NVP

In this section, we apply the model-transformation framework detailed in Section 3 to the augmentation of the N-Version Programming dependability pattern.

4.1. N-Version Programming with an internal DSL

Software is particularly prone to design and implementation errors. Such errors may not be tolerable in safety or mission critical systems. NVP [22] is a proven effective mechanism that targets such kind of errors and enhances system dependability by increasing diversity and redundancy. More specifically, instead of a single version of a software component, it requires the software designer to provide N different implementations (called *member versions*) that realize the same functional logic so as to avoid common-mode faults. At run time, each member version computes independently its own outputs based on the same inputs. A decision algorithm, typically majority voting, determines the final outputs. Different actions can be taken upon detection of a faulty member version: continue, recovery, or even termination.

It is natural that member versions exhibit different complexities, leading to variable execution times and jitters. This increases the difficulty of assessing the suitability and impact of adopting fault tolerance mechanisms, such as NVP, in real-time systems. This concern can be partially addressed by exploring the timing annotation capability offered by CPAL. Estimated execution time and jitter can be specified for individual member versions and the decision algorithm. And timing annotations will be respected during simulation, when the NVP module is assessed alone, or as a component of a more complex system.

Furthermore, thanks to the executability of CPAL models, not only during simulation, but also on selected hardware platforms, it is possible to collect the worst-case execution time of a process on real hardware. This is provided as a built-in feature of the CPAL execution engine. Hence, realistic timings can be collected and re-injected into the models so as to carry out a more time-accurate simulation. This effectively complements schedulability analysis, which usually demonstrates a certain pessimism, to achieve a better understanding of the timing behavior of the system under design.

Typically, a control function in a CPS executes either periodically or upon the occurrence of events, computing commands for actuators based on sensor inputs. This can be straightforwardly modeled as a CPAL process that is either time-triggered or event-triggered, and whose interface is identified by sensor inputs and actuator outputs. For instance, the following code statement instantiates a control function that runs every 100 ms, whose functional logic is defined in `Original_Process`, working on two inputs and generating two outputs, which come from and are to be used by other components of the modeled CPS.

```
process Original_Process: p1[100ms](input1, input2,
                                    output1, output2);
```

The view of the functional architecture, that is, processes and data flows among them, as produced by the CPAL development environment is shown in the upper part of Figure 3. Processes are represented by rounded rectangles, with the process instance name and its activation pattern indicated within, while arrowhead lines represent data flows. Regular rectangles are for global variables, except buffered communication channels are shown with directed rectangles.

If a control function, e.g. the one represented by `p1`, is identified as a safety critical component by software designers, various fault-tolerance patterns can be applied to augment `p1` and enhance its dependability. Automated model augmentation can be driven by dependability annotations specified either in a configuration file, or directly in the user model as shown in Figure 2 and explained in Section 3.4. For simplicity, it is assumed the following dependability annotation is indicated directly in the user model, using NVP as an example.

```
@dependability:nvp:majority_voting:p1:Mem_1:Mem_2:Mem_3{}
```

As we can see, this is a specialization of the general dependability annotation outlined in Section 3.4, in which `nvp` indicates a NVP transformation is desired for enchancing dependability and also determines the `MT_plugin` to be used for the transformation. The arguments needed for NVP transformation consist of the decision algorithm (e.g. `majority_voting`), the component of the user model to apply NVP transformation to (e.g. the control function represented by process instance `p1`), and the member versions (e.g. `Mem_1`, `Mem_2`, and `Mem_3`).

Among them, the decision algorithm is provided as part of the NVP skeleton. More than one decision algorithm can be made available and can be referred to in the annotation simply by their name. Contrarily, the member versions shall come from the user model since their implementation is application specific. For the purpose of demonstration, only three member versions are indicated through their name in the above example. No limit is imposed by the MT framework on the number of member versions that can be specified in the annotation.

As mentioned in Section 3.4, more than one dependability annotation can be specified for the same or diverse components of the modeled system. For instance, a NVP dependability annotation can also be specified for another component, e.g. `p2`. If they relate to the same model file, their occurrence determines the order in which transformations will be performed. Dependability annotations for other fault tolerance mechanisms can be specified in a similar way, which together form the internal DSL for dependability presented in Section 3.4.

4.2. Generative transformation

This section illustrates the key transformation steps implemented in the MT driver for NVP, based on the example and the dependability annotation given in Section 4.1. The output of the MT process, namely the CPAL code of the transformed model is not shown here for conciseness. Instead, the functional architecture view as seen in the CPAL development environment is depicted in the lower part of Figure 3, with comments added to highlight the key functionality of each individual component in the transformed model.

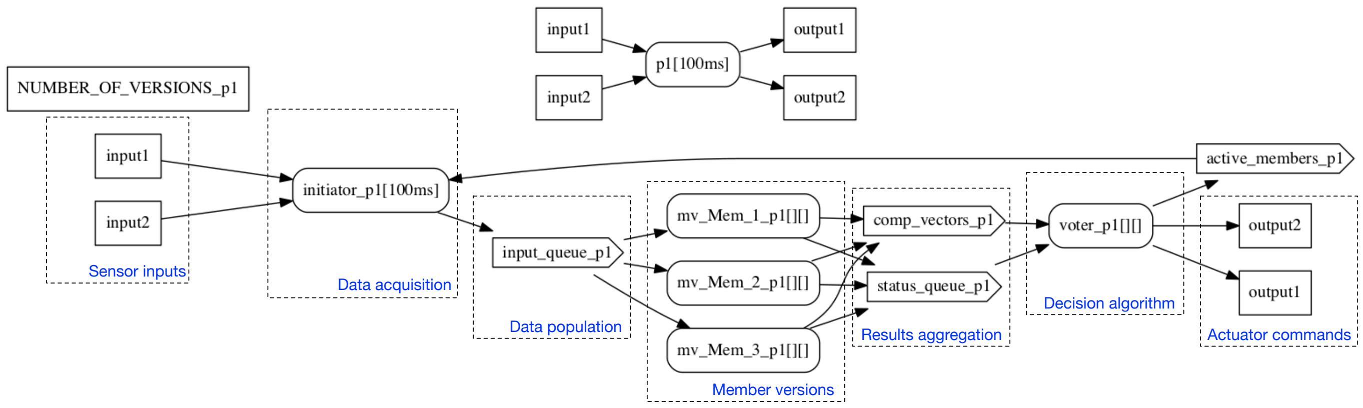


Figure 3: Functional architecture of the original model (upper part) and the transformed model (lower part) as seen in the CPAL development environment, with comments highlighting the key functionality of each individual component.

