Workshop Proceedings

ACES^{MB} 2009

Second International Workshop on Model Based Architecting and Construction of Embedded Systems

October 6th, 2009, Denver, Colorado, USA

Organized in conjunction with MoDELS'09
12th International Conference on Model Driven Engineering Languages and Systems

Edited by:
Stefan Van Baelen (K.U.Leuven - DistriNet, Belgium)
Thomas Weigert (Missouri University of Science and Technology, USA)
Ileana Ober (University of Toulouse - IRIT, France)
Huascar Espinoza (CEA - LIST, France)

Table of Contents

Foreword

The development of embedded systems with real-time and other critical constraints raises distinctive problems. In particular, development teams have to make very specific architectural choices and handle key non-functional constraints related to, for example, real-time deadlines and to platform parameters like energy consumption or memory footprint. In this context, the last few years have seen an increased interest in using model-based engineering (MBE) techniques. MBE techniques are interesting and promising for the following reasons: They allow to capture dedicated architectural and non-functional information in precise (and even formal) domain-specific models, and they support a layered construction of systems, in which the (platform independent) functional aspects are kept separate from architectural and non-functional (platform specific) aspects, where the final system is obtained by combining these aspects later using model transformations.

The objective of this workshop is to bring together researchers and practitioners interested in model-based software engineering for real-time embedded systems. We are seeking contributions relating to this subject at different levels, from modeling languages and semantics to concrete application experiments, from model analysis techniques to model-based implementation and deployment. Given the criticality of the application domain, we particularly focus on model-based approaches yielding efficient and provably correct designs. Concerning models and languages, we welcome contributions presenting novel modeling approaches as well as contributions evaluating existing ones. The workshop targets in particular:

- Architecture description languages (ADLs). Architecture models are crucial elements in system and software development, as they capture the earliest decisions which have a huge impact on the realization of the (non-functional) requirements, the remaining development of the system or software, and its deployment. We are particularly interested in examining:
    - Position of ADLs in an MDE approach;
    - Relations between architecture models and other types of models used during requirement engineering (e.g., SysML, EAST-ADL, AADL), design (e.g., UML), etc.;
    - Techniques for deriving architecture models from requirements, and deriving high-level design models from architecture models;
    - Verification and early validation using architecture models.

- Domain specific design and implementation languages. To achieve the high confidence levels required for critical embedded systems through analytical methods, in practice languages with particularly well-behaved semantics are often used, such as synchronous languages and models (Lustre/SCADE, Signal/Polychrony, Esterel), super-synchronous models (TTA, Giotto), scheduling-friendly models (HRT-UML, Ada Ravenscar), or the like. We are interested in examining the model-oriented counterparts of such languages, together with the related analysis and development methods.
- Languages for capturing non-functional constraints (MARTE, AADL, OMEGA, etc.)
- Component languages and system description languages (SysML, MARTE, EAST-ADL, AADL, BIP, FRACTAL, Ptolemy, etc.).

We accepted 10 papers for the workshop from 8 different countries: 7 full papers and 3 short papers. We hope that the contributions for the workshop and the discussions during the workshop will help to contribute and provide interesting new insights in Model Based Architecting and Construction of Embedded Systems.

The ACES^{MB} 2009 organizing committee,

Stefan Van Baelen,
Thomas Weigert,
Ileana Ober,
Huascar Espinoza.

The ACES^{MB} 2009 steering committee,

Mamoun Filali,
Sébastien Gérard,
Susanne Graf,
Iulian Ober.

September 2009.

Acknowledgments

The Organizing Committee of ACES^{MB} 2009 would like to thank the workshop Program Committee for their helpful reviews.

Nicolas Belloir (LIUPPA, France)
Jean-Michel Bruel (IRIT, France)
Agusti Canals (CS, France)
Arnaud Cuccuru (CEA-LIST, France)
Huascar Espinoza (CEA LIST, France)
Jean-Marie Farines (UFSC, Brasil)
Peter Feiler (SEI, USA)
Mamoun Filali (IRIT, France)
Robert France (CSU, USA)
Pierre Gaufillet (Airbus, France)
Sébastien Gérard (CEA LIST, France)
Susanne Graf (VERIMAG, France)
Bruce Lewis (US Army, USA)
Ileana Ober (IRIT, France)
Iulian Ober (IRIT, France)
Isabelle Perseil (Telecom ParisTech, France)
Dorina Petriu (Carleton University, Canada)
Bernhard Rumpe (RWTH Aachen, Germany)
Douglas C. Schmidt (Vanderbilt University, USA)
Bran Selic (Malina Software, Canada)
Jean-Bernard Stefani (INRIA, France)
Richard Taylor (UCI, USA)
Martin Törngren (KTH, Sweden)
Stefan Van Baelen (K.U.Leuven DistriNet, Belgium)
Tullio Vardanega (University of Padua, Italy)
Eugenio Villar (Universidad de Cantabria, Spain)
Thomas Weigert (Missouri S&T, USA)
Tim Weilkiens (OOSE, Germany)
Sergio Yovine (VERIMAG, France)

This workshop is organised as an event in the context of

- The IST-004527 ARTIST2 Network of Excellence on Embedded Systems Design
- The research project EUREKA-ITEA SPICES (Support of Predictable Integration of mission Critical Embedded Systems)

Invited Talk

Semantics Preservation Issues in the Design and Optimization of SW Architectures for Automotive Systems

Marco Di Natale

Scuola Superiore Sant'Anna, Pisa, Italy

Abstract. Architecture selection and design optimization are critical stages of the Electronics/Controls/ Software (ECS) -based vehicle design flow. In automotive systems design, complex functions are deployed onto the physical HW and implemented in a SW architecture consisting of a set of tasks and messages.

The talk will present work performed in cooperation with GM R&D and UC Berkeley, in which we optimized several aspects of the software architecture design, including the definition of the task periods, the task placement and the signal-to-message mapping and we automated the assignment of priorities to tasks and messages in order to meet end-to-end deadlines and minimize latencies.

Architecture selection can be accomplished by leveraging worst case response time analysis within an optimization framework and we provide hints on how to use stochastic or statistical analysis to further improve the approach. However, current work has only scantly addressed the issues of preserving the semantics of functional models during implementation. Semantics preservation issues impose additional constraints on the optimization problem, but also reveal very interesting tradeoffs between memory and time/performance. In addition, the need to deal with heterogeneous models and standards (like AUTOSAR in the automotive business) further complicates the scenario.

SOPHIA: a Modeling Language for Model-Based Safety Engineering

Daniela Cancila[1], Francois Terrier[1], Fabien Belmonte[2], Hubert Dubois[1], Huascar Espinoza[1], Sébastien Gérard[1], and Arnaud Cuccuru[1]

CEA LIST[⋆], ALSTOM[⋆⋆]

Abstract. Development of increasingly more sophisticated safety-critical embedded systems requires new paradigms, since manual approaches are reaching their limits. Experiences have shown that model-driven engineering is an approach that can overcome many of these limitations. Using model-based approaches however lead to new challenges regarding the cohesive integration of both safety engineering and system design along the system development process. In this paper, we present SOPHIA, a modelling language that formalizes safety-related concepts and their relations with system modelling constructs. We particularly focus on accident models and on how to achieve confidence that the frequency of possible accidents will be tolerable. In addition, we explore some strategies to implement SOPHIA as a complementary modelling language to SysML and reuse some useful constructs form the UML MARTE profile.

1 Introduction

In order to cope with the increasing design complexity of safety-critical systems, safety assurance should be considered as early as possible in the design process. Among other goals, safety assurance allows achieving confidence that the frequency of accidents will be acceptable. For this purpose, safety engineers need to specify all possible safety parameters that directly impact the software architecture design, and then to determine the probability rates of the deviation from fulfilling the system functions. The *Safety Integrity Level* (SIL) attribute is an example of such a parameter. The design of a given system and its subsystems changes according to the value of the SIL associated with each functionality of the system. Possible values range between "0" (less critical) and "4" (most critical) [19]. Thus, a system architecture including SIL4-functionalities must guarantee the maximum level of safety integrity, which would for example imply to add redundant hardware nodes.

⋆ CEA LIST, Laboratoire d'Ingénierie dirigée par les modèles pour les Systèmes Embarqués Point Courrier 94, Gif-sur-Yvette, F-91191 France {`daniela.cancila, francois.terrier, hubert.dubois, huascar.espinoza, sebastien.gerard,arnaud.cuccuru`}`@cea.fr`

⋆⋆ Alstom Transport Information Solution 48 rue Albert Dhalenne, 93482 Saint-Ouen Cedex `fabien.belmonte@transport.alstom.com`

Recently, the railway safety community has proposed a new methodological guidance to enhance safety evaluation. In this proposal, SIL-to-function allocations exploit a new attribute named *Tolerable Accident Rate* (TAR). The TAR is defined as the "threshold between what is tolerable and what is undesirable with respect to the consequence of an accident" [8, 3]. In current industrial practice, the TAR is manually calculated, typically by using pre-defined tables. As the value of TAR influences the value of SIL, it may impact the architecture of a system. More precisely, the value of the TAR for a given accident (e.g. the head-on collision between trains) is used to calculate the value of the *tolerable hazard* for the same accident. Hence, we have a 1:1 correspondence between tolerable hazard and SIL. (We refer the reader interested in the technical details to [8, 3].)

Most of current practices on system safety assurance rely mainly on manual processes. They are therefore very dependent of the skill and experience of the engineers. This problem is exacerbated by the fact that safety engineering and software design domains have developed their own techniques and methodologies. Let us consider the example of the railway application domain. On the one hand, safety actors adopt standards that provide recommendations for safety assessment. Illustrative examples are fault tree analysis [22] and formal verification techniques, such as the B method [2]. On the other hand, actor from the software design and development community follow component-based techniques, such as [33, 9, 14]. In this context, defining the "right mapping" between safety models and models for software design/development is an essential challenge.

In order to avoid error-prone processes and to integrate both safety engineering and system design, we adopt a *model-based safety engineering* process. Model-Driven Engineering (MDE) [30, 31] is being successfully adopted in several industrial research projects [21, 18, 32, 29]. Two kinds of approaches are actually put into practice. In the first case, safety engineers and system designers share the same model of the system while using different views of it. In the second case, they use different models with clearly and formally defined relationships (using for example model transformation descriptors). In both cases, the direct benefits of MDE concern the possibility of automating part of the process of safety assurance, e.g. by automatically calculating certain information such as TAR parameters from the input safety parameters. This capability does not only simplify the process. It also enables to save time and, more importantly, it makes safety assurance as explicit part of an iterative design process. Indeed, new results can be more easily generated once the model has been changed. Moreover, the fact that models are more formally defined reduces the probability of introducing errors or omitting important details, since the analysis information is linked to the architectural model of the system.

As a main contribution within this paper, we introduce for the first time SOPHIA, a modelling language for safety concerns. SOPHIA provides an answer to safety industrial concerns by allowing designers to specify the safety attributes in a software design model. Our paper focuses on the infrastructure of SOPHIA, which is similar to that of MARTE [25]: It is based on a metamodeling, a profiling and a modeling space. As a result, SOPHIA has an independent language specification that can complement more general-purpose languages such as UML or SysML [24]. At the profile level, we propose to use some suitable concepts from MARTE, the OMG's UML profile for real-time embedded systems [25], in particular the *Value Specification Language* (MARTE::VSL).

In Section 2, we discuss some related works and we identify fundamental criteria and principles for model-based safety engineering. In Section 3, we explain our industrial motivations. We also provide a rationale for SOPHIA as well as a description of its Fundamental Concepts. Moreover, we investigate the strategies regarding the integration of safety and software design. Section 4 is the central part of the paper. For the first time, we discuss the whole structure of SOPHIA: from its Fundamental Concepts to the implementation. In Section 5, we compare our approach with those given in Section 2. Finally, conclusions and on-going works are presented.

2 Related Work

Integrating safety concerns in general-purpose modelling process is a big challenge that has been explored in many directions. In this section, we focus on a few works which are receiving specific attention in the MDE community.

In order to study dependability in AADL (Architecture Analysis & Design Language) [1], P. Feiler and al. introduce a framework to model the error state propagations in a hierarchical architecture [17]. Error propagation can occur at component level (by composition of the components), at the hardware level (by interconnecting processors) and between the hardware and the components ("due to their binding to the execution platform" [17]). In order to limit, or even avoid, the error propagation, the authors provide suitable filters (guards), for example between the interconnection of components. In [17], P. Feiler and al. addresses error modelling as a complementary view to system architecture, which is an important topic related to safety. However, it does not cope with the problem of accident case modelling and the specification of safety attributes such as the SIL.

In order to complement AUTOSAR (the European industrial standard to specify component-based software infrastructures in automotive applications [6]), some European industries and academics have defined an architecture description language, so called EAST-ADL [5]. This includes requirements modelling, feature content at the level of a vehicle description, architecture variability, functional structure of applications, middleware, plant (environment), abstract hardware architecture, and preliminary functional allocation. In addition, EAST-ADL enables the modelling of system failure behaviour and allows analysis of that behaviour using safety analysis tools. In particular, EAST-ADL aimed at using a safety design flow compatible with that defined by the upcoming ISO 26262 standard, including support for concepts such as hazards, safety goals and requirements, and the representation of ASILs (Automotive SILs). Many of these concepts were represented in the first version of EAST-ADL, but there were many others not considered, e.g. accident and its consequences, or ASIL decomposition.

FTA is one of the main safety analysis tools. In [10], Douglass introduces a UML safety profile defining notions such as fault, hazard, and traceability of requirements. Such notions allow us to create fault tree analysis (FTA) diagrams and, hence, to study how "conditions and faults combine to create hazard". One of the main contributions of this approach is to adopt UML and its profiling mechanism to provide a common specification language to integrate safety and design activities. This facilitates the collaboration and common understanding between safety engineering teams and system design teams. The underlying approach is the following. First, designers create a model with safety attributes, from which FTA is automatically generated. Engineers then study FTA and then they may manually change the model architecture. In other words, this approach does not deal with "safety reverse engineering". As a result, safety analysis is made *a posteriori*. When we deal with real industrial cases, a model quickly increases in complexity and in number of components. Consequently, it also occurs in the related FTA. The study of FTA is then a very complex work. In order to reduce such a complexity, one possible way is to *iteractivly* integrate safety engineering into model-based engineering of an architectural system. The underlying process is to have an automatic propagation of safety attributes in the architectural model such that it is correct with respect to "given safety requirements".

In [15], de Miguel and al. propose an approach similar to work [10]. Therefore, it has similar advantages and drawbacks. Finally, in [26], the authors introduces the UML profile for quality of service and fault tolerance analysis, called QoS&FT profile. In this profile, some aspects of safety analysis are covered (such that fault, errors, fault, non desirable events, etc). Notions, such as accidents and SIL are however not here considered.