The transformation for NVP is performed based on two AST trees: the user model tree $\mathcal{T}_u$ and the NVP skeleton tree $\mathcal{T}_{ft}$. The general principle is to expand and customize $\mathcal{T}_{ft}$ based on information about the component to be transformed (denoted as C, and corresponding to `p1` in our example) and the parameters of the desired dependability pattern, both located in $\mathcal{T}_u$. In particular, the component to be transformed can be recognized in the dependability annotation, and its functional behavior is specified in the user model. Once customization completes, the expanded NVP tree $\mathcal{T}'_{ft}$ is merged with $\mathcal{T}_u$, by replacing the subtree corresponding to the component to be transformed, namely `p1`, with $\mathcal{T}'_{ft}$. This process may be repeated multiple times, if more than one dependability annotation has been indicated for the same user model.

The key transformation steps for NVP are as follows: first of all, inputs processed by C, e.g. the control function represented by `p1`, need to be populated to member versions. This task is delegated to an *initiator* process (as shown in Figure 3), which is provided as part of the NVP skeleton. Importantly, the initiator inherits the activation patterns (both time-triggered and event-triggered ones, if any) of `p1`. This ensures that the sampling times of inputs with the transformed model will remain the same as with the original user model. This is achieved by first locating `p1` in $\mathcal{T}_u$, recognizing its activation patterns, copying the subtree corresponding to the activation patterns from $\mathcal{T}_u$ and replacing the placeholders in the initiator template in $\mathcal{T}_{ft}$.

The decision algorithm, for example majority voting as indicated in the dependability annotation, is executed within a dedicated *voter* process. The *voter* process is responsible for collecting results generated by member versions and determining the outputs of the NVP-augmented model based on the decision algorithm. A template of the voter is available in the NVP skeleton as well. The dummy decision algorithm referred in the *voter* process will be replaced by name with the one specified in the annotation.

Inputs and outputs of C, such as `input1`, `input2`, `output1`, `output2` of `p1`, are passed as arguments to the initiator and the voter respectively, by copying the corresponding AST nodes from $\mathcal{T}_u$ to $\mathcal{T}_{ft}$ where the initiator and the voter instance are. This ensures that the augumented model works on the same set of inputs and outputs as C. It is worth remarking that this transformation step handles any number of inputs and outputs in C.

The NVP skeleton also includes a standard template for member versions, which contains a dummy subprocess declaration and call. This subprocess determines the functional logic of a member version. The template is instantiated once per each member version cited in the dependability annotation, with the dummy subprocess replaced by name with the user-defined member version. Hence, in the case of `p1`, the template should be instantiated three times, corresponding to the three member versions indicated in the dependability annotation, namely `Mem_1`, `Mem_2` and `Mem_3`, as shown in the middle of Figure 3. The subtree corresponding to the expanded templates are then appended to $\mathcal{T}_{ft}$. In this way, an instance of the user-defined member version is created and will be executed within the expanded template at run time. Hence, the functional logic is preserved as well.

The last step is to uniquely rename global objects in the expanded NVP skeleton $\mathcal{T}'_{ft}$, such as data types, function definitions, and variable declarations, with the utility function `MT_Make_Unique` introduced in Section 3.2. This is because the skeleton may need to be instantiated and customized multiple times if NVP transformation is indicated for more than one component of the same user model. If no renaming were applied, global objects with the same name would be defined more than once after all NVP transformations, which would be erroneous. Unique renaming is achieved by using the process instance name of C as suffix, e.g. `initiator_p1` as shown in Figure 3, in which the target process instance name `p1` is appended to the skeleton-derived base name `initiator` to obtain the full process instance name.

Once the NVP skeleton expansion based on $\mathcal{T}_{ft}$ is completed for a dependability annotation, C in the user model (e.g. the control function represented by `p1`) is replaced with its NVP-augmented version by substituting the subtree corresponding to C in $\mathcal{T}_u$ with the expanded $\mathcal{T}'_{ft}$. The subtree corresponding to the processed dependability annotation is removed from $\mathcal{T}_u$ as well. It is common that C interacts with other components of the modeled CPS, and hence, they are left untouched in $\mathcal{T}_u$ so as to preserve their way of interaction.

It is worth noting that member versions and the voter are both event-driven, with their execution triggered by input data from the initiator and populated through the inter-process communication channel, and by outputs computed by member versions and aggregated to the voter, respectively. Hence the sampling time of the transformed model only depends on the initiator. Last, the aforementioned generative transformation procedure is standard for any user model, regardless of its functional logic and complexity, as well as the way it interacts with other components of the modeled system through inputs and outputs. Hence, it can be implemented as an algorithm and can be fully automated.

As demonstrated in Figure 3, in order to augment `p1` with NVP, namely, transforming it from how it looks like in the upper part of the figure to the lower part, end users only need to specify the desired dependability pattern by means of a dependability annotation, and the rest is automatically handled by the framework. As shown in the figure, inputs/outputs as well as timings are retained. `p1` is replaced with three kinds of modules, namely the *initiator* which inherits activation patterns and populates inputs, the *member versions* which run the user-provided control logic, and the *voter* which encapsulates the decision algorithm.

4.3. Comparison with manual code development

Two distinct considerations can be brought forward to estimate the amount of programming time and effort that could be saved by using the MT framework described in this paper for NVP, instead of developing equivalent code by hand.

The first one is related to the quantity of CPAL code in the NVP skeleton. It currently consists of 252 lines and approximates the amount of additional code that programmers would have to write, were they not using the MT framework. This definitely implies a significant development effort, considering the conciseness of the CPAL language and the fact that the models of the two case studies to be discussed in Sections 5 and 6 comprise 353 and 328 lines of code, respectively.

In addition, when appraising manual code development, the extra testing time and effort that stem from deploying newly developed code, instead of instantiating a proven design pattern, must also be taken into account. They are harder to quantify exactly, but published literature states that typically testing consumes more than 40% of the resources and a lead time of 15–50% of a software development project [43].

The second consideration involves the hand-written NVP implementation outlined in [23], which is made up of about 1400 lines of C code. This figure gives an idea of the development effort programmers would face if they opted for a lower-level NVP implementation, rather than developing it in CPAL or by means of automated MT.

In summary, these data corroborate the idea that savings coming from automated MT are going to be significant, also in terms of code quality, as discussed in Section 3. Although the assessment has been performed based specifically on the NVP pattern, there is no reason to believe that other patterns would behave differently from this point of view.

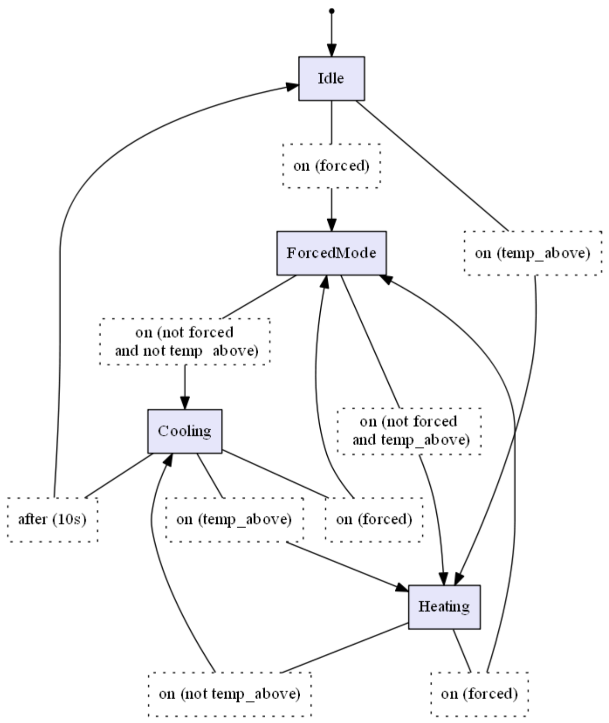


Figure 4: Logic of the chip temperature controller described as a FSM whose transitions between states (solid rectangles) are triggered by Boolean conditions that evaluate to true, or after a certain time spent in a continuous manner in a state (screenshot from CPAL-Editor).

5. Illustration: chip temperature control

In this section, the automated model transformation framework is further illustrated with NVP augmentation applied to a case study, namely a chip temperature controller design, to demonstrate the properties preserved during the transformation process.

5.1. Case study description

The controller under design maintains the temperature of a chip, equipped with a heatsink (i.e., convection heat spreader) and a fan, within a predefined range by controlling the fan speed in discrete steps. Below a certain temperature threshold (30 °C), the fan is idle. As soon as the temperature exceeds the threshold, the fan is set to full speed. When the temperature falls below the threshold, the fan speed is set to half speed for 10 s before going back to idle, unless the temperature exceeded the threshold again. In the latter case, the controller immediately brings the fan back to full speed. The controller includes a "forced" mode, which can be used to keep the fan at full speed regardless of the temperature. The functional logic of the controller is specified with the Finite State Machine (FSM) shown in Figure 4, activated every 100 ms. Threshold crossings

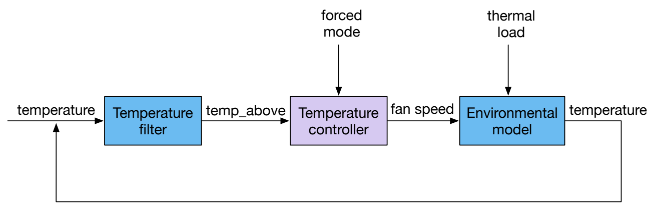


Figure 5: Abstract structure of the modeled CPS: sensing with the temperature filter; control through the temperature controller; actuation impact on the environment.

are dampened by a hysteresis-based temperature filter, implemented by another FSM not shown in the figure for simplicity.

The CPAL model of the temperature controller was originally conceived to execute on a Cortex®-M4 NXP FRDM-K64F target and interact with a temperature sensor (thermistor) and a 12 V fan controlled by pulse-width modulation (PWM). For the simulation experiments of this work, it was complemented by an environmental model, which represents the thermal behavior of a heatsink/fan combination and a heat source that supplies a varying thermal load at an ambient temperature of 20 °C. For the sake of illustration, the model considers natural and forced convection as described in [44], but neglects radiative heat transfer and the non-linearity of air flow velocity with respect to fan speed.

This case study, even though intuitive, represents a classic feedback control system. As depicted in Figure 5, the temperature filter module, which models a sensor node, samples the environment temperature every 100 ms, assesses if the predefined threshold is exceeded or not, and notifies the temperature controller accordingly. The temperature controller, which implements the control logic demonstrated in Figure 4, determines the actuator command. In other words, it sets the fan speed. Depending on the fan speed and the current thermal load, e.g heat generated on the chip due to computation, the temperature goes up and down. This is represented by the environmental model.

To improve the dependability of the modelled CPS, NVP augmentation is applied to the temperature controller using the MT framework. In the following, the CPAL model corresponding to the system depicted in Figure 5 is referred to as the *original* model, whereas the one augmented with NVP is denoted the *NVP-augmented* model. Observation of key properties of the NVP augmentation and the MT framework in general are made, for what concerns both timing domain and value domain correctness. Even though illustrated with the case study, the observed properties apply whatever the model the NVP pattern is applied to.

5.2. Timing domain correctness

As already illustrated in Figure 3, the NVP-augmented controller model works on the same set of inputs and outputs as the original model. At run time, its execution is driven by the initiator process as the member versions and the voter are purely event-triggered. Hence, the sampling times of the inputs depend only on the initiator, whose activation pattern is inherited from the original controller model. *Sampling times* are thus respected by the design of the NVP transformation.

The *actuation times* will differ because all member versions plus the voter should be executed before actuation can take place. Whether this departure from the initial timing behavior raises issues depends on the dynamics of the controlled system, the control law and its implementation. This is studied in Section 5.3 for our case-study. If required, like classically done in synchronous systems to offset temporal variabilities due to scheduling and varying execution times [45, 46], it is possible for both models (the original model and the NVP-augmented model) to postpone the application of the control at a time that ensures that member versions and voter have finished executing, for instance, immediately before the next activation of the controller. With this latter implementation, referred to as “deferred control application” in the following, the original and augmented models behave identically from a timing point of view.