3 Safety Engineering

This section provides some background information that have been taken into account for the definition of SOPHIA. Before describing the safety fundamental concepts, we want to discuss the high-level requirements for safety modelling from an industrial perspective.

Standards EN 50126 [11], EN 50128 [12] and EN 50129 [13] define a safety process plan for programmable electronic signalling devices including risk evaluation, SIL to function mapping and the life cycle recommendations by SIL. In particular, these standards recommend applying fully formal specification to ensure SIL 4. It means that engineers must provide mathematical proven demonstration for the safety properties of a given component.

Typically, in industry there is a gap between formal methods and textual system specifications, as well as between subsystem specifications. The main reason for this gap is that there is no standard and common language can be used to capture the different aspects. Semi-formal modelling approaches can bring a common basis to interconnect these different specification aspects. This is the main motivation for formalizing safety attributes into system models, from the early phases of the development process.

Therefore, SOPHIA has the following objectives:

1. enabling the specification of safety attributes in the architectural model of as sytem;
2. automating the calculation of some safety parameters in order to afford model-based engineering of safety;
3. providing an environment for system development in which coherence (compatibility of all requirements at the same level of abstraction, i.e., horizontal development) and correction ("good" decomposition of parent requirements into children requirements abstraction, i.e., vertical development) properties can be guaranteed by construction and/or verified a posteriori.

Provided these general needs, we present in the next section an excerpt of some fundamental concepts of SOPHIA related to accident case concerns

3.1 SOPHIA Fundamental Concepts

The concepts of SOPHIA (and their relationships) are based on Alstom Ontology [7] and on Alstom works such as [8, 3], which model their safety domain knowledge. In this section, we will use metamodels to describe the Fundamental Concepts of SOPHIA. They are organized into a set of packages and libraries. We use packages to introduce notions and their relationships, and we use libraries to specify data types. The package SOPHIA Fundamental Concepts contains two main subpackages, so called respectively SystemDesing and SafetyConcepts. Package SystemDesing specifies the relationships between safety concepts and model elements of a system. Package SafetyConcepts contains the following packages:

- **package ACCIDENTS,** which describes notions and relationships that are involved in an *accident.*
- **package MITIGATIONS,** which describes notions and relationships about mechanisms (*barriers*) to mitigate an *accident*;
- **package FaultContainmentRegion,** which describes notions and relationships that are involved in *error propagations.*

In this paper, we focus on package ACCIDENTS. Our intent is to show the details of the whole language design chain, from the formalization of the TAR attribute in the SOPHIA Fundamental Concepts, to the language implementation details. The result of our work is a first, but firm step towards model-based safety engineering.

Figure 1 shows some notions of package ACCIDENTS. Among them, we depict the TAR attribute. The notions are represented as metaclasses.

Hazard is “an event observable at the system boundary, which has potential either directly or in combination with other factors (external to the system), for giving rise to an accident at railway system level” [3].

AccidentCase is an unintended event with undesirable outcomes. AccidentCase leads to AccidentConsequenques. An AccidentCase is identified by the following properties: a unique ID; an AccidentType chosen from a statically pre-defined list; AutomaticTolerableAccidentRate and TolerableAccidentRate.

AutomaticTolerableAccidentRate is the maximum rate of occurrence that is tolerable for a likely Accident [3]. It is specified by a number of events per hour (real number) and it is derived from the frequency and the severity of an accident.

TolerableAccidentRate and AutomaticTolerableAccidentRate are similar properties. The only difference is that TolerableAccidentRate is manually set by safety engineers when they have to deal with exceptional cases (i.e., for which a pre-defined table is not available). In Figure 5, Table “a:” identifies the *Risk Tolerability* of an accident. It is described with combinations of the following properties: severity of the consequences and frequency of the accident. TolerableAccidentRate is undefined by default. However, if TolerableAccidentRate is set to a different value than undefined, then it has a higher priority with respect to AutomaticTolerableAccidentRate. The importance of having both properties (i.e., one automatically specified and the other one manually set), is that: 1.) the modelling process can be automated in a *correct* way that respects table *Risk Tolerability*, 2.) we have traced to the computation, which is automatically derived from table *Risk Tolerability.* Furthermore, we can identify the divergence points specified by the exceptional cases. In case of divergence, designers must motivate their choices with respect to the value automatically calculated from table *Risk Tolerability.* From an implementation standpoint, designers could motivate their decisions in a suitable dialog box. The implementation of this latter is part of our ongoing work.

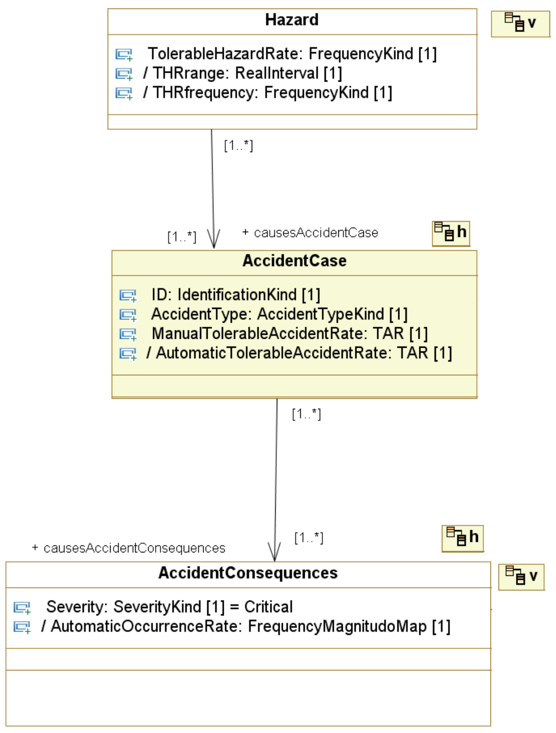

Hazard
TolerableHazardRate: FrequencyKind [1]
/ THRrange: RealInterval [1]
/ THRfrequency: FrequencyKind [1]
v
[1..*]
[1..*]
+ causesAccidentCase
h
AccidentCase
ID: IdentificationKind [1]
AccidentType: AccidentTypeKind [1]
ManualTolerableAccidentRate: TAR [1]
/ AutomaticTolerableAccidentRate: TAR [1]
[1..*]
+ causesAccidentConsequences
[1..*]
h
v
AccidentConsequences
Severity: SeverityKind [1] = Critical
/ AutomaticOccurrenceRate: FrequencyMagnitudoMap [1]

Fig. 1. SOPHIA : AccidentCase

AccidentConsequences is the result of a given AccidentCase. It is defined by the severity of the consequences with respect to the given AccidentCase. Severity may take only one of the following four predefined values: Catastrophic, Critical, Marginal, or Insignificant. These values of severity are captured by an enumeration which is part of our SOPHIA Fundamental Model library.

Next, we discuss the strategies to integrate the safety conceptual concepts defined above with a given general-purpose modelling language, in this case SysML.

3.2 Integration Strategies for SOPHIA and SysML

SysML was chosen by Alstom since it is an OMG standard specification for modelling of complex systems. Although SysML provides a formalism to manage requirements and system design together, SysML is lacking of concepts for dealing with specific concerns of safety. We have three possible strategies to integrate SOPHIA and SysML.

Strategy a defines SOPHIA as an extension to SysML. It has the advantage to be optimally tailored to the aimed integration with SysML. One of the main drawbacks of this strategy is that safety concepts will strongly depend on SysML. Then, any modification of SysML might lead to a modification in the SOPHIA extensions. In addition, safety concepts and SysML are conceptually disjoint, although complementary, and directly extending SysML does not make sense in our context.

Strategy b defines SOPHIA from scratch, i.e. as a pure domain-specific modeling specification language (DSML) (i.e. independently of UML) and then combining this metamodel with SysML. Consequently, *Strategy b* surmounts the drawbacks of *Strategy a*: safety concepts are independent not only of SysML but also of UML. It provides a framework that is fully dedicated to safety concepts and it is independent from other formalisms. As discussed in work [16], *Strategy b* has the following drawback: having safety models defined using two independent formalisms leads to strong difficulties for interfacing both types of models of the same system. This is particularly problematic for tracing safety information with the system architecture models. This problem is mainly reflected at tool level, since traceability always imply an important endeavour.

Strategy c proposes to firstly introduce SOPHIA as a package of Fundamental Concepts via a metamodel, in a way that is independent of the UML formalism. In a second stage, this metamodel (also called *domain model*) is implemented as a UML profile. In this way, we overcome the drawbacks of *Stategies a* and *b*, because the concepts are defined independently of UML, and, thereby, gain the benefits of a domain-specific approach. Moreover, this approach improves tool interoperability and facilitates the interface and traceability between different modelling aspects of the same system. SOPHIA and SysML languages (which are both designed as UML profiles) may indeed be used jointly in the same UML tool. A successful example of this approach is MARTE [25].

	Language Engeneering Domain		End User Domain	Tool environment
	Metamodel	Profile	One single Model	One single Tool
strategy a	NO	YES	YES	YES
strategy b	YES	NO	NO	NO
strategy c	YES	YES	YES	YES

Fig. 2. Strategies to integrate safety modeling language in the system architecture

4 From SOPHIA Safety Concepts to Implementation

4.1 SOPHIA Architecture

We adopt *Strategy c* and we develop it further. First af all, we strategically use the definition of profile, firstly introduced by S. Cook, and successfully adopted by other researchers: a profile is a family of related languages. It suggests the idea of exploiting the composition of pre-existing profiles. Indeed, our intent is not to define completely "new" metamodel and profile, covering all concepts from safety to architectural design. Our intend is indeed to reuse the existing work as much as possible, so that we can take advantage of pre-existing works and related tools.

In spite of SysML role for requirements and system's architecture (requirement and block diagrams), SysML lacks in the specification of temporal attributes [4]. Several European research projects are therefore willing to define a combined usage of both SysML and MARTE.

In the context of SOPHIA, we were particularly interested in MARTE to define non-functional properties (`MARTE::NFP`) and `MARTE::VSL` to valuate these properties. The `MARTE::NFP` package allows designers to annotate a UML model with non-functional properties. `VSL` stands for *Value Specification Language* and allows designers to specify "parameters/variables, constants, and expressions in textual form" [25]. Moreover, `VSL` supports arithmetic and logical expressions. Beyond the benefits of the SOPHIA alignment with a recognized international standard, the main advantage of this integration is the ability to support a well-formed syntax and semantics for safety parameters and to consequently enable automated derivation of dependent safety variables (See Section 4.2).

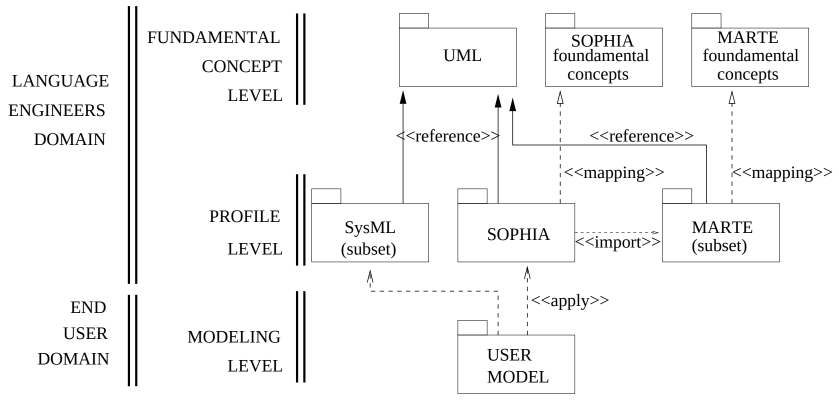


Fig. 3. Overall Structure

Figure 3 shows the overall structure. We have two main domains: end user domain and language engineering domain, which is in turn subdivided in two levels, Profile and Fundamental Concept. End user domain corresponds to M1 level in the OMG four-level hierarchy [23]. Designers only work in the modeling level. The language engineering domain corresponds to M2 level in the OMG four-level hierarchy. In the Profile level, we specify namesake profiles. In the Fundamental Concept level, we have UML metamodel and data (Fundamental Concepts for safety and MARTE in the figure). In the following, we discuss the overall structure, as illustrated in Figure 3.

At Modelling level, designers specify the model of a system. In order to specify the architecture of a system and associated requirements, designers need to apply SysML to their UML model. Next, designers annotate the model with safety attributes by applying the SOPHIA profile. In order to specify temporal attributes, designers exploit the MARTE stereotypes that are already imported by SOPHIA.

At Profile level, we have suitable languages of (at least) three families (profiles): SysML, SOPHIA and MARTE. One of our on-going works is to identify the minimum subset of SysML and MARTE to specify the requirements given by Alstom.

SysML is a UML profile. In Figure 3, UML stereotype "reference" shows such a relationship [27]. Note that SysML is only introduced as a UML extension and, then, SysML intentionally has not a fundamental concepts level.

SOPHIA is a UML profile for safety modelling modelling whose definition is based on SOPHIA Fundamental Concepts. In UML, there is not a specific symbol between a profile and its fundamental concepts. To explicitly capture this relationship, we use a dashed arrow annotated with the word "mapping", following the OMG notation introduced in work [28].

Like SOPHIA, MARTE extends UML and it is based on MARTE Fundamental Concepts, so-called MARTE Domain Model.

At fundamental concepts level, we have UML metamodels respectively denoting SOPHIA Fundamental Concepts and MARTE Fundamental Concepts.

4.2 SOPHIA, a UML profile for safety

In this section, we describe the UML profile for SOPHIA. It consists of a set of UML extensions and libraries concretized through stereotypes and data types. They map to the SOPHIA Fundamental Concepts (see Figure 3 for a big picture). Similarly to the SOPHIA Fundamental Concepts packages, the corresponding UML profile the profile is designed following a modular approach by grouping language constructs into individual packages, with the ability to select only those packages that are of direct interest in a given model. Due to space limitations, it is not possible to provide details covering the all profile. Therefore, we will focus on the SOPHIA package ACCIDENTS described in Section 3.1.

In the package ACCIDENTS every fundamental concept will directly result in a UML stereotype with its corresponding properties. In this case, there is a 1:1 mapping between the Fundamental Concepts and the profile element. The bottom package in Figure 4 defines how the metaclasses of the UML metamodel are extended with SOPHIA concepts, while the top-hand package shows a subset of the SOPHIA library with some enumeration types of interest.

Before describing the details of how SOPHIA exploits `MARTE::VSL`, let us introduce a real industrial railway example of risk assessment.

Example Figure 5 shows a typical example of risk assessment tables used in the railway domain. Such tables are used to identify the tolerable accident rate (TAR) of a given accident case (stereotype AccidentCase in Figure 4) and the occurrence rates of the different consequences of an accident case (stereotype AccidentConsequences in Figure 4). Typically, these tables are standardized by the territory authorities. For the sake of simplicity, we focus on the calculation of the TAR and the consequence occurrence rate parameters for a given country.