While NVP improves reliability, it also increases CPU load due to extra *execution times* and may create deadline misses when the NVP-augmented model becomes part of a more complex software system. This shall be verified using a real-time schedulability analysis method suited to the task model and scheduling policy of the application. For instance, this is currently possible when tasks are periodic or triggered by periodic tasks, such as the processes introduced by NVP transformation, and they are scheduled with the default scheduling model of CPAL, that is, FIFO with priorities as secondary criterion to distinguish between tasks released at the same time. An open source implementation of this schedulability analysis published in [6] is available in [47].

5.3. Value domain correctness

With deferred control application, the values sampled (that is, the state of the controlled process) and the control applied in the original and the augmented models will always match in a fault-free environment. Here we are interested in the most common implementation where in both models the control is applied immediately after it has been computed, and thus at a later time for the NVP-augmented model.

Regarding the case study, we assume here an extreme value of 90 ms of additional execution time per cycle for NVP, while the cycle time remains at 100 ms. Under this assumption, Figure 6 shows the evolution of the fan speed and system temperature under the same thermal load scenario. For the sake of clarity, both the fan speed and the thermal load are normalized to 1 in the figure. More specifically, a fan speed of 1 corresponds to full speed. Similarly, a thermal load of 1 is the maximum thermal load the system can sustain while still maintaining the temperature at or below the predefined threshold, e.g. 30 °C. When the thermal load is higher than unity, heatsink convection will stabilize the temperature at a higher value while the controller keeps the fan at full speed.

In simulations without faults (e.g., the first 400 s of Figure 6), the outputs of the two models match very closely, leading to a difference in temperature of maximum 0.8 °C in long experiments (not shown here for conciseness) with varying thermal

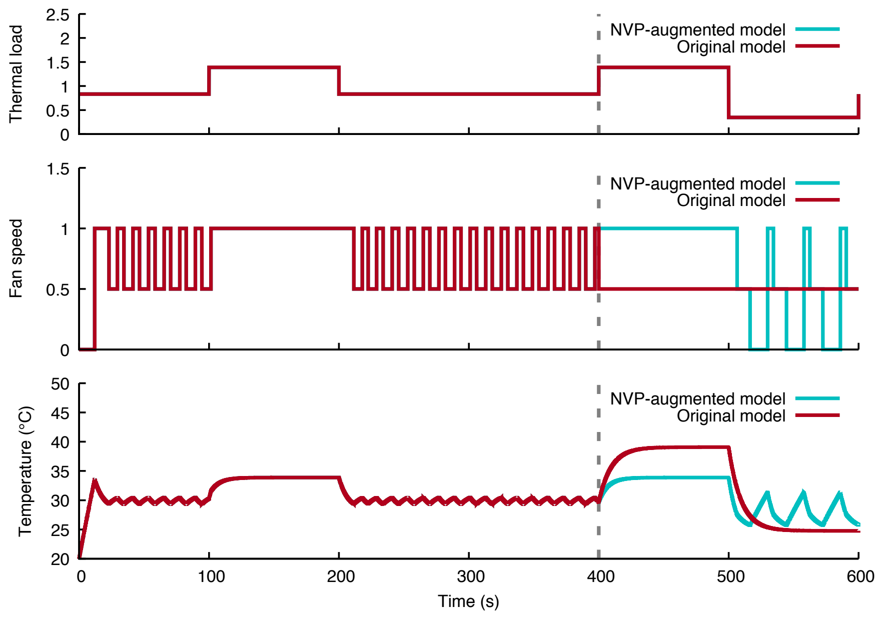


Figure 6: Fan speed computed by the original and NVP-augmented model (middle plot) for a given thermal load scenario (upper plot) and resulting temperature of the chip (lower plot). Faults are injected from time 400 s onward to both models.

load. It should be pointed out that a sensitivity analysis based on input and output jitter margin [48] can be used to determine the maximum tolerable additional execution time induced by the NVP so that the quality of control remains acceptable.

In the second part of Figure 6, from time 400 s onward, a member version becomes permanently faulty and the same fault is also injected into the original model. If no fault is injected, the system should behave similarly as during the time interval 100 s to 200 s, since the thermal load and the ambient temperature are the same. However, as we can see, in the original model the control is no longer correct as the fan operates steadily at half speed, while it should be working at full speed from 400 s to 500 s and alternate between idle and full speed from 500 s to 600 s when the thermal load is significantly lowered down. As a result, the chip temperature in the original model is either too high or too low, with respect to the set point.

5.4. Code coverage of the NVP-augmented model

It is worth noting that, in order to obtain evidence of the correctness of the augmented model and check that no dead code was introduced during model transformation, long simulations have been performed, with and without fault injection.

In particular, the original model and the NVP-augmented model are executed in parallel under the same configuration. “Forced mode” and “thermal load”, the two free variables of the models, are varied randomly during long simulations in order to comprehensively stimulate the models. From half of the simulation onward, a fault is introduced at every cycle to one of the member versions. As expected, the NVP-augmented controller was always able to reach an agreement between member versions. The output computed by both models, that is, the fan speed, are compared at every cycle and they always matched,

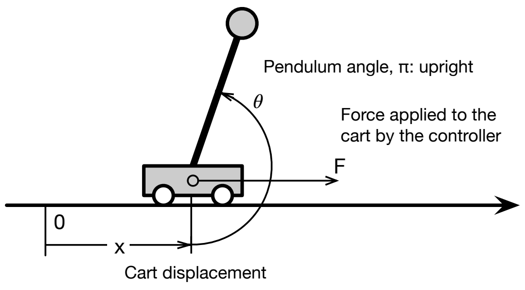

Pendulum angle, π: upright
θ
Force applied to the cart by the controller
F
0
x
Cart displacement

Figure 7: Schematization of an inverted pendulum on a cart.

regardless of the presence of the fault in the member version. The 9600 s simulations performed were able to achieve 100% code coverage, modulo the “no majority reached” statements in the NVP-augmented model, which cannot be executed with a single fault.

6. Illustration: inverted pendulum

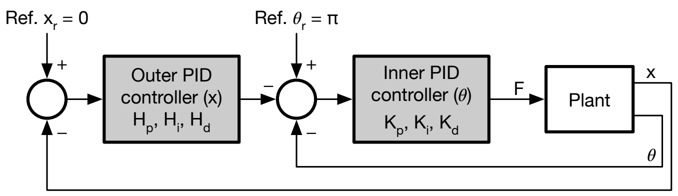
To further illustrate its suitability for different application scenarios, the MT framework was also applied to a dissimilar case study, more specifically to the NVP augmentation of a controller that balances an inverted pendulum on a movable cart. This is a well-known example of non-linear, unstable control problem. With respect to the chip temperature controller analyzed in Section 5, it is much more sensitive to any disturbances NVP augmentation may introduce in the control loop.