Starting from "Table a:" of Figure 5, safety engineers define a severity level for every accident case. This information allows for identifying the threshold of the accident case risk. (annotated with "T" in the figure.) The threshold risk identifies the upper limit of a tolerable risk. For instance, let us consider a Critical severity level. The corresponding threshold risk can be identified in the fourth row of the Critical column. This corresponds to the Undesirable risk level. The obtained threshold risk level can then be used to identify a corresponding threshold frequency of the accident case. In our example, such frequency level is Remote. This yields an input value for "Table b:".

"Table b:" describes a mapping of frequencies of accident cases and numerical information about the magnitude order of such frequencies. This magnitude order is specified as an interval of real numbers. The lower bound value of this interval, corresponding to the threshold frequency level of a given accident case, represents the

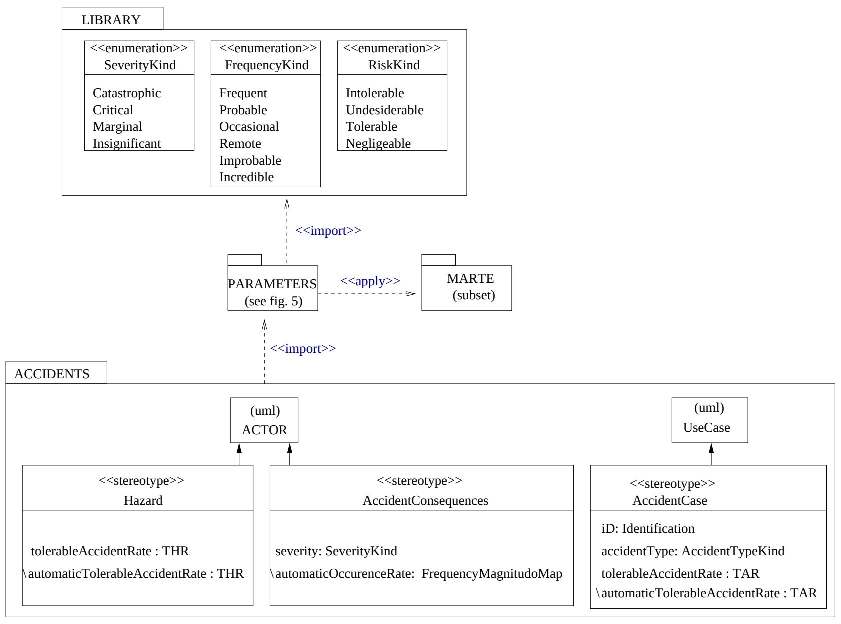


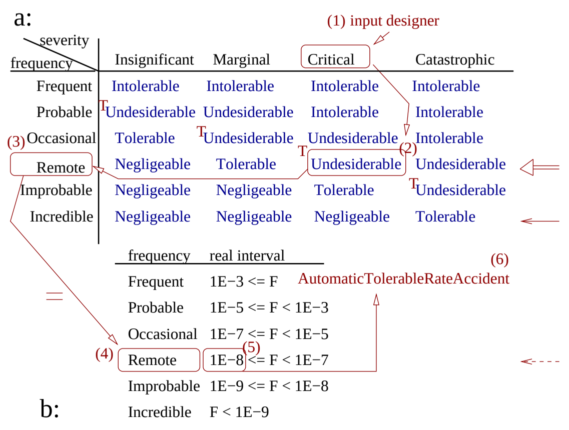
Fig. 4. SOPHIA: focus on TAR

TAR value for this accident case. In Figure 5, the TAR value the studied accident case is 1x10-8.

Figure 6 shows the model representation of "Table a:". In a:RiskTolerabilityAccident, each line of attribute RiskMapping represents a line of "Table a:". Consider "Table a:". We can read it as: we taken two values, one for colomn and one for row, then, we uniquely identify one cell, which contains the value of the risk. For example, for the colomn we select severity $= critical$ and, for the row, frequency $= Incredible$ Hence, we achieve a unique cell, which contains risk $= Negligeable$. The first line in Figure 6 represents the list of these attributes and their values. In instance a:RiskTolerabilityAccident, each line is given by the above procedure. In order to model the threshold between what is tolerable and what is undesiderable (noted by "T" in Figure 5), we introduce the Boolean attribute isThreshold in Figure 6. Therefore, if severity $= critical$, then risk $= Undesiderable$, because isThreshold $= True$.

a:RiskTolerabilityAccident is an instance of class RiskTolerabilityAccident, which contains one attribute riskMapping of type RiskMappingType. We stereotype RiskMappingType with VSL::TupleType. As might be expected, the `VSL` package (which contains a set of stereotypes extending the data type of UML) is applied to some of the SOPHIA data types. By definition, a TupleType is a data type that combines different types into a single aggregated type [25]. This allows instances of these tuple types to be annotated as composite values following the textual syntax defined for VSL tuple specifications.

Similarly to Table "a:", we represents "Table b:" by Figure 7, where some predefined MARTE data types are imported and reused in SOPHIA constructs. For instance, RealInterval (a MARTE's data type stereotyped VSL::IntervalType) is typing the magnitudoOrder property of FrequencyMapType. This allows instances of IntervalType to



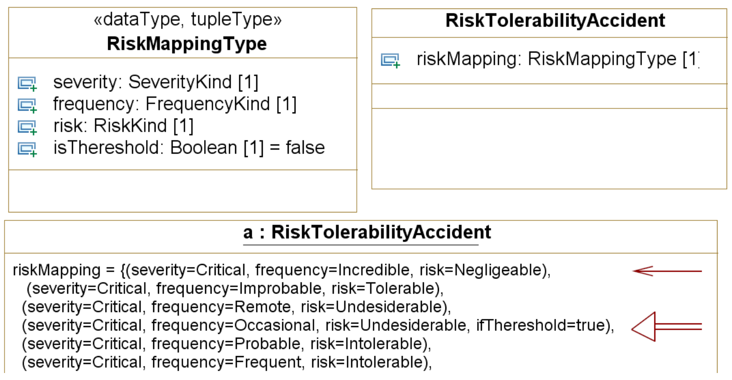
a:

severity / frequency	Insignificant	Marginal	Critical	Catastrophic
Frequent	Intolerable	Intolerable	Intolerable	Intolerable
Probable	Undesiderable	Undesiderable	Intolerable	Intolerable
Occasional	Tolerable	Undesiderable	Undesiderable	Intolerable
Remote	Negligeable	Tolerable	Undesiderable	Undesiderable
Improbable	Negligeable	Negligeable	Tolerable	Undesiderable
Incredible	Negligeable	Negligeable	Negligeable	Tolerable

b:

frequency	real interval
Frequent	1E−3 <= F
Probable	1E−5 <= F < 1E−3
Occasional	1E−7 <= F < 1E−5
Remote	1E−8 <= F < 1E−7
Improbable	1E−9 <= F < 1E−8
Incredible	F < 1E−9

Fig. 5. Table Risk Tolerability and Table Frequency



«dataType, tupleType»
RiskMappingType

severity: SeverityKind [1]
frequency: FrequencyKind [1]
risk: RiskKind [1]
isThereshold: Boolean [1] = false

RiskTolerabilityAccident

riskMapping: RiskMappingType [1]

a : RiskTolerabilityAccident

riskMapping = {(severity=Critical, frequency=Incredible, risk=Negligeable),
(severity=Critical, frequency=Improbable, risk=Tolerable),
(severity=Critical, frequency=Remote, risk=Undesiderable),
(severity=Critical, frequency=Occasional, risk=Undesiderable, ifThereshold=true),
(severity=Critical, frequency=Probable, risk=Intolerable),
(severity=Critical, frequency=Frequent, risk=Intolerable),

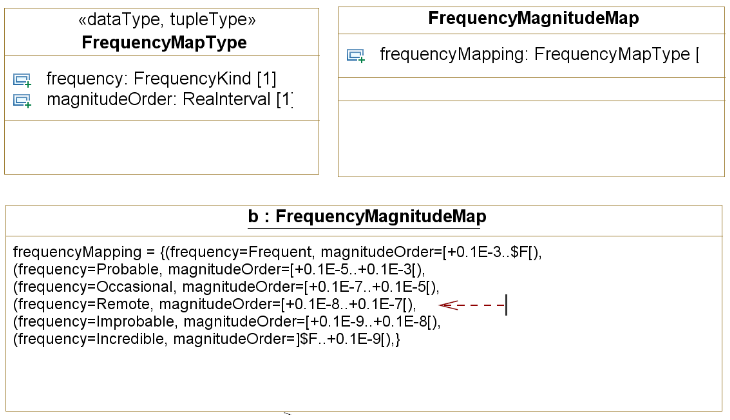
Fig. 6. package PARAMETERS: SOPHIA and MARTE for "Table a:"

be specified with the VSL syntax for interval values, as depicted in b:FrequencyMagnitudoMap.

Algorithm to calculate the TAR parameter: In the sequel, we describes the algorithm used to derivate the TAR parameter.

```
    GLOBAL VAR RiskTolerableAccident : ARRAY[SEVERITY_KIND][FREQUENCY_KIND]:
[risk:RISK_KIND,IsThereshold:BOOLEAN];
    GLOBAL VAR FrequencyMagnitudeMap: ARRAY[FREQUENCY_KIND]: REAL_INTERVAL;
            AutomaticTARCalculate (UserSeverityValue:SEVERITY_KIND): TAR;
            VAR MyFrequency: FREQUENCY_KIND;
            VAR MyInterval: REAL_INTERVAL;
            VAR j: INTEGER;

            j := 0;
```


«dataType, tupleType»
FrequencyMapType
frequency: FrequencyKind [1]
magnitudeOrder: RealInterval [1]
FrequencyMagnitudeMap
frequencyMapping: FrequencyMapType [
b : FrequencyMagnitudeMap

```
frequencyMapping = {(frequency=Frequent, magnitudeOrder=[+0.1E-3..$F[),
(frequency=Probable, magnitudeOrder=[+0.1E-5..+0.1E-3[),
(frequency=Occasional, magnitudeOrder=[+0.1E-7..+0.1E-5[),
(frequency=Remote, magnitudeOrder=[+0.1E-8..+0.1E-7[),
(frequency=Improbable, magnitudeOrder=[+0.1E-9..+0.1E-8[),
(frequency=Incredible, magnitudeOrder=]$F..+0.1E-9[),}
```

Fig. 7. package PARAMETERS: SOPHIA and MARTE for “Table b:”

```
WHILE (RiskTolerableAccident [UserSeverityValue][j].IsThereshold ⟨⟩ TRUE)
        DO j := j+1;
MyFrequency := FREQUENCY_KIND[j];
MyInterval := FrequencyMagnitudeMap[MyFrequency];

 RETURN AutomaticTARCalculate := MyInterval.LowerBound;
```

Although it is written in pseudo-code, it can be implemented in Java code and easly introduced in a static profile implementation of SOPHIA.

5 Discussion

In Section 2, we have discussed some works on safety that have a great impact in the MDE community. Most of them provide both a way to specify the safety attributes in the design model, and tool support for safety analysis. Often, we have faced on two different models: one for safety and another one for design modelling. The focus is then on the “right mapping” and in the a-posteriori verification of the safety attributes.

SOPHIA is based on four capital pillars:

– SOPHIA Fundamental Concepts;
– reuse of pre-existing profiles (and then their tool support);
– automation on the propagation of the safety attributes in the design model;
– a-priori verification of the safety attributes in a correct way regarding to pre-defined risk tables.

In the sequel, we discuss each SOPHIA pillar with respect to some of the works presented in Section 2.

Altough SOPHIA is a UML profile, SOPHIA Fundamental Concepts have been created as free as possible from considerations related to specific solution technologies so as to not embody any premature decisions that may hamper later language use. This means that the fundamental concepts model can be concretized not only as a

UML profile, but also as an independent modelling language, possibly implemented as an Ecore metamodel or an XML schema, as well. Note that, although the SOPHIA Fundamental Concepts are specified in the form of a metamodel with a textual semantic description (like in MARTE), it represents only conceptualization entities synthesizing the "universe of discourse". This pillar is similar to that presented in work [15] in which the authors first define a safety conceptual model of safety-aware component-based architectures and just then define a safety UML profile.

The second pillar introduces SOPHIA as a UML profile, by adopting the definition of profile firstly given by S. Cook. As a result, SOPHIA profile strategically reuses some packages of MARTE and can be easily integrated in a SysML system architecture.

The third pillar put the strength in improving the automation of the modelling process. In particular, SOPHIA provides a framework to automatically generate the value of TAR and the frequency of an accident, from the specification of only one attribute by users, which is the severity attribute of a consequence. This attribute is given by engineers by choosing one of four possible values.

Finally, the fourth pillar's objective is to enable safety ensurance calculation along the development process, in a way that is correct with respect to pre-defined tables.

6 Conclusions

In this paper, we present for the fist time SOPHIA, a *model-based safety engineering* approach. SOPHIA responds to industrial needs regarding the integration of safety engineering and system design. SOPHIA provides a metamodeling and profiling infrastructure to specify and propagate the safety information on design models. We have particularly focused on the TAR calculation, which is the first step of the risk evaluation of an accident. The result of some safety attributes, such as TAR, influences the SIL and, hence, changes the model architecture. Such safety information is *a-priori correct* regarding to pre-defined risk tables. Currently we are performing tests on industrial real cases. We are applying the same process (as discussed for TAR) to other safety attributes. We also intend to mathematically formalize correctness of the automatic propagation of the safety attributes in the design model.

Acknowledgment

This work has been performed in the context of the IMOFIS project of the System@tic Paris Région Cluster. It is sponsored by the "Safe, reliable and adapted transportation" program (PREDIT) of the "Agence Nationale pour la Recherche". The authors would like to thank the all member of the IMOFIS project [20] and the reviewers of ACES^{MB} Workshop for their valuable suggestions.

References

1. AADL. Architecture Analysis & Design Language. `www.aadl.info/aadl/currentsite/index.html`.
2. J. R. Abrial. *The B-book: assigning programs to meanings.* Cambridge University Press, 1996.