6.1. Plant model

The plant being modeled is depicted in Figure 7. It consists of a cart of mass M that moves in a mono-dimensional space

Parameter		Value
M	(mass of the cart)	0.5 kg
m	(mass of the pendulum)	0.2 kg
l	(length to the pendulum center of mass)	0.3 m
I	(moment of inertia of the pendulum)	$0.018\,\mathrm{kg}\cdot\mathrm{m}^2$
b	(coefficient of friction)	$0.1\,\mathrm{N}\cdot\mathrm{s/m}$
g	(acceleration of gravity)	$9.8\,\mathrm{m/s}^2$

Table 1: Physical parameters of the plant shown in Figure 7.



Ref. x_r = 0
Ref. θ_r = π
Outer PID controller (x) H_p, H_i, H_d
Inner PID controller (θ) K_p, K_i, K_d
F
Plant
x
θ

Figure 8: Cascaded PID controller schematic. In the case study, $K_p = 9$, $K_i = 0.95$, $K_d = 300$, $H_p = 0.001$, $H_i = 0$, $H_d = 10$ with a sampling interval of 1 KHz.

and carries an inverted pendulum of mass m, length l, and moment of inertia I. For simplicity, friction is considered to be proportional to the speed of the cart with proportionality constant b. The whole system is subject to gravity acceleration g, which makes the pendulum unstable in the upright position. At each instant, x represents the relative position (displacement) of the cart with respect to its initial position and θ is the angle of the pendulum. The controller applies a force F to the cart with the goal of keeping the pendulum upright ($\theta = \pi$) and nullify its displacement ($x = 0$). Table 1 summarizes the physical parameters of the plant, derived from [49].

The two governing equations that express x and θ (and their derivatives) as a function of F and the physical parameters of the plant can be written as:

$$\begin{aligned} (M+m)\,\ddot{x} + (ml\cos(\theta))\,\ddot{\theta} &= F - b\dot{x} + ml\theta^2\sin(\theta) \\ (ml\cos(\theta))\,\ddot{x} + (I+ml^2)\,\ddot{\theta} &= -mgl\sin(\theta) \end{aligned} \quad (1)$$

These equations provide a straightforward way to simulate the plant, by solving them with respect to $\ddot{x}$ and $\ddot{\theta}$ and then proceeding to calculate x and θ through numerical integration. Carrying out the simulation in a process with a period of $100\,\mu$s ensures simulation fidelity, because this period is at least one order of magnitude smaller than the plant time constants.

To evaluate the controller's behavior, the plant model also includes an additional process that runs every 20 s. This process perturbs the system by applying an impulse of random magnitude to the pendulum, thus changing its angle θ and bringing the system out of equilibrium. For the sake of simplicity, it is assumed that the variation of θ induced by the impulse takes place instantaneously.

6.2. Inverted pendulum controller

The controller used in the case study is a classic cascaded discrete-time proportional-integrative-derivative (PID) controller, whose structure is depicted in Figure 8. The inner controller controls the plant parameter that varies more rapidly, the pendulum angle θ in this case. At the same time, the outer controller adjusts the set point of the inner controller to control the plant parameter that varies more slowly, that is, the cart displacement x.

Considering that the controller itself was not a focal point of this work, the controller parameters were determined in an approximate way after linearizing the model in θ, and were not optimized afterwards. For this reason, a certain amount of ringing at the natural frequency of the pendulum is visible in the simulation results to be discussed in the following.

The discrete-time controller is implemented as a periodic process with a period of 1 ms that coincides with the sampling interval. As illustrated in Figure 8, the controller acquires the current values of θ and x from the plant model on each cycle and calculates the force F to be imparted to the cart. Backward finite differences are used as approximations for first-order derivatives. Similarly, summations weighted by the sampling time approximate the integral term, according to usual practice.

As in the previous case study, automatic model transformation was applied to this controller process to obtain a NVP-augmented model with $N = 3$, conforming to the general structure outlined in the bottom part of Figure 3.

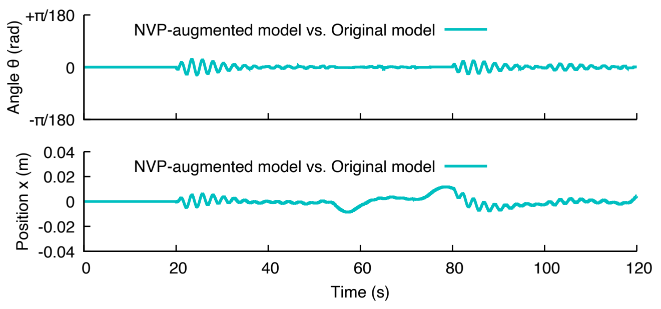


Figure 9: Difference in x and θ between the NVP-augmented and the original model, showing the effect of an additional execution time of $900\,\mu$s in the NVP-augmented controller, random perturbations introduced every 20 s.

6.3. Timing and value domain correctness

The negative effect of the extra execution time of the NVP-augmented controller, due to its higher complexity, was evaluated by introducing a delay of $900\,\mu$s in output actuation by the voter process. Although this delay is rather extreme (it amounts to 90% of the sampling time) the experimental results shown in Figure 9 show that its actual effect on control accuracy is quite limited.

The figure depicts the difference in the pendulum angle θ and cart position x between the original and the NVP-augmented controller during a 120 s simulation in absence of faults. Differences in θ stayed well within one degree and differences in x were limited to no more than ± 2 cm. As also discussed in Sections 5.2 and 5.3, if this limited degradation is deemed unacceptable, deferred control application remains a viable option also in this case.

At the same time, the very good agreement between the two controllers indirectly confirms that MT transformation operated correctly from the timing and value correctness point of view.

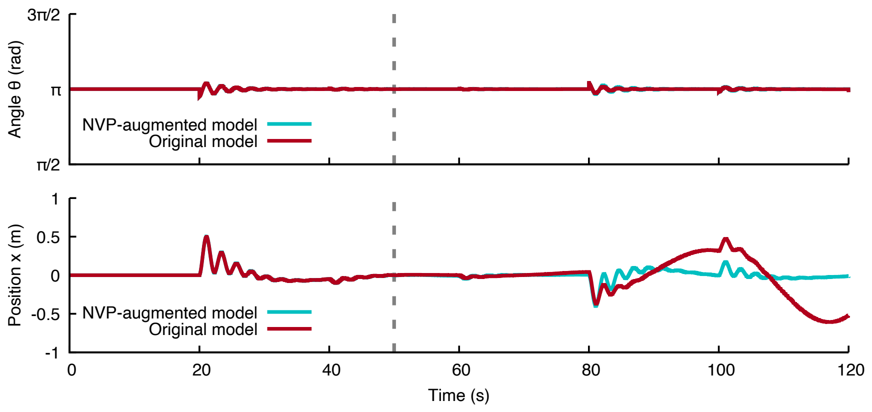


Figure 10: Effect of the lack of K_i in the original and NVP-augmented controller due to a fault injected from $t = 50$ s onward (vertical dashed line), random perturbations introduced every 20 s.

6.4. Fault injection

The next set of simulations was aimed at verifying that the NVP-augmented model is fault tolerant. To this purpose, a fault was injected in both the original controller and one of the member versions of the NVP-augmented controller. The fault starts at $t = 50$ s and mimics the loss of the integrative term of the inner-loop PID controller, that is, it forces $K_i = 0$.

As shown in Figure 10, the fault remained silent until $t = 80$ s due to the limited magnitude of the system perturbations introduced meanwhile. Afterwards, the now-faulty original controller still retained the ability of keeping the pendulum in an upright position, since no significant deviations of the pendulum angle around the equilibrium point $\theta = \pi$ are visible in the upper right part of the figure.

However, the original controller lost the ability to limit cart displacement and significant deviations from the set point $x = 0$ became evident, as highlighted in the bottom right part of Figure 10. On the contrary, and as expected, the NVP-augmented controller remained fully functional although one of its member versions failed.

Finally, code coverage analysis of the NVP-augmented model was performed. The 120 s simulation whose results have been shown in Figure 10 was sufficient to achieve 100% coverage. As described in Section 5.4 regarding the first case study, the only statements not executed have to do with the voter being unable to determine a majority, which is impossible in presence of a single fault.

In summary, this second case study further confirmed the viability of the model transformation process, as well as its applicability to a dissimilar and more complex control problem.

7. Conclusion

In this work, we develop a framework and the associated toolset to augment a model with dependability mechanisms in an automated manner. In particular, end-users such as system designers are only required to indicate the part of a modelled system to apply fault tolerance augmentation to, by means of annotations. Then the augmentation process is fully automated by the framework. The framework is showcased by augmenting an intuitive, but yet realistic Cyber-Physical System with one representative fault-tolerance pattern, namely N-Version Programming. It is worth remarking that other fault-tolerance patterns can be included in the framework in a similar way.

Importantly, the abstraction level of the transformation retains both the benefits of high-level modeling and the executability, which allows for early stage validation, for example, by means of simulation. The natural next step is to automate the implementation of fault-injection patterns to inject both data and timing errors.

In the longer run, this work is a building block towards the use of design-space exploration to select, implement and validate the best set of dependability mechanisms to meet certain dependability objectives (e.g. a reliability requirement imposed by a given safety standard). Our open-source MT framework is a key component in that regard, as it can be extended with plugins and has been designed for efficiency with internal caches. Multi-threading is also an attractive feature from the users' perspective. It is especially useful when performing compute-intensive analyses, like design-space exploration, in which multiple design options and choices of different dependability mechanisms can be evaluated concurrently and then compared.

References

[1] J. Cabot, R. Clarisó, M. Brambilla, S. Gérard, Cognifying model-driven software engineering., in: M. Seidl, S. Zschaler (Eds.), STAF Workshops, Vol. 10748 of Lecture Notes in Computer Science, Springer, 2017, pp. 154–160.

[2] E. Lee, The past, present and future of cyber-physical systems: A focus on models, Sensors 15 (3) (2015) 4837–4869. doi:10.3390/s150304837.

[3] P. Derler, E. A. Lee, A. Sangiovanni Vincentelli, Modeling cyber–physical systems, Proceedings of the IEEE 100 (1) (2012) 13–28. doi:10.1109/JPROC.2011.2160929.

[4] D. Ameller, X. Franch, C. Gómez, S. Martínez-Fernández, J. Araujo, S. Biffl, J. Cabot, V. Cortellessa, D. Méndez, A. Moreira, H. Muccini, A. Vallecillo, M. Wimmer, V. Amaral, W. Bühm, H. Bruneliere, L. Burgueño, M. Goulão, S. Teufl, L. Berardinelli, Dealing with non-functional requirements in model-driven development: A survey, IEEE Transactions on Software Engineering (2019) 1–1. doi:10.1109/TSE.2019.2904476.

[5] S. Lampke, S. Schliecker, D. Ziegenbein, A. Hamann, Resource-aware control – model-based co-engineering of control algorithms and real-time systems, SAE Int. J. Passeng. Cars – Electron. Electr. Syst. 8 (2015) 106–114. doi:10.4271/2015-01-0168.

[6] S. M. Sundharam, N. Navet, S. Altmeyer, L. Havet, A model-driven co-design framework for fusing control and scheduling viewpoints, Sensors 18 (2) (2018) 628. doi:10.3390/s18020628.

[7] X. Wang, N. Hovakimyan, L. Sha, L1simplex: Fault-tolerant control of cyber-physical systems, in: Proc. ACM/IEEE International Conference on Cyber-Physical Systems (ICCPS), 2013, pp. 41–50.

[8] S. Yoon, J. Lee, Y. Kim, S. Kim, H. Lim, Fast controller switching for fault-tolerant cyber-physical systems on software-defined networks, in: Proc. 22nd IEEE Pacific Rim International Symposium on Dependable Computing (PRDC), 2017, pp. 211–212. doi:10.1109/PRDC.2017.35.

[9] Y. Xu, I. Koren, C. M. Krishna, AdaFT: A framework for adaptive fault tolerance for cyber-physical systems, ACM Trans. Embed. Comput. Syst. 16 (3) (2017) 79:1–79:25. doi:10.1145/2980763.