3. Alstom. Guidance for safety analysis. MODTRAIN, MODCONTROL Sub-Project, 2008.
4. C. André. Time Modeling in MARTE. In *FDL'07 Forum on specification and Design Languages*, Barcelona, Spain, 2007.
5. ATESST Project. Advancing Traffic Efficiency and Safety through Software Technology. ATESST STREP - FP6 project. `http://www.atesst.org`.
6. AUT@SAR. Automotive Open System Architecture. `www.autosar.org`.
7. F. Belmonte. T1.1 guide de modélisation. Projet IMOFIS, Alstom Transport, System@tic, 2009.
8. A. Blas and J. L. Boulanger. Comment améliorer les méthodes d'analyse de risques et l'allocation des THR, SIL et autres objectifs de sécurité. In *Lambda-Mu, 16e Congrès de Maîtrise des Risques et de Sûreté de Fonctionnement*, Avignon, France, 2008.
9. S. Bliudze and J. Sifakis. The Algebra of Connectors - Structuring Interaction in BIP. In *Int. Conf. EMSOFT*, pages 11–20, 2007.
10. B.P. Douglass. Build Safety-Critical Designs with UML-based Fault Tree Analysis-Defining a Profile. `www.embedded.com/design/opensource/217200312?pgno=1`.
11. CENELEC. EN-50126: Application ferroviaires -Spécification et démonstration de Fiabilité, Disponibilité, Maintenabilité et Sécurité (FMDS). Norme, CENELEC, 1999.
12. CENELEC. EN-50128: Applications ferroviaires - Système de signalisation, de télécommunication et de traitement - Logiciels pour systèmes de commande et de protection ferroviaire. Norme, CENELEC, 2001.
13. CENELEC. EN-50129: Application ferroviaires - système de signalisation, de télécommunication et de traitement - systèmes électroniques relatifs à la sécurité pour la signalisation. Norme, CENELEC, 2001.
14. L. de Alfaro and T. A. Henzinger. Interface automata. In *Proceedings of the Ninth Annual Symposium on Foundations of Software Engineering*, pages 109–120. ACM Press, 2001.
15. M. de Miguel, J. Briones, J. Silva, and A. Alonso. Integration of satety analysis in model-driven software development. 2008.
16. H. Espinoza, B. Selic, D. Cancila, and S. Gérard. Challenges in Combining SysML and MARTE for Model-Based Design of Embedded Systems. In *In Proc. of Int. Conf. on Model Driven-Architecture Foundations and Applications (ECMDA 09)*, volume 5562. LNCS, 2009.
17. P. Feiler and A. Rugina. Dependability Modeling with the Architecture Analysis & Design Language (AADL). Technical report, Software Engineering Institute, Carnegie Mellon, 2007.
18. B. Hamid, A. Radermacher, A. Lanusse, C. Jouvray, S. Gerard, and F. Terrier. Designing fault-tolerant component based applications with a model driven approach. In *IFIP Workshop on Software Technologies for Future Embedded and Ubiquitos Systems (SEUS 2008)*, Springer LNCS.
19. IEC. *61508:1998 and 2000, part 1 to 7. Functional Safety of Electrical, Electronic and Programmable Electronic Systems.*, 2000.
20. IMOFIS Project. Ingénierie des MOdèle de FonctIons Sécuritaires. `www.imofis.org/`.
21. E. Jouenne and V. Normand. Tailoring IEEE 1471 for MDE Support. In *UML Modeling Languages and Applications*, LNCS, Springer, 2005.
22. N. Limnios. *Fault trees.* ISTE, 2007.
23. OMG. http://www.omg.org/.

24. OMG. Systems Modeling Language SysML. `www.sysml.org`.
25. OMG. UML Profile for MARTE: Modeling and Analysis of Real-Time Embedded systems, Beta 3. `www.omgmarte.org`.
26. OMG. UML Profile for Modeling Quality of Service and Fault Tolerance Characteristics & Mechanisms (QoS & FT profile). `www.omg.org`.
27. OMG. Unified Modeling Language (UML) Specification: Infrastructure. Version 2.0. `www.uml.org`, 2004.
28. OMG. UML Profile for Schedulability, Performance, and Time Specification. `www.uml.org`, 2005.
29. F. Ougier and F. Terrier. ADONA: an open Integration Platform for Automative Systems Development Tools. In *Euopean Congress Embedded real Time Software (ERTS)*, 2008.
30. D. Schmidt. Model-driven engineering. *IEEE Computer*, pages 25–31, February 2006.
31. B. Selic. From Model-Driven Development to Model-Driven Engineering. Keynote talk at ECRTS'07. `http://feanor.sssup.it/ecrts07/keynotes/k1-selic.pdf`.
32. F. Terrier and S. Gerard. MDE benefits for distributed, real-time and embedded systems. In *From Model-Driven Design to Resource Management for Distributed Embedded Systems, IFIP TC 10 Working Conference on Distributed and Parallel Embedded Systems (DIPES 2006)*, 2006.
33. Veryard Projects. Component-based Development FAQ. `http://www.users.globalnet.co.uk/~rxv/CBDmain/cbdfaq.htm`, February 2008.

PaNeCS: A Modeling Language for Passivity-based Design of Networked Control Systems

Emeka Eyisi, Joseph Porter, Joe Hall, Nicholas Kottenstette, Xenofon Koutsoukos and Janos Sztipanovits

Institute for Software Integrated Systems
Vanderbilt University
2015 Terrace Place, Nashville, TN 37203 USA
emeka.p.eyisi@vanderbilt.edu

Abstract. The rapidly increasing use of distributed architectures in constructing real-world systems has led to the urgent need for a sound systematic approach in designing networked control systems. Communication delays and other uncertainties complicate the development of these systems. This paper describes a prototype modeling language for the design of networked control systems using passivity to decouple the control design from network uncertainties. The modeling language includes an integrated analysis tool to check for passivity and a code generator for simulation in MATLAB/Simulink using the TrueTime platform modeling toolbox. The resulting designs are by construction more robust to platform effects and implementation uncertainties.

1 Introduction

The heterogeneous composition of computing, sensing, actuation, and communication components has enabled a modern grand vision for real-world Cyber Physical Systems (CPS). Real-world CPSs such as automotive vehicles, building automation systems, and groups of unmanned air vehicles are monitored and controlled by networked control systems (NCS). NCS involve the interaction of physical dynamics, computational dynamics, and communication networks. This heterogeneity does not go well with current methods of compositional design. The most important principle used in achieving compositionality is separation of concerns which works if the design views are orthogonal, i.e. design decisions in one view do not influence design decisions in other views. Unfortunately, achieving compositionality for multiple physical and functional properties simultaneously is a very hard problem because of the lack of orthogonality among the design views.

Model-based design for embedded control systems involves creating models and checking correctness at different stages in the development process [1]. Model-based design flow progresses along precisely defined abstraction layers, typically starting with control design followed by system-level design for the specification of platform details, code organization, and deployment details, and the final stage of integration and testing on the deployed system. This design approach cannot be applied directly to NCS because domain heterogeneity and tight coupling between design concerns create a number of challenges. Ensuring controller stability and performance for physical systems in the presence of network uncertainties (e.g. time delay, packet loss) couples the control and system-level design layers. In addition, downstream code modifications during testing and debugging invalidate results from earlier design-time analysis and any component change often results in "restarting" the design process.

A number of research projects seek to address the problems of model-based design for NCS. The ESMoL modeling language for designing and deploying time-triggered control systems explicitly captures in model structure many of the essential relationships in an embedded design[2]. The ESMoL tools include schedule determination for time-triggered communications, code generation, and a portable time-triggered virtual machine. AADL [3] is a textual language and standard for specifying deployments of control system designs in data networks [4]. AADL projects also include integration with verification and scheduling analysis tools. The Metropolis modeling framework [5] aims to give designers tools to create verifiable system models. Metropolis integrates with SystemC, the SPIN model-checking tool, and other tools for scheduling and timing analysis.

In order to tackle the challenges of designing NCS, we propose an automated model-based approach based on passivity control theory. We used Model-Integrated Computing [1] to develop a domain specific modeling language (DSML) called the Passive Network Control Systems language (PaNeCS). Our approach is based on the passive control architecture presented in [6] which provides the theoretical foundations for analysis and design of NCS emphasizing robustness to network delays and packet loss. This paper focuses on the design of the DSML as well as a compositional tool for passivity analysis and code generation for Matlab/Simulink/Truetime models. We aim to address a number of significant challenges:

- Changes made during design, development, and testing cycles may cause extensive software revisions and force expensive re-verification. Model-integrated computing tools provide automated software generation, analysis, and system configuration directly from models. PaNeCS supports forward generation of platform-specific simulation models as well as passivity analysis of system components.
- Control systems are often verified using complex optimization techniques. For example, linear matrix inequalities (LMIs) can model many important controller properties (e.g. stability, response time, reachability). In a system built from the composition of multiple blocks, such analysis quickly becomes intractable. In order to assess global stability, designers would have to build a single, large analysis model which includes all possible state variables in the system. In contrast, the passive control architecture can ensure global stability (in a robust way) by a combination of component analysis and specific rules for composition of passive components.

– Control designers create models for both physical systems and controllers using tools like Simulink and Stateflow [7]. Deployment of a control design such as a Simulink model to a networked architecture introduces uncertainties due to time-varying delay, data rate limitations, jitter, and packet loss. Deployment of the design is often expensive, and failure during testing can be costly. An increasingly accepted way to address these problems is to enrich abstractions in each layer with implementation concepts. An excellent example for this approach is TrueTime [8] that extends Simulink with platform-related modeling concepts (i.e., networks, clocks, schedulers) and supports simulation of networked and embedded control systems with the modeled implementation effects. While this is a major step in improving understanding of implementation effects, it does not help in decoupling design layers and improving orthogonality across design concerns. A control designer can factor in implementation effects (e.g., network delays), but if the implementation changes the controller may need to be redesigned. Our approach imposes passivity constraints on the component dynamics, so that the design becomes insensitive to network effects, thus establishing orthogonality (with respect to network effects) across the controller design and implementation design layers.

The paper is organized as follows: Section 2 presents a passive control architecture for NCS. Section 3 presents our prototype modeling language. Section 4 discusses an integrated analysis tool for automatically checking passivity. Section 5 presents a model interpreter for generating Matlab/Simulink simulation code using the TrueTime platform modeling toolbox. Section 6 shows a case study of a NCS consisting of two discrete plants and a controller. Section 7 provides our conclusion.

2 Passivity-Based Control of Networked Control Systems

Our approach for designing NCS is based on passivity theory. There are various precise mathematical definitions for passive systems [9]. Essentially all definitions state that the output energy must be bounded so that the system does not produce more energy than was initially stored. Passive systems have a unique property that when connected in either a parallel or negative feedback manner the overall system remains passive. Passivity provides an inherent safety – passive systems are insensitive to certain implementation uncertainties [10] [11][12], so passivity can be exploited in the design of NCS. The main idea is that by imposing passivity constraints on the component dynamics, the design becomes insensitive to network effects, thus establishing orthogonality (with respect to network effects) across the various design layers. This separation of concerns allows the model-based design process to be extended to networked control systems which is what our model-based approach provides.

We briefly discuss the passivity based control architecture for multiple plants controlled by a single controller via a network [6]. Fig. 1 depicts a sample networked control system where only one plant is shown. The Bilinear Transform

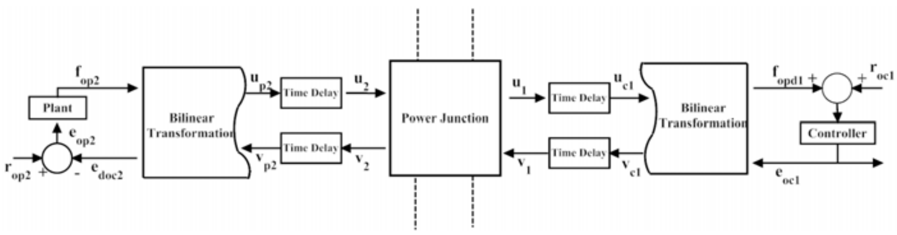


Fig. 1. A networked control system

block represents a transformation between signals and wave variables. Wave variables were introduced by Fettweis in order to circumvent the problem of delay-free loops and guarantee a realizable implementation for digital filters [10]. Wave variables also allow systems to remain passive while transmitted data over a network subject to arbitrary fixed time delays and data dropouts [11], [12]. In Fig. 1, $u_{pk}(i)$ (k=1,2), can be thought of as sensor output data in wave variable form from each plant. Likewise, $v_{cj}(i)$ (where j=1,2) can be thought of as a command output in wave variable form from the controller.

The power junction in Fig. 1 is an abstraction used to interconnect wave variables from multiple controllers and multiple plants in parallel such that the total input power is always greater than or equal to the total output power. This provides a formal way to construct a networked control system. The power junction makes it possible for a single controller to control multiple plants over a network and guarantee that the overall system remains stable. A detailed mathematical definition of the power junction can be found in [6]. In Fig. 1, the power junction has waves entering and leaving as indicated by the arrows. The waves entering the power junction from the controller are the network-delayed version of the waves leaving the controller, as indicated by the time delay block. Also, the waves entering the controller are the delayed version of the waves leaving the power junction. Likewise, the waves entering the plant are the delayed version of waves leaving the power junction and the waves entering the power junction are the delayed version of waves leaving the plant.

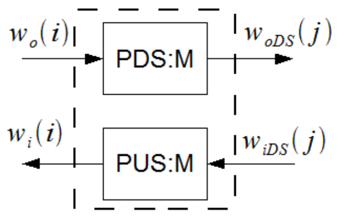


Fig. 2. The passive upsampler and passive downsampler.

Due to bandwidth constraints, the controller typically runs at a slower rate than the sensors and actuators of the plants. In order to preserve passivity in the multi-rate digital control network we use the passive upsampler (PUS) and passive downsampler (PDS) pair to handle the data rate transitions. Fig. 2 depicts the passive upsampler (PUS) and passive downsampler (PDS). $w_o(i)$ denotes a discrete wave variable going out of a wave transform block. For example, in Fig. 1, $v_{c1}(i) and\ u_{p2}(i)$, the wave variables going out of the Bilinear Transformation block, are each connected as $w_o(i)$ to their respective downsampler blocks. Similarly, $w_i(i)$ represents the respective discrete wave variable going into a wave transform block. $u_{c1}(i)$ and $v_{p2}(i)$, correspond to the $w_i(i)$ connections in Fig. 2. The PUS and PDS provide the upsampled and downsampled versions of their respective wave variable inputs while preserving passivity. The block parameter M is the sampling ratio – the data rate of the fast side of the connection divided by the data rate on the slow side.

3 PaNeCs

We introduce the passivity-based modeling language (PaNeCS). The modeling language is developed using a meta-configurable tool, the Generic Modeling Environment (GME), from the Model Integrated Computing (MIC) tool suite [13]. GME provides a metamodeling environment similar to UML. The class stereotypes are defined as follows: Models are entities that may contain other objects while Atoms are indivisible entities which cannot contain other objects; Connections are association classes used to describe the relationship between two entities. It represents a line that connects two entities of a model. Connectors signified by "**.**" specify a visualization for a connection in the model. Associations to the connector have possible roles ("src" and "dst") to define the allowed direction of a connector.