[10] S. Bernardi, J. Merseguer, D. C. Petriu, Dependability modeling and analysis of software systems specified with UML, ACM Comput. Surv. 45 (1) (2012) 2:1–2:48. doi:10.1145/2379776.2379778.

[11] A.-E. Rugina, K. Kanoun, M. Kaaniche, Software dependability modeling using AADL (architecture analysis and design language), International Journal of Performability Engineering 7 (2011) 313–325.

[12] C. Buckl, D. Sojer, A. Knoll, FTOS: model-driven development of fault-tolerant automation systems, in: Proc. 15th IEEE Conference on Emerging Technologies Factory Automation (ETFA), 2010, pp. 1–8. doi:10.1109/ETFA.2010.5641211.

[13] M. Antoni, Formal validation method and tools for computerized interlocking system, Presentation at the 18th International Symposium on Formal Methods (FM 2012), Industry Day, available at `http://fm2012.cnam.fr/fm2012/ID2012-Marc-Antoni.pdf` (August 2012).

[14] B. Selic, Programming $\subset$ modeling $\subset$ engineering, in: Proc. 7th Intl. Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA), 2016, pp. 11–26. doi:10.1007/978-3-319-47169-3_2.

[15] N. Navet, L. Fejoz, CPAL: High-level abstractions for safe embedded systems, in: Proc. 16th Workshop on Domain-Specific Modeling, DSM'16, ACM, Amsterdam, Netherlands, 2016.

[16] G. J. Holzmann, The model checker SPIN, IEEE Transactions on Software Engineering 23 (1997) 279–295.

[17] I. Cibrario Bertolotti, T. Hu, N. Navet, Model-based design languages: A case study, in: Proc. 13th IEEE International Workshop on Factory Communication Systems (WFCS), 2017, pp. 1–6. doi:10.1109/WFCS.2017.7991964.

[18] A. Benveniste, P. Caspi, S. A. Edwards, N. Halbwachs, P. L. Guernic, R. de Simone, The synchronous languages 12 years later, Proceedings of the IEEE 91 (1) (2003) 64–83. doi:10.1109/JPROC.2002.805826.

[19] M. Voelter, D. Ratiu, B. Schaetz, B. Kolb, Mbeddr: An extensible C-based programming language and IDE for embedded systems, in: Proc. 3rd Annual Conference on Systems, Programming, and Applications: Software for Humanity (SPLASH '12), 2012, pp. 121–140.

[20] L. Fejoz, B. Régnier, P. Miramont, N. Navet, Simulation-based fault injection as a verification oracle for the engineering of time-triggered Ethernet networks, in: Proc. of Embedded Real-Time Software and Systems (ERTSS'18), 2018.

[21] G. Bloom, G. Cena, I. Cibrario Bertolotti, T. Hu, N. Navet, A. Valenzano, Event notification in CAN-based sensor networks, IEEE Transactions on Industrial Informatics 15 (10) (2019) 5613–5625. doi:10.1109/TII.2019.2904082.

[22] A. Avižienis, The methodology of N-version programming, in: M. R. Lyu (Ed.), Software Fault Tolerance, John Wiley & Sons, Inc., 1995, pp. 23–46.

[23] T. Hu, I. Cibrario Bertolotti, N. Navet, Towards seamless integration of N-Version Programming in model-based design, in: Proc. 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2017, pp. 1–8.

[24] K. Ding, A. Morozov, K. Janschek, More: Model-based redundancy for simulink, in: B. Gallina, A. Skavhaug, F. Bitsch (Eds.), Computer Safety, Reliability, and Security, Springer International Publishing, Cham, 2018, pp. 250–264.

[25] S. Bernardi, J. Merseguer, D. C. Petriu, A dependability profile within MARTE, Software & Systems Modeling 10 (3) (2011) 313–336. doi:10.1007/s10270-009-0128-1.

[26] L. Montecchi, P. Lollini, A. Bondavalli, Towards a MDE transformation workflow for dependability analysis, in: Proc. 16th IEEE International Conference on Engineering of Complex Computer Systems, 2011, pp. 157–166. doi:10.1109/ICECCS.2011.23.

[27] A. El-Hokayem, Y. Falcone, M. Jaber, Modularizing behavioral and architectural crosscutting concerns in formal component-based systems – application to the behavior interaction priority framework, Journal of Logical and Algebraic Methods in Programming 99 (2018) 143–177. doi:https://doi.org/10.1016/j.jlamp.2018.05.005.

[28] V. Sfyrla, G. Tsiligiannis, I. Safaka, M. Bozga, J. Sifakis, Compositional translation of Simulink models into synchronous BIP, in: Proc. International Symposium on Industrial Embedded System (SIES), 2010, pp. 217–220. doi:10.1109/SIES.2010.5551374.

[29] M. Kölbl, S. Leue, H. Singh, From SysML to model checkers via model transformation, in: M. d. M. Gallardo, P. Merino (Eds.), Model Checking Software, Springer International Publishing, Cham, 2018, pp. 255–274.

[30] T. Mens, P. V. Gorp, A taxonomy of model transformation, Electronic Notes in Theoretical Computer Science 152 (2006) 125–142, Proc. International Workshop on Graph and Model Transformation (GraMoT 2005). doi:https://doi.org/10.1016/j.entcs.2005.10.021.

[31] J. Brunel, D. Doligez, R. R. Hansen, J. L. Lawall, G. Muller, A foundation for flow-based program matching: Using temporal logic and model checking, in: Proc. 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '09, ACM, 2009, pp. 114–126.

[32] A. Ataíde, J. P. Barros, I. S. Brito, L. Gomes, Towards automatic code generation for distributed cyber-physical systems: A first prototype for Arduino boards, in: Proc. 22nd IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), 2017, pp. 1–4. doi:10.1109/ETFA.2017.8247737.

[33] X. Koutsoukos, G. Karsai, A. Laszka, H. Neema, B. Potteiger, P. Volgyesi, Y. Vorobeychik, J. Sztipanovits, SURE: A modeling and simulation integration platform for evaluation of secure and resilient cyber–physical systems, Proceedings of the IEEE 106 (1) (2018) 93–112. doi:10.1109/JPROC.2017.2731741.

[34] J. Sztipanovits, T. Bapty, X. Koutsoukos, Z. Lattmann, S. Neema, E. Jackson, Model and tool integration platforms for cyber–physical system design, Proceedings of the IEEE 106 (9) (2018) 1501–1526. doi:10.1109/JPROC.2018.2838530.