3.1 Components

The language top level consists of four main components: the **PlantSystem**, the **ControllerSystem**, the **PowerJunction** and the **WirelessNetwork**.

PlantSystem Fig. 3 shows a part of the PaNeCS metamodel that describes the plant subsystem. *Plant* represents a model for any discrete linear time-invariant (LTI) system and can be extended to a nonlinear system. The dynamics of the *Plant* are represented by the following state space equations:

$$\begin{aligned} x(k+1) &= Ax(k) + Bu(k) \\ y(k) &= Cx(k) + Du(k). \end{aligned} \tag{1}$$

The *Plant* dynamics are parameterized by matrix attributes A, B, C, D, and a scalar *SamplingTime*. The attributes can be specified using any valid Matlab expression that evaluates to the proper dimensions. *BilinearTransformP* represents a model for the wave scattering technique for transforming the wave variables received from the power junction into control input to the plant and for transforming the plant output signal into wave variables that are transmitted over the network. *PassiveUpSampler and PassiveDownSampler* pair represent the PUS and PDS pair discussed in Section 2.

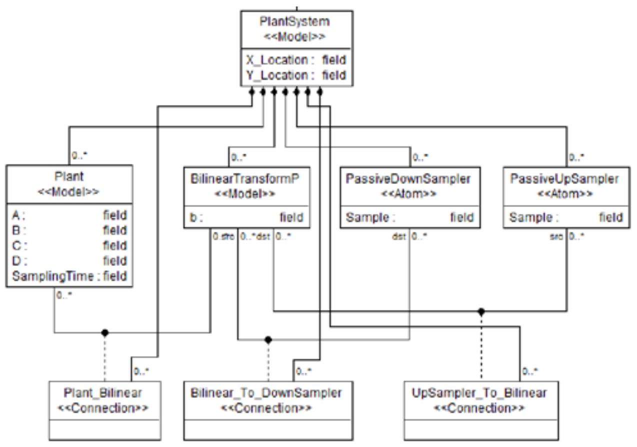


Fig. 3. PlantSystem portion of the Metamodel

ControllerSystem Fig. 4 shows the part of the language that describes the controller subsystem. *DigitalController* is a model representing the algorithm

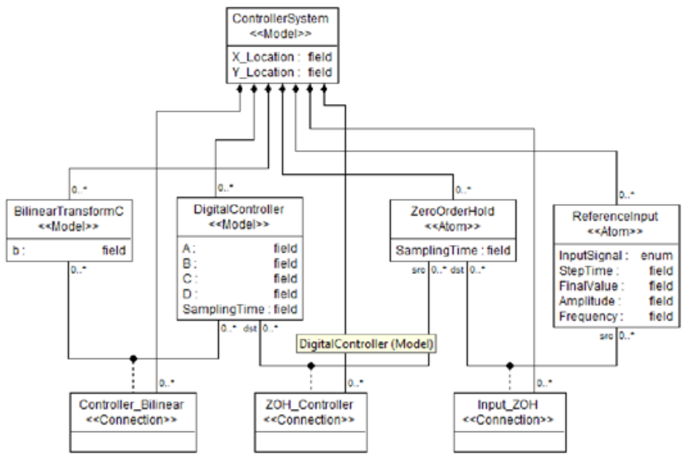


Fig. 4. ControllerSystem portion of the Metamodel

for controlling the networked plants. Similar to the model of the *Plant* in the **PlantSystem**, the *DigitalController* is modeled as a LTI system and its dynamics can also be represented in the state space form of Eq. (1). Therefore, the *DigitalController* parameters have similar attributes to the *Plant*. *BilinearTransformC* is similar to the *BilinearTransformP* described in the **PlantSystem**. *ZeroOrderHold* represents a component that holds its input for the time period specified in the sampling time attribute. *ReferenceInput* represents the desired signal to be tracked by the plants.

Power Junction Fig. 5 shows the part of the language that describes the power junction. The PowerJunction can contain ports for the connection of the

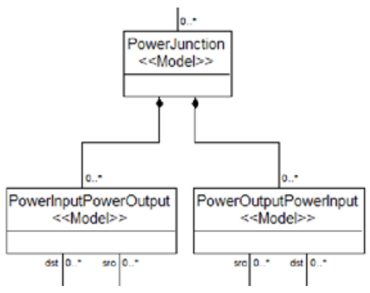


Fig. 5. PowerJunction portion of the Metamodel

plants and controllers. They are briefly described as follows: *PowerInputPowerOutput* represents a port through which the **PlantSystem** connects to the **PowerJunction**. Through it, the **PowerJunction** sends calculated control signals to the **PlantSystem** and also receives sensor signals from the **PlantSystem**. *PowerOutputPowerInput* represents a port through which the **ControllerSystem** can connect to the **PowerJunction**. Through it, the **PowerJunction** sends the averaged sensor signal to the **ControllerSystem** and receives the calculated control signal from the **ControllerSystem**.

WirelessNetwork Fig. 6 represents the network and its parameters for the NCS. The **WirelessNetwork** model provides modifiable parameters for simu-

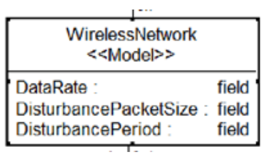


Fig. 6. Wireless Network portion of the Metamodel

lation. *Data rate* sets the throughput for simulating network activity. *DisturbancePacketSize* configures the size of simulated disturbance attack packets on the network (introduces delays). This provides a way for simulating the NCS under non-optimal conditions. *DisturbancePeriod* configures the frequency of disturbance attacks on the network.

3.2 Language Aspects

Our modeling language has two aspects (GME aspects are similar to modeling views in other tools): **Control Design Aspect** and **Platform Aspect**. The

Control Design Aspect visualizes the controller modeling layer. This includes the plants, controller, and power junction, as well as their interconnections – indicating the flow of control and sensor signals.
The **Platform Aspect** visualizes the physical platform layer. This model view shows the physical components of the NCS. The entities in this view include the plants, controller, and the wireless network as well as their interconnections indicating the flow of data packets over the network. Though the plants and controller appear in both aspects, in the Platform aspect they represent physical entities rather than control design concepts.

3.3 Structural Semantics

The main objective of our language design is to ensure the "correctness-by-construction" for passive designs of NCS designed using PaNeCS. In order to achieve this objective, we impose constraints on the properties of components of NCS as well as their interconnections. The metamodel notations described above does not capture all the required structural constraints. Using the Object Constraint Language (OCL), we can describe well-formedness rules for defining precise control of static semantics of the language. GME is embedded with an OCL engine which can be used to define constraints that are enforced at design time, giving direct feedback when the user attempts to create faulty connections in the model or violates any of the specified constraints. In Section 4, we will describe an analysis tool that is used to verify that system components satisfy the component-level passivity constraints.

We implemented three classes of constraints: Cardinality Constraints, Connection Constraints and Unique Name Constraints. *Cardinality Constraints* ensure that the required and correct number of components are used in the NCS design. For example, for each **PlantSystem** model there must be one *Plant*. *Connection Constraints* restrict the number of allowable connections between components. For example, in the **PlantSystem** model there can be only one bidirectional connection between the *Plant* and *BilinearTransformP*. *Unique Name Constraints* ensure the uniqueness of the names of components in the Plant and Controller subsystems as well as in the top level model of the NCS.

An example of an OCL constraint implementation is shown below. This specifies that the number of allowable connections from a BilinearTransformC model to a DigitalController to be one.

```
Description: There must be only one bidirectional connection
between BilinearTransformC to the DigitalController

Equation: let dstCount = ...
 self.attachingConnections("src",Controller_Bilinear)->size in
 dstCount <> 0 implies dstCount = 1
```

3.4 Operational Semantics

NCS modeled in PaNeCS are implemented in MATLAB/Simulink using models generated by an integrated code generator which is discussed in Section 5.

The *Plant* and *DigitalController* entities are implemented as Simulink blocks that model the behavior of each entity based on user-specified parameters. The **PlantSystem** and **ControllerSystem** are modeled as Simulink subsystems. In order to model the behavior of a real network, Simulink is extended with TrueTime as described previously.

Each **PlantSystem** and **ControllerSystem** are connected to a TrueTime Kernel block. The TrueTime Kernel essentially represents each subsystem as a node in the network. It is responsible for I/O and network data acquisition as well as implementing other user-defined tasks, and models the computer or processor on which the subsystem is implemented. The task in each TrueTime kernel connected to a **PlantSystem** is performed periodically based on the specified sampling time of the subsystem. For this version of our language, the **PowerJunction** is implemented as a task in the TrueTime Kernel connected to the **ControllerSystem**. The task that implements the power junction operates based on the occurrence of an event such as the arrival of sensor data or control signal. Each TrueTime kernel has two main scripts: 1) The Initialization script specifies the number of inputs and outputs, the function code name and also indicates the kernel's node id which is used to identify the kernel on the network. 2) The function script essentially implements the user specified task such as sending and receiving of wave variables over a network. The function code for the TrueTime kernel connected to the **ControllerSystem** also performs the additional task of implementing the **PowerJunction**. Hence, the **PowerJunction** sends and receives wave variables from the **ControllerSystem** locally while the **PlantSystem** sends and receives wave variables from the power junction over the simulated wireless network.

The wireless network is implemented using the TrueTime Wireless network block. It simulates the network dynamics, implementing the transfer of data packets over a wireless network from one node to another. It essentially simulates the routing of data received from the TrueTime kernels over the wireless network to their respective destination.

In a typical cycle of operation of the NCS, the wave variables from a **PlantSystem** or multiple **PlantSystem**s are computed and sent to the **PowerJunction** from each TrueTime kernel. The received wave variables are sent to the **ControllerSystem** to compute the control signal which is then sent back to the **PlantSystem**s.

4 Passivity Analysis

4.1 Component Analysis

In order to achieve the desirable properties observed in passive systems, we have to analyze the components of the networked control system and make sure they satisfy passivity constraints.

The analysis of the *Plant* and *DigitalController* components of the networked control system for passivity is done automatically by an integrated Matlab analysis function. Each component is assumed to have a linear time-invariant (LTI) discrete-time model, so we use LMIs together with the `CVX` semidefinite programming tools for Matlab [14, 15]. On invocation (i.e. the modeler presses a button), a C++ model interpreter within GME [13] visits each component, and invokes the analysis function. Any components failing the passivity test are reported to the user.

The dynamics of the *Plant* and *DigitalController* models can each be defined by Eq.(1) and are characterized by the matrices A, B, C, D of size compatible with the number of inputs and outputs in the system and the number of states in the model. The passivity constraints for these models is defined by Linear Matrix Inequality (LMI) constraints [16]. For example, a LMI formula for strict output passivity for an LTI digital controller is given by

$$\begin{bmatrix} A^TPA - P - \hat{Q} & A^TPB - \hat{S} \\ (A^TPB - S)^T & -\hat{R} + B^TPB \end{bmatrix} \leqslant 0$$

$$\begin{aligned} \hat{Q} &= C^TQC, \quad \hat{S} = C^TS + C^TQD \\ \hat{R} &= D^TQD + (D^TS + S^TD) + R \end{aligned} \tag{2}$$

$$\exists \varepsilon > 0,\ Q = -\varepsilon I,\ R = 0,\ S = \frac{1}{2}I$$

The CVX semidefinite programming (SDP) tool is used in a Matlab script to solve the LMI for each component.

4.2 System-Level Analysis

Due to the "correct-by-construction" approach we use in designing networked control, we only analyze the *Plant* and *DigitalController* elements for passivity. If the *Plant* and *DigitalController* both satisfy the passivity constraints, the network control system as whole also satisfies the passivity principles.

The realization of the power junction element enforces some simple mathematical constraints which ensure passivity for interconnected components at runtime. These effects are also captured in the simulation of the power junction, so simulation should reveal any destabilizing effects. Further, the component interconnections are restricted in such a way that they are "correct-by-construction". Only valid (parallel) connections are allowed to the power junction, so any interconnected system of passive components in the language will be globally passive. The modeling language and its constraints encode the passive composition semantics, greatly reducing the analysis burden for determining passivity (and hence stability [6], [9], [17]) of the composed system design.

5 Code Generation

The main objective of the code generator is to generate MATLAB code that maps the models designed using the modeling language to Simulink models that represent the networked control system.

We developed a model interpreter that is used to synthesize simulation code from an instance model of the passivity based modeling language. The interpreter is developed in C++ using the Builder Object Network (BON2) API provided with GME [13]. The interpreter traverses all the entities of a particular networked control system instance model and extracts model parameters. These parameters and model structure are used to generate MATLAB files for configuring and building Simulink and TrueTime models to simulate the NCS.

The model interpreter creates translation rules between models and desired outputs. The entities in the instance model each map to a set of equivalently-defined components in Simulink and components from an advanced Simulink passivity-based control library. For example, the *Plant* and *DigitalController* entities discussed in Section 3 each map to an equivalent discrete state-space Simulink block. For these two entities the parameters for the equivalent Simulink blocks are instantiated using the parameter values entered by the user describing the dynamics of the entities. These parameters include the A, B, C and D matrices as well as the sampling time.

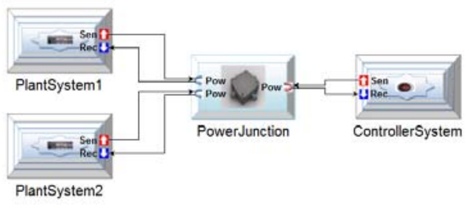
6 Case Study

We introduce a case study to demonstrate our design approach and also show that networked control systems designed using this approach are robust and remain stable when subject to uncertain network effects.

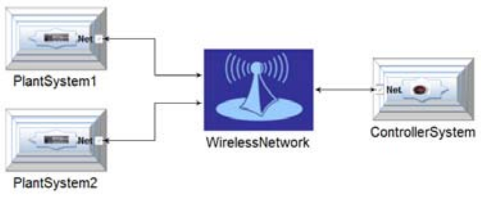
We created a networked control system which involves the control of two discrete plants using a single controller. The controller controls the two discrete plants to track a specified reference signal. The goal of the experiment was to model the network control system and generate a simulation of the behavior of the system. Although we used only two discrete plants for this case study, PaNeCS can model and simulate an arbitrary number of plants.

Fig. 7a and 7b respectively show the control design and platform aspects of the instance model respectively. Also, Fig. 7c shows the details of the plant system while Fig. 7d show the details of the controller system. The two plants modeled in the experiment are simple integrators (corresponding to physical models of inertial masses of 2kg and .25kg respectively) which are discretized. The plants' dynamics were modeled in state space form and the corresponding A, B, C and D matrices as well as the sampling time, T_s were provided as parameters to the instance model.

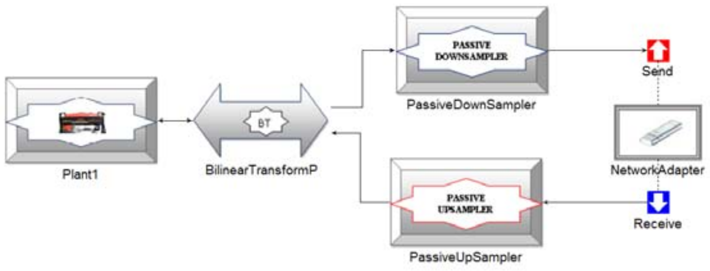
We used a proportional controller as the digital controller to command the plants to track a user-specified reference. The digital controller was also modeled in state space form and the A, B, C and D matrices and also the sampling time, T_s were provided as input parameters to the instance model. The parameters for



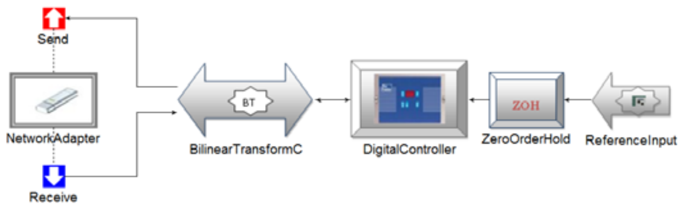
(a) Control Design Aspect



(b) PlatformAspect



(c) Plant Subsystem



(d) Controller Subsystem

Fig. 7. Sample Model of a Networked Control System

the dynamics of the plants and controller are provided in Table 1. The analysis tool checked and verified that the *Plant* and *DigitalController* models satisfied the passivity constraints. Then the code generator was used to generate code for creating a platform-specific Simulink simulation model from the parameters and design models in the modeling language.

PaNeCS provides the flexibility to easily model networked control systems using passivity and more quickly configure the model parameters of the system for many different adaptations. Using PaNeCs we tested the dynamics of the

NCS by running different experiments under different network conditions by adjusting parameters in the language and then generating code for simulating each configuration of the model. Table 2 shows the parameters for the simulations.

Experiment 1: Nominal Conditions In experiment 1, the system operated without the introduction of disturbance attacks. The three sample periods considered were 0.1s, 0.5s and 1s. The data rates were achieved by modifying the *Sample, M* parameters of the PassiveUpSampler and PassiveDownSampler entities. We only present plots for the results of the NCS having a sample period of 0.1s. Fig. 8 displays the velocity of the plants and the reference velocity provided to the controller. The plants closely tracked the reference velocity. The round trip delay for each plant seemed to have very little effect on the stability of the plants' velocity response. The delay can be attributed to the internal processing of the plants and controllers rather than network delay itself.

Table 1. Plant and Controller Dynamics.

	A	B	C	D	T_s
Plant1	1	1	.005	.0025	$.01s$
Plant2	.996	1	.04	.02	$.01s$
Controller	0	0	0	10π	$.1s$

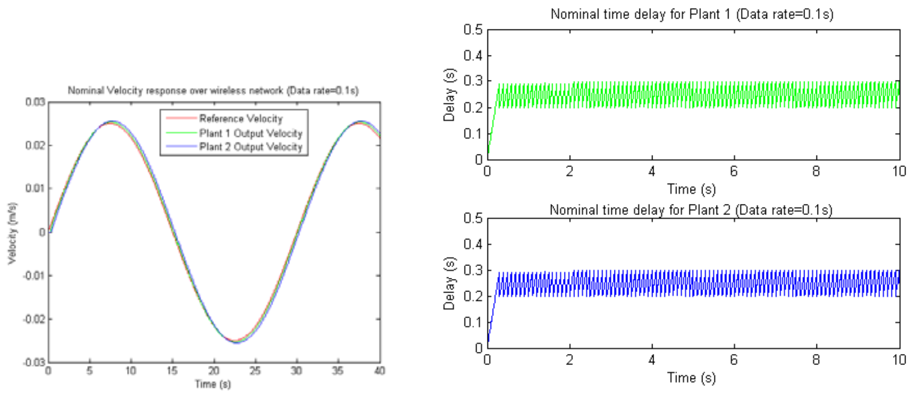


Fig. 8. Nominal velocity response and time delays (Data rate=0.1s)

Experiment 2: Network disturbances In experiment 2, a disturbance attack was introduced in the network. A disturbance node is configured using the *DisturbancePeriod* and DisturbancePacketSize from the **WirelessNetwork** model. Disturbance packets were sent over the network based on the value of a uniformly generated random number. Similar to Experiment 1, three different sample rates were tested, but we only present the results for the 0.1s sample period. Fig. 9 shows the velocity response of the plants and the time delay for each plant. The results show that even in the presence of disturbance attacks, the plants remain stable in tracking the reference velocity. This demonstrates the advantage of the passivity approach we use in designing networked control systems which guarantees the stability of the NCS in the presence of uncertainties due to network effects.

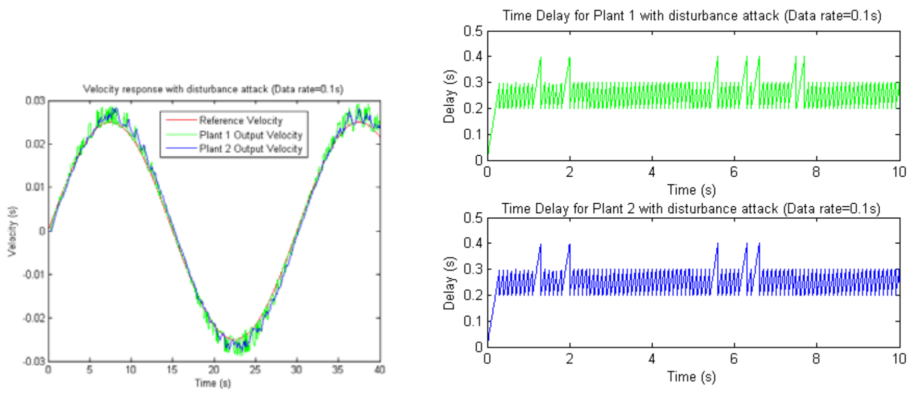


Fig. 9. Velocity response and time delays with disturbance attack (Data rate=0.1s)

Table 2. Simulation Parameters Summary.

	Sample Periods		
	$0.01s$	$0.05s$	$0.1s$
Plant1,M	10	50	100
Plant2,M	10	50	100
Disturbance $T_s = 0.01$ $Packetsize = 110,000bits$			

7 Conclusion and Future Work

Our model-based approach simplifies the process of designing passive networked control systems. We presented PaNeCS, a prototype modeling language for that purpose. We have presented an analysis tool that is used to test system components for passivity. We have also described model interpreters that generate code for simulation in MATLAB/Simulink using the TrueTime platform modeling toolbox. A case study involving the control of multiple discrete plants over a wireless network was used to demonstrate the details of models generated using the modeling language as well as the resulting simulation of the generated networked control system. The results showed that a networked control system could be designed using our approach which is robust and insensitive to uncertainties due to a few particular network effects. Our future work focuses on two major directions: (i) extending the language to include nonlinear and more complex systems,(ii) generating executables for deployment on actual systems.

References

1. Karsai, G., Sztipanovits, J., Ledeczi, A., Bapty, T.: Model-integrated development of embedded software. Proceedings of the IEEE **91**(1) (Jan. 2003)
2. Porter, J., Karsai, G., Volgyesi, P., Nine, H., Humke, P., Hemingway, G., Thibodeaux, R., Sztipanovits, J.: Towards model-based integration of tools and techniques for embedded control system design, verification, and implementation. In: Workshops and Symposia at MoDELS 2008, Springer LNCS 5421, Toulouse, France
3. AS-2 Embedded Computing Systems Committee: Architecture analysis and design language (aadl). Technical Report AS5506, Society of Automotive Engineers (November 2004)
4. Hudak J. and Feiler P.: Developing aadl models for control systems: A practitioner's guide. Technical Report CMU/SEI-2007-TR-014, CMU SEI (2007)
5. Balarin, F., Watanabe, Y., Hsieh, H., Lavagno, L., Paserone, C., Sangiovanni-Vincentelli, A.L.: Metropolis: an integrated electronic system design environment. IEEE Computer **36**(4) (April 2003)
6. Kottenstette, N., Hall, J., Koutsoukos, X., Antsaklis, P., Sztipanovits, J.: Digital control of multiple discrete passive plants over networks. Intl. Journal of Systems, Control and Communications, Special Issue on Progress in Networked Control Systems (2009)
7. The MathWorks, Inc.: Simulink/Stateflow Tools. http://www.mathworks.com
8. Ohlin, M., Henriksson, D., Cervin, A.: TrueTime 1.5 Reference Manual. Dept. of Automatic Control, Lund University, Sweden. (January 2007) http://www.control.lth.se/truetime/.
9. Kottenstette, N., Antsaklis, P.J.: Stable digital control networks for continuous passive plants subject to delays and data dropouts. In: Proceedings of the 46th IEEE Conference on Decision and Control. (2007) 4433 – 4440
10. Fettweis, A.: Wave digital filters: theory and practice. Proceedings of the IEEE **74**(2) (1986) 270 – 327
11. Secchi, C., Stramigioli, S., Fantuzzi, C.: Digital passive geometric telemanipulation. In: IEEE Intl. Conference on Robotics and Automation. (2003) 3290 – 3295
12. Berestesky, P., Chopra, N., Spong, M.W.: Discrete time passivity in bilateral teleoperation over the internet. In: IEEE International Conference on Robotics and Automation. (2004) 4557 – 4564
13. Ledeczi, A., Maroti, M., Bakay, A., Karsai, G., Garrett, J., IV, C.T., Nordstrom, G., Sprinkle, J., Volgyesi, P.: The generic modeling environment. Workshop on Intelligent Signal Processing (May 2001)
14. Grant, M., Boyd, S.: Cvx: Matlab software for disciplined convex programming. http://stanford.edu/ boyd/cvx (February 2009)
15. Grant, M., Boyd, S.: Graph implementations for nonsmooth convex programs. Recent Advances in Learning and Control (a tribute to M. Vidyasagar), Springer Lecture Notes in Control and Information Sciences (2008) 95–110
16. Kottenstette, N., Antsaklis, P.J.: Time domain and frequency domain conditions for passivity. Technical Report ISIS-2008-002, Institute for Software Integrated Systems, Vanderbilt University and University of Notre Dame (November 2008)
17. Kottenstette, N., Koutsoukos, X., Hall, J., Antsaklis, P.J., Sztipanovits, J.: Passivity-based design of wireless networked control systems for robustness to time-varying delays. RTSS (December 2008) 15–24

Formal Design Models for Distributed Embedded Control Systems

Christo Angelov, Krzysztof Sierszecki, Yu Guo

Mads Clausen Institute for Product Innovation
University of Southern Denmark
Alsion 2, 6400 Soenderborg, Denmark
{angelov, ksi, guo}@mci.sdu.dk

Abstract. The paper presents a formal specification of the software design models used in *COMDES-II* – a component-based framework for distributed control systems, featuring open architecture and predictable operation under hard real-time constraints. In this framework, an application is modelled as a network of distributed embedded actors that communicate transparently by exchanging labeled messages (signals), independent of their allocation on network nodes. Actors are configured from prefabricated executable components such as modal function blocks controlled by a master state machine, whereby actor structure is specified by a data flow model (function block network). Accordingly, actor behaviour is specified by composite functions representing signal transformations - from input to output signals, and system behaviour - by actor-level composite functions representing the overall sequence of computation – from system input to system output signals. Input and output signals are exchanged with the controlled plant at precisely specified time instants in accordance with the concept of Distributed Timed Multitasking, resulting in the elimination of transaction I/O jitter. System operation is ultimately described by a clocked synchronous model of computation featuring communicating actors, atomic (zero-time) execution of input and output actions and constant, non-zero execution time of system reactions.

Keywords: distributed control systems, component-based design of embedded software, domain-specific frameworks, correct-by-construction systems

1 Introduction

Nowadays, embedded software development is still dominated by conventional design methods and manual coding techniques. However, these are not able to cope with continuously growing demands for high quality of service, reduced development and operational costs, reduced time to market, as well as ever growing demands for software safety and dependability. In particular, software safety is severely affected by design errors that are typical for informal design methods, as well as implementation errors that are introduced during the process of manual coding.

This situation has stimulated the development of new software design methods based on formal design models (frameworks) specifying system structure and behaviour, which can be verified and validated before the generation of the program code [1, 2]. Furthermore, model-driven development can be combined with component-based design, whereby design models are implemented by means of reusable and reconfigurable components. Thus, embedded applications can be configured using repositories of prefabricated and validated components (rather than programmed), whereby the configuration specification is stored in data structures containing relevant information such as component parameters, input/output connections, execution sequences, etc. Hence, it is possible to reconfigure applications by updating data structures rather than reprogramming and reloading the entire application.

The main problem that has to be addressed with this method is to develop a *comprehensive*, yet *intuitive* and *open* framework for embedded systems. There are a considerable number of frameworks developed in the traditional Software Engineering domain that employ components with operational interfaces as well as various types of port-based objects, e.g. actor frameworks [4-8]. However, it can be argued that the architecture of the framework (i.e. models used to specify component functionality, interfacing and interaction) should be derived from areas such as Control Engineering and System Science, taking into account that modern embedded systems are predominantly control and monitoring systems. This approach has been used for some time with industrial control systems, whose software is built from component objects *(function blocks)* that implement standard application functions and interact by exchanging signals. Accordingly, function blocks are 'softwired' into function block networks that are mapped onto real-time control tasks, e.g. standards *IEC 61131-3* [10] and *IEC 61499* [11].

Unfortunately, this is a relatively low-level approach, which is inadequate for modern embedded applications. These vary from simple controllers to highly complex, time-critical and distributed systems featuring autonomous subsystems with concurrently running activities (tasks) that have to interact with one another within various types of distributed transactions. The above standards do not provide modeling techniques and component definitions at this level and do not define concurrency, whereby the mapping of function block networks on real-time tasks, as well as task scheduling and interaction are considered implementation details that are not a part of the standard.

In order to overcome the above problems, the Control Engineering models must be augmented with concepts and techniques developed in the Computer Science domain (concurrency, scheduling, communication, state machines, etc.), as advocated by leading experts in the area of Embedded Software Design, e.g. [2], [3]. The resulting framework must support compositionality and scalability through a well-defined hierarchy of reusable and reconfigurable components, including both actors and function blocks. On the other hand, it has to adequately specify system behaviour for a broad range of sequential, continuous and hybrid control applications.