[35] F. Cremona, M. Morelli, M. Di Natale, TRES: A modular representation of schedulers, tasks, and messages to control simulations in simulink, in: Proc. 30th Annual ACM Symposium on Applied Computing, SAC'15, ACM, New York, NY, USA, 2015, pp. 1940–1947. doi:10.1145/2695664.2695876.

[36] A. Cervin, D. Henriksson, B. Lincoln, J. Eker, K.-E. Arzen, How does control timing affect performance? Analysis and simulation of timing using Jitterbug and TrueTime, IEEE Control Systems Magazine 23 (3) (2003) 16–30. doi:10.1109/MCS.2003.1200240.

[37] Y. Jiang, H. Song, Y. Yang, H. Liu, M. Gu, Y. Guan, J. Sun, L. Sha, Dependable model-driven development of cps: From stateflow simulation to verified implementation, ACM Trans. Cyber-Phys. Syst. 3 (1) (2018) 12:1–12:31. doi:10.1145/3078623.

[38] J. Colaço, B. Pagano, C. Pasteur, M. Pouzet, Scade 6: From a Kahn semantics to a Kahn implementation for multicore, in: Proc. Forum on Specification Design Languages (FDL), 2018, pp. 5–16. doi:10.1109/FDL.2018.8524052.

[39] N. Kahani, M. Bagherzadeh, J. R. Cordy, J. Dingel, D. Varró, Survey and classification of model transformation tools, Softw. Syst. Model. 18 (4) (2019) 2361–2397. doi:10.1007/s10270-018-0665-6.

[40] L. Burgueño, J. Cabot, S. Gérard, The future of model transformation languages: An open community discussion., The Journal of Object Technology 18 (2019) 7:1. doi:10.5381/jot.2019.18.3.a7.

[41] R. Hebig, C. Seidl, T. Berger, J. K. Pedersen, A. Wąsowski, Model transformation languages under a magnifying glass: A controlled experiment with Xtend, ATL, and QVT, in: Proc. 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE), ACM, New York, NY, USA, 2018, pp. 445–455. doi:10.1145/3236024.3236046.

[42] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object-oriented Software, Addison-Wesley Longman Pub-

lishing Co., Inc., Boston, MA, USA, 1995.

[43] C. Ebert, R. Dumke, Software Measurement: Establish – Extract – Evaluate – Execute, Springer Berlin Heidelberg, 2007.

[44] J. A. Visser, P. Gauché, A computer model to simulate heat transfer in heat sinks, Transactions on Engineering Sciences 12 (1996) 569–578.

[45] T. A. Henzinger, B. Horowitz, C. M. Kirsch, Giotto: a time-triggered language for embedded programming, Proceedings of the IEEE 91 (1) (2003) 84–99. doi:10.1109/JPROC.2002.805825.

[46] R. Gerber, S. Hong, Slicing real-time programs for enhanced schedulability, ACM Trans. Program. Lang. Syst. 19 (3) (1997) 525–555. doi:10.1145/256167.256394.

[47] S. Altmeyer, S. Sundharam, FIFO scheduling analysis, `https://github.com/SebastianAltmeyer/FIFO-Scheduling-Analysis` (2016).

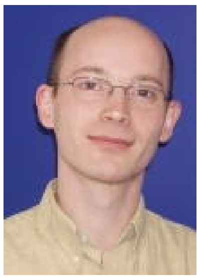
[48] A. Cervin, Stability and worst-case performance analysis of sampled-data control systems with input and output jitter, in: Proc. American Control Conference (ACC), 2012, pp. 3760–3765. doi:10.1109/ACC.2012.6315304.

[49] University of Michigan, Control tutorials for MATLAB and Simulink: Inverted pendulum, Available online, at `http://ctms.engin.umich.edu/CTMS/`.

Tingting Hu received her master degree in Computer Engineering in 2010 and Ph.D. degree with the best dissertation award in Computer and Control Engineering in 2015 both from Politecnico di Torino, Turin, Italy.
She works as a research scientist in the University of Luxembourg with the Faculty of Science, Technology and Communication. Formerly, she was with the University of Luxembourg as a post-doc researcher (2017-2018). Between 2010 and 2016, she also worked as a research fellow in the National Research Council of Italy (CNR), Turin, Italy. Her primary research interest concerns embedded systems design and implementation, spanning through topics such as real-time operating systems, communication protocols, formal verification of software modules and communication protocols, as well as security, with a special focus on the practical application of these concepts. Currently, she is focusing on the research of model driven engineering for safety-critical embedded systems. In the meantime, she serves as a program committee member and technical referee for several primary conferences and journals in her research area.

Ivan Cibrario Bertolotti received the Laurea degree *(summa cum laude)* in computer science from the University of Torino, Turin, Italy, in 1996.
Since then, he has been a Researcher with the National Research Council of Italy (CNR), Rome, Italy. Currently, he is with the Institute of Electronics, Computer and Telecommunication Engineering (IEIIT), Turin. He has taught several courses on real-time operating systems at Politecnico di Torino, Turin; has co-authored two books on the same topics; and serves as a Technical Referee for primary international journals and conferences. His research interests include real-time operating system design and implementation, industrial communication systems and protocols, as well as modeling languages and runtime support for cyber-physical systems. He received, as a coauthor, the Best Paper Award presented at the 8th IEEE Workshops on Factory Communication Systems (WFCS 2010).

Nicolas Navet has been a professor in Computer Science at the University of Luxembourg since May 2012. Formerly, from 1995 to 2012, he was with INRIA in France, as doctoral candidate, researcher then head of a research team in real-time systems. His research interests include real-time and embedded systems, communication protocols and risk assessment. Since the mid-1990s, he has worked on many projects with OEMs and suppliers in the automotive and aerospace domains. More information on his work can be found at `http://nicolas.navet.eu`.



Lionel Havet has been a research engineer at RealTime-at-Work since 2013. He is responsible for the development of the CPAL language interpreter and its Integrated Development Environment. His research interests include real-time scheduling analysis and dependability evaluation. He worked in the area of embedded systems at Sagem, Giat Industries, Alspace, Philips and lead during 3 years the development of the software embedded in automatic gearboxes at General Motors. He graduated in 1996 from ISAE engineer school in Toulouse.