These guidelines have been instrumental in developing the framework *COMDES-II* [13]. This is a domain-specific framework for time-critical distributed control applications, featuring a hierarchical component model as well as transparent signal-based communication at all levels of specification. In *COMDES-II*, an embedded application is composed from actors, which are configured from prefabricated function blocks. This is an intuitive and simple model that is easy to use and understand by application experts, i.e. control engineers.

An informal description of the above component models is given elsewhere [13]. This paper presents a formal specification of *COMDES-II* design models focusing on two interrelated aspects, i.e. system structure and behaviour. It is organized as follows: Section 2 presents a top-down specification of system structure in terms of data flow models describing actors and actor interactions, as well the internal structure of actors, which are composed of prefabricated function blocks. Section 3 presents a bottom-up specification of system behaviour starting with function block behaviour, followed by actor behaviour and finally - system behaviour. These are defined as composite functions specifying signal transformations - from input to output signals - of function blocks, actors and the system itself, respectively. Section 4 presents related research. The concluding section summarizes the main features of the framework and their implications for a software development process aimed at designing systems that are correct by construction.

2 Specification of System Structure

2.1 *COMDES-II* Design Models - an Introduction

In *COMDES-II*, an embedded system is conceived as a composition of active objects (actors) that communicate via labelled state messages (signals) encapsulating process variables, such as *speed*, *pressure*, *temperature*, etc. Communication is transparent, i.e. independent of the allocation of actors on network nodes. Accordingly, the system can be modelled by an actor network specifying constituent actors and the signals exchanged between them (see e.g. Fig. 1).

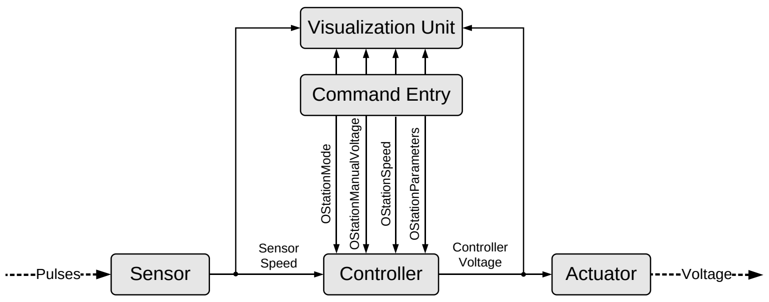


Fig. 1. COMDES-II actor network – an example: the DC Motor Control System

An actor is modelled as an integrated circuit consisting of a signal-processing block, which is mapped onto a non-blocking (basic) task, as well as input and output signal drivers that are used to exchange signals with other actors and the outside world (see Fig. 2). Actor tasks are configured from function blocks (FBs) and are modelled by function block networks. A function block is a *reusable executable component* that may have multiple instances within a given configuration. There are four kinds of function block: basic, composite, state machine and modal function blocks that can be used to implement a broad range of sequential, continuous and hybrid applications.

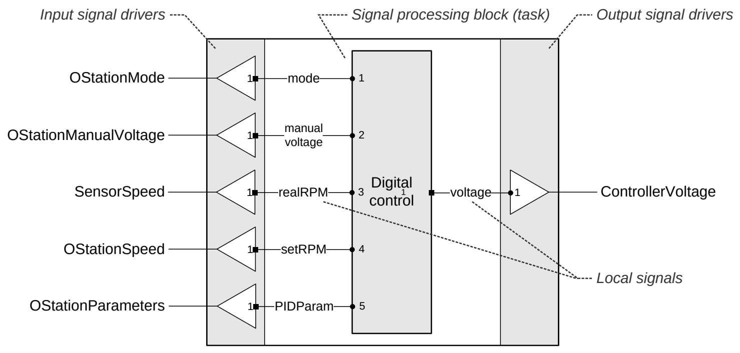


Fig. 2. COMDES-II *Controller* actor

Basic function blocks have simple stateless behaviour, which is specified by functions defining signal transformations - from input signals to output signals (e.g. a PID controller function block). Complex stateful behaviour is implemented with modal function blocks (MFBs). These may be viewed as a generalization of stateless function blocks: a MFB has a number of operational modes where each mode encapsulates one or more FB instances used to execute a control action associated with that mode. A modal function block receives indication of current mode from a supervisory state machine (SSM), whereby it executes the corresponding control action, in the context of a continuous or sequential control actor, e.g. *manual/automatic* control of DC motor rotation speed (see Fig. 3). A function block network may be encapsulated into a composite function block, which can be subsequently reused as an integral component.

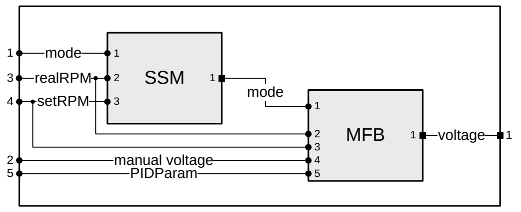


Fig. 3. The *Digital control* task composed of state machine and modal function blocks

Signal drivers are a special class of component - these are wrappers providing an interface to the system operational environment by executing kernel- or hardware-dependent functions. Specifically, signal drivers can invoke kernel primitives to transparently broadcast and receive signals, independent of the allocation of sender and receiver actors on network nodes [14].

A detailed informal description of the above component models is given elsewhere [13]. The following discussion presents a formal specification of *COMDES-II* components and component configurations. The latter takes into account the two levels of the framework, i.e. system and actor levels, which are treated in a top-down fashion. At the top level, the system is described as an *actor network* - a data flow model involving system actors and the global signals exchanged between them, as well as a definition of the signals in terms of identifiers and constituent signal variables. At the next level, each system actor is described by a *function block network*, i.e. a data flow model involving constituent function blocks and the local signals exchanged.

2.2 Distributed Control System Specification

A distributed embedded control system (ECS) is modelled as an actor network:

$$ECS = < \boldsymbol{A}, \boldsymbol{S}, \boldsymbol{C} > , \tag{1}$$

where $\boldsymbol{A}$ is the set of system actors, $\boldsymbol{S}$ is the set of system signals and $\boldsymbol{C}$ is the set of channels used to exchange signals between actors. The set of system actors $\boldsymbol{A}$ consists of environment actors $\boldsymbol{A}_{env}$ modelling the plant, and control actors $\boldsymbol{A}_{con}$ operating in a distributed system environment:

$$\boldsymbol{A} = \boldsymbol{A}_{env} \cup \boldsymbol{A}_{con} . \tag{2}$$

The set of system signals $\boldsymbol{S}$ can be represented as:

$$\boldsymbol{S} = \boldsymbol{S}_{in} \cup \boldsymbol{S}_{com} \cup \boldsymbol{S}_{out} , \tag{3}$$

where $\boldsymbol{S}_{in}$ is the subset of physical input signals, $\boldsymbol{S}_{com}$ is the subset of signals (messages) exchanged over the communication network, and $\boldsymbol{S}_{out}$ is the set of output physical signals. Furthermore, $\forall s_i \in \boldsymbol{S}$: $s_i = < Id_i, V_i >$, where Id_i is a signal identifier and V_i is a set of signal variables defined in terms of variable names and the corresponding data types:

$$V_i = \{ < s^i_1 : type^i_1 >, < s^i_2 : type^i_2 >, \; \ldots \; , < s^i_{ki} : type^i_{ki} > \} , \tag{4}$$

e.g. signal *OStationParameters* consisting of PID parameters, such as proportional, integral and derivative gain values (see Fig. 2).

The communication relationship between actors is specified in terms of channels that are defined by a *source* - *signal* - *destination* relation:

$$\boldsymbol{C} \subset \boldsymbol{A} \times \boldsymbol{S} \times 2^A , \tag{5}$$

e.g. one of the channels depicted in Fig. 1, which is specified by the tuple < *Sensor*, *Sensor_Speed*, {*Controller*, *Vizualization_Unit*} > .

In an actual implementation, control actors will be allocated to network nodes, and channels – to the network communication channel and physical I/O channels. The subsequent discussion assumes a real-time network with predictable message latency, such as CAN, which has been used for the experimental validation of *COMDES-II*.

A system control actor can be defined as:

$$a_{con} = < \boldsymbol{X} , L_{in}, N_{FB}, L_{out}, \boldsymbol{Y} > , \tag{6}$$

where: $\boldsymbol{X}$ is the set of input signals received by the actor, $\boldsymbol{X} \subset \boldsymbol{S}$, L_{in} is an input signal latch, N_{FB} is a signal-processing network of function blocks, L_{out} is an output signal latch and $\boldsymbol{Y}$ is a set of output signals generated by the actor, $\boldsymbol{Y} \subset \boldsymbol{S}$.

The input latch is used to receive input signals and decompose them into input signal variables constituting the set V, which may be viewed as local signals that are processed by the function block network. The latter computes output variables constituting the set W, which are used to compose the output signals generated by the output latch (see e.g. Figs. 2 and 3).

The I/O latches are composed of communication objects called *signal drivers*, denoted as D^{in} and D^{out}. In particular:

$$L_{in} = \{ D^{in}_i \}; D^{in}_i : s^{in}_i \rightarrow V_i , V_i \subset V , \\ L_{out} = \{ D^{out}_i \}; D^{out}_i : W_i \rightarrow s^{out}_i , W_i \subset W , \qquad (\mathbf{7})$$

where V_i and W_i denote the constituent variables of the corresponding I/O signals s^{in}_i and s^{out}_i , respectively.

The I/O latches are activated at the release and deadline instants of the actor task. This is a basic (non-blocking) task, whose internal structure is specified as a function block network performing the transformation of input signal variables into output signal variables: $V \rightarrow W$.

The FB network is modelled by an *acyclic* data flow graph (see e.g. Fig. 3), which can be defined as follows:

$$N_{FB} = < B, Z, Con > , \qquad (\mathbf{8})$$

where B is a set of function blocks (FBs), Z is a set of FB network variables and *Con* is the set of FB network connections.

A function block performs the signal transformation $X \rightarrow Y$, where X is the set of FB input variables, $X \subset Z$, and Y is the set of FB output variables, $Y \subset Z$. Specifically, a function block can be defined as:

$$FB = < X, Y, P, F > , \qquad (\mathbf{9})$$

where X, Y and P denote input, output and persistent variables, respectively and F is a set of functions.

Input variables X are generated by input drivers or other function blocks, $X \subset Z$. These are used together with persistent variables to compute output variables Y, $Y \subset Z$. Persistent variables P represent the internal *state* of the function block, which is retained from one execution to the next, e.g. various types of controllers, filters, etc. [10]. Simple function blocks may not have internal state, e.g. arithmetic function blocks, comparators. Output variables are computed by functions $f \in F$ that are defined as $y = f(x, p)$, where $y \in Y$, $x \in X$ and $p \in P$.

The variables constituting the set Z may be viewed as *local signals* associated with the function block network:

$$Z = V \cup I \cup W , \qquad (\mathbf{10})$$

where the input signal variables V are generated by input drivers and processed by function blocks; internal variables I are generated and processed by function blocks; output signal variables W are generated by function blocks and used by output drivers to compose output signals (see e.g. Fig. 3).

FB network connections are used to wire function blocks with input and output signal drivers, and with each other. The corresponding set can be specified as a union of subsets denoting input, internal and output connections: $Con = Con_{in} \cup Con_{int} \cup Con_{out}$. These are defined as *source* - *local signal* - *destination* relations as follows:

$$\begin{aligned} Con_{in} &\subset L_{in} \times V \times B \,, \\ Con_{int} &\subset B \times I \times B \,, \\ Con_{out} &\subset B \times W \times L_{out} \,, \end{aligned} \qquad \textbf{(11)}$$

e.g. the connection represented by the tuple < *SSM*, *mode*, *MFB* > shown in Fig. 3.

3 Specification of System Behaviour

3.1 *COMDES-II* Model of Computation – an Introduction

System operation is specified in terms of distributed transactions executed in accordance with a model of computation known as Distributed Timed Multitasking [12, 13], which is presently supported by the distributed real-time kernel *HARTEXμ* [14]. The distributed transaction involves a number of actors that execute transaction phases by invoking sequences of function blocks within the corresponding actor tasks. Actors interact with each other by exchanging labelled state messages (signals) using dedicated communication objects (signal drivers) that provide for transparent one-to-many communication between the actors involved.

Distributed Timed Multitasking (DTM) combines the concepts of Timed Multitasking [5] and transparent *signal-based* communication. With this model, it is assumed that signal drivers are short pieces of code that are executed atomically in logically *zero* time at precisely specified time instants, which is typical for control applications. Specifically, input signal drivers are executed when the actor task is released, and output drivers - when the task deadline arrives or when the task comes to an end, if it has no deadline (see Fig. 4). Consequently, task I/O jitter is effectively eliminated as long as the task comes to an end before its deadline.

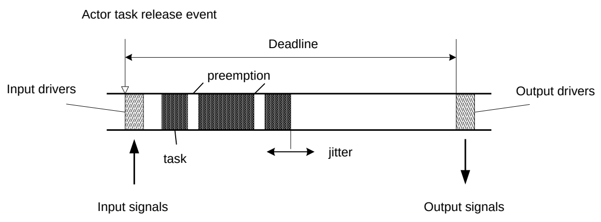


Fig. 4. Actor execution under Distributed Timed Multitasking

Jitter-free operation can be extended to distributed systems, e.g. a phased-aligned transaction involving the actors *Sensor* (*S*), *Controller* (*C*) and *Actuator* (*A*) from Fig. 1, which are triggered by a periodic timing event, such as a synchronization (*sync*) message denoting the initial instant of the transaction period (*T*), with deadline $D \leq T$

(see Fig. 5). In this case, input and output signals are generated at transaction start and deadline instants, resulting in the elimination of transaction I/O jitter.

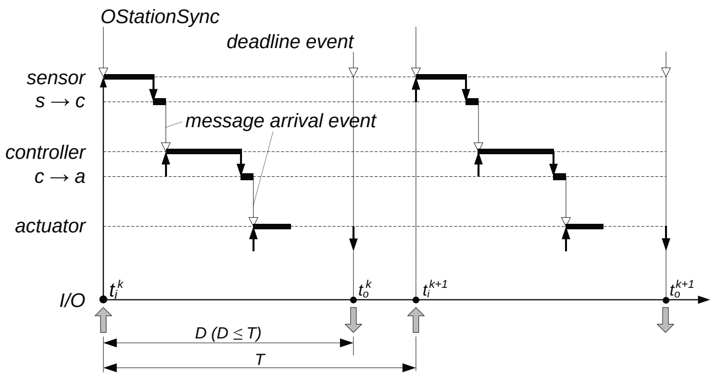


Fig. 5. Jitter-free execution of distributed transactions

The following discussion presents a formal specification of system operation, taking into account the adopted model of computation and the model of system structure developed in the preceding section.

3.2 Specification of Function Block Behaviour

Function block operation is specified with simple and/or composite functions from FB input variables *x(k)* to FB output variables *y(k)*, $x \in X$, $y \in Y$, assuming periodic execution of system actors and constituent function blocks, which are invoked at time instants *kT*, $k = 1, 2, \ldots$, where *T* is the execution period of the host actor.

Basic function blocks implement standard signal-processing functions, such as:

$y(k) = f(x(k))$ - with simple FBs implementing various kinds of mathematical operations, comparators, etc. (**12**)

$y(k) = f(x(k), p(k-1), p(k-2), \ldots p(k-l))$ - with FBs having persistent state, (**13**)

where the state is defined in terms of one or more persistent variables $p(k-1), p(k-2), \ldots, p(k-l)$, retained from previous periods $1, 2 \ldots, l$ and updated during each period (as specified by the concrete FB algorithm, e.g. the discrete-time versions of filters, various control algorithms, etc. [10]).

A composite function block (CFB) encapsulates a FB network whose behaviour is described with one or more functions such as $y(k) = f(x(k))$, where *f* is a composite function specifying the transformation of signals from CFB inputs to CFB outputs, which is defined in terms of the functions executed by the constituent function blocks. Assuming that the CFB encapsulates a sequence of *r* function blocks, this function can be represented as:

$f = f_r \circ f_{r-1} \circ \ldots \circ f_1$, or using another notation: $y(k) = f_r (f_{r-1} (\ldots (f_1(x(k)))\ldots))$ (**14**)

In the general case, this function will have a different expression for each particular configuration of the FB network, which has to be always modelled by an *acyclic* data flow diagram. However, cycles are allowed at actor level but these are effectively broken by one-period delays due to the adopted clocked synchronous model of computation (see below).

The supervisory state machine (SSM) implements the reactive aspect of actor behaviour, in separation from the transformational (signal processing) aspect, which is delegated to the modal function block. The SSM generates two output signals - *m* and *u*, meaning *mode* and *mode-updated*, which are specified by the corresponding functions:

$m(k) \;=\; f\,(m(k\text{-}1), e(k), pr(e(k)))$ - a mode transition function, and

$u(k)$ - a Boolean function, which is defined as follows: (**15**)

$u(k) \;=\; true$ when $m(k) \neq m(k\text{-}1)$, i.e. when a mode transition has taken place,
$u(k) = false$ when $m(k) = m(k\text{-}1)$, and no transition has taken place.

In the above expression *e(k)* denotes a *transition trigger,* i.e. an event specified as a Boolean expression involving binary input signals that are *present* at time *kT*, *T* is the period of the host actor, and *pr(k)* is the priority of the event triggering the transition from *m(k-1)* to *m(k).*

The modal function block (MFB) implements the signal processing aspect of actor behaviour by executing constituent function blocks within the corresponding modes of operation. These compute control signals y_i, $i\ =\ 1,\ 2,\ ...,\ r,$ by invoking signal transformation functions $f_1, f_2, \ldots, f_r$ – from input to output signals. Subsets of these functions are selected for execution, depending on the *mode* and *mode-updated* input signals indicated by the state machine function block, such that:

$\forall y_i \in A_p\, , y_i(k) = \; f_i(x(k))$, and $\forall y_i \in A_q,\ q \neq p,\ \ y_i(k) = \; y_i(k\text{-}1)$ - when $m(k) = p$ and $u(k) = true;$ (**16**)

$\forall y_i\, , y_i(k) = \; y_i(k\text{-}1)$ - when $u(k) = false$,

where A_p denotes the control action, i.e. the subset of control signals generated in mode *p*, and f_i is the function executed by the corresponding function block(s) in order to generate the signal y_i, $y_i \in A_p$. For instance, the control signal *voltage* of Fig.3 will be generated by a PID function block if *mode* has been updated to *automatic.*

The composition of supervisory state machine and modal function block operates as a periodically executed event-driven state machine whose operational semantics and implementation are presented in [15]. This state machine is invoked within a periodically executing host actor but a state transition takes place only when the corresponding transition trigger is present, much in the same way as event-driven state machines triggered by external interrupts.

3.3 Specification of Actor Behaviour

Actors generate reactions to execution triggering events in the form:

$$\boldsymbol{e} \;\rightarrow\; \boldsymbol{Y_e}\, , \tag{17}$$

where $\boldsymbol{Y_e} \subset \boldsymbol{Y}$, $\boldsymbol{Y_e}$ being the set of output signals generated by the actor in response to the execution trigger $\boldsymbol{e}$. The latter may be a local timing event $\uparrow(kT)$, a global timing event $\uparrow sync(kT)$ generated by a periodic synchronization message or an external event $\uparrow \boldsymbol{x}_{trigger}$, where $\boldsymbol{x}_{trigger}$ is one of the actor input signals (e.g. a message arrival event)[1].

Actor output signals $\boldsymbol{y} \in \boldsymbol{Y}$ are specified by functions of input signals $\boldsymbol{x} \in \boldsymbol{X}$ that are latched by input drivers at the time of input t_{in}. With periodic actors triggered by local or global timing events $t_{in} = kT, k = 0, 1, 2, \dots$

Output signals are composed of output signal variables generated by the actor FB network, which has a zero *logical execution time (LET)*. Hence, the output signal variables are logically related to the input time instant *kT*:

$$w(k) \;=\; \varphi(v(k))\,, \tag{18}$$

where φ is a composite function – from input signal variables $v \in V$ to output signal variables $w \in W$ that constitute actor input signals $\boldsymbol{x}$ and output signals $\boldsymbol{y}$, respectively.

With actors having purely transformational behaviour, φ can be defined like a CFB function, e.g.:

$$\varphi \;= f_r \circ f_{r-1} \circ \,.....\, \circ f_1\,, \tag{19}$$

where f_i are basic and/or composite signal-transformation functions executed by constituent function blocks, $i = 1, 2, \dots, r$.

With complex actors built from supervisory state machines coupled to modal function blocks, each mode generates certain control signals specified by the corresponding functions, for example:

$$\begin{aligned} w_1(k) \;&=\; \varphi^1\,(v(k)) \quad \text{- generated in mode } 1\\ w_2(k) \;&=\; \varphi^2\,(v(k)) \quad \text{- generated in mode } 2\\ &\dots\dots\dots\dots\\ w_s(k) &= \varphi^s\,(v(k)) \quad \text{- generated in mode } s \end{aligned} \tag{20}$$

In this case, for each φ^i, $\varphi^i = f_i \circ m$, where m is the mode transition function of the SSM function block and $f_i(v(k))$ is the signal transformation function executed by the modal function block when the supervisory state machine has indicated that $m(k) = i$.

In the general case:

$$\varphi^i \;=\; f_i \;\circ\; m \;\circ\; g\,, \tag{21}$$

where g denotes a pre-processing function. The latter is executed by a pre-processing (basic or composite) function block, generating a transition-trigger signal for the supervisory state machine (e.g. various types of arithmetic, comparators, counters, etc.)

The output variables generated by the actor task are used to compose output signals, which are latched into the output drivers at the time of output:

$$\boldsymbol{y}(t_{out}) \;=\varphi(\boldsymbol{x}(t_{in}))\,,\, t_{out} = \; t_{in} + D = kT + D,\, k = 0,\, 1,\, 2,\, \dots\,;\, 0 \le D \le T\,, \tag{22}$$

Hence:

$$\boldsymbol{y}(kT + D) \;= \varphi(\boldsymbol{x}(kT))\,, \tag{23}$$

[1] Bold symbols denote actor-level events and input/output signals.

and the actor as a whole has a *clocked synchronous semantics* [19], chracterized by a *non-zero* logical execution time (LET).

In the special case of actor without deadline, it is assumed that $D = 0$, and $t_{in} = t_{out} = kT$. Hence: $\mathbf{y}(k) = \varphi(\mathbf{x}(k))$, and the actor has a perfect *synchronous* semantics (zero LET). This is the case with intermediate actors of phase-aligned transactions, where the deadline is usually associated with the last actor, which has to generate the control signal at the transaction deadline instant (see next section).

3.4 Specification of System Behaviour

System operation is specified in terms of distributed transactions, such as the transaction shown in Fig. 5, assuming: 1) Periodic phase-aligned transactions involving non-blocking *basic* tasks, such as the one shown in Fig. 5, which are typical for distributed control applications [18]; 2) Non-blocking signal-based communication; 3) Distributed Timed Multitasking, which is an extension of Timed Multitasking for distributed transactions.

Under these assumptions, a periodic phase-aligned transaction with a period T_{trans} can be represented as a sequence of transaction phases, involving a number of actors, which are executed in response to a global timing event $\uparrow sync(kT_{trans})$ represented by the arrival of a synchronisation (*sync*) message generated by a sync master node:

$$\begin{aligned} &\uparrow sync(kT_{trans}) \rightarrow \mathbf{y}_1 ; \quad \mathbf{y}_1 = \varphi_1(\mathbf{x}_1), \\ &\uparrow \mathbf{x}_2 \rightarrow \mathbf{y}_2 ; \quad \mathbf{y}_2 = \varphi_2(\mathbf{x}_2), \\ &\ldots\ldots\ldots\ldots \\ &\uparrow \mathbf{x}_n \rightarrow \mathbf{y}_n ; \quad \mathbf{y}_n = \varphi_n(\mathbf{x}_n), \end{aligned} \qquad (24)$$

where: $\mathbf{x}_1 = \mathbf{x}_{in}$, $\mathbf{x}_2 = \mathbf{y}_1$, $\mathbf{x}_3 = \mathbf{y}_2, \ldots, \mathbf{x}_n = \mathbf{y}_{n-1}$, $\mathbf{y}_n = \mathbf{y}_{out}$.

Hence, transaction execution can be modelled with a composite function:

$$\Phi = \varphi_n \circ \varphi_{n-1} \circ \ldots\ldots \circ \varphi_1 , \qquad (25)$$

where φ_i is the function implemented by the *i*-th actor, $i = 1, 2, \ldots., n$.

Taking into account Distributed Timed Multitasking, transaction execution can be represented as a transformation from input signals $\mathbf{x}_{in}(t_{in})$ to output signals $\mathbf{y}_{out}(t_{out})$, where t_{in} and t_{out} are determined by the transaction period T_{trans} and deadline D_{trans}:

$$\begin{aligned} &\uparrow sync(kT_{trans}) \rightarrow \mathbf{y}_{out} , \\ &\mathbf{y}_{out}(kT_{trans} + D_{trans}) = \Phi(\mathbf{x}_{in}(kT_{trans})); D_{trans} \leq T_{trans} . \end{aligned} \qquad (26)$$

For the particular example illustrated by Figures 1 and 5, the behaviour of the control system can be represented in the form:

$$Voltage(kT_{trans} + D_{trans}) = \Phi(pulses(kT_{trans})), \Phi = \varphi_{actuator} \circ \varphi_{controller} \circ \varphi_{sensor} .$$

In the general case, the distributed system may consist of multiple subsystems executing distributed transactions with different rates of activation (multi-rate system), e.g. a multi-loop distributed control system. Accordingly, subsystem actors are allocated onto network nodes, and subsystem channels – onto the physical communication channel(s). This raises the issue of concurrent execution of transaction tasks/communications within the corresponding operational domains.

Following the adopted model of computation (Fig. 4), actor tasks are executed in a dynamic priority-driven scheduling environment provided by node-resident kernels, which are instances of the *HARTEXμ* timed multitasking kernel [14]. Communication takes place in a real-time network supporting predictable interactions, such as CAN. Transparent signal-based communication is supported by a dedicated protocol provided by the *HARTEXμ* kernel. With this protocol, signal drivers are executed *atomically* at precisely specified time instants that are fixed on the time axis. This makes it possible to eliminate the undesirable effects of task preemption and network communication, i.e. transaction I/O jitter, as long as transaction (end-to-end) response times are less than the corresponding end-to-end deadlines. This requirement can be checked using response time analysis developed for distributed real-time systems, e.g. the analysis method and tool presented in [18].

4 Related research

COMDES-II is a follow-on version of *COMDES-I* [12]. It employs an actor-based system model, whereby actors are conceived as units of concurrency as well as functionality (e.g., sensor, controller, actuator, etc.), whereas in the previous version a system is composed from function units encapsulating multiple threads of control. It also incorporates a different, i.e. composite state machine model emphasizing the separation of reactive and transformational (signal-processing) behaviour.

In *COMDES-II*, system operation is described by the Distributed Timed Multitasking (DTM) model of computation, which has been inspired by the original *Timed Multitasking* model [5] and is similar to the LET model adopted in the *xGiotto* language [6]. However, both of these models use port-based communication between actors, whereas DTM employs broadcast communication with labeled state messages (signals). This solution rules out artifacts such as ports, message queues, mailboxes, operational interfaces, etc., and provides for transparent interactions that are independent of the allocation of the actors on network nodes. Furthermore, the above frameworks use flat actor models with actors programmed in a conventional fashion, whereas *COMDES-II* actors are configured from prefabricated *executable* components – function blocks.

The adopted communication mechanism is characterized by complete separation of computation and communication, as recommended in [9], since signal drivers are executed in separation from actor tasks and from each other. That is not the case with port-based objects, where ports are usually defined as communication objects whose methods are invoked within task I/O drivers in a conventional call-return manner, see e.g. [5]. Consequently, the communication pattern is ‘hardwired’ in the code of I/O drivers and cannot be reconfigured without reprogramming.

The presented model of computation bears certain similarities with the models used in synchronous languages [20], and in particular: atomic execution of input and output actions; clocked operation similar to the execution pattern used in *LUSTRE* and *SIGNAL*; compositional data flow models inspired by the Control Engineering domain.

At the same time, there are substantial differences that have to be highlighted in order clearly differentiate the two models:

- Synthetic, component-based approach using prefabricated executable components vs. a conventional language-based approach used in synchronous languages
- True actor-level concurrency vs. conceptual concurrency, which is 'compiled away' during program compilation
- Constant *non-zero* reaction time vs. instantaneous (zero-time) reaction assumed by perfectly synchronous systems.

The last feature facilitates the engineering of distributed systems and eliminates problems related to fixpoints, instantaneous loops, etc., which have been major issues with synchronous systems. Furthermore, the synchronous model does not address the problem of task and transaction jitter because of the very nature of the synchrony hypothesis, whereas it is practically eliminated with the *COMDES* model of distributed computation.

5 Conclusion

The paper presents the formal specification of *COMDES-II* - a domain-specific framework for distributed embedded control systems, which combines open architecture and predictable behaviour under hard real-time constraints. The framework employs a hierarchical system model combining the concepts of both *actor* and *function block*: an embedded system is composed from autonomous system agents (actors), which are configured from prefabricated executable components – function blocks. Actors interact by exchanging signals, i.e. labeled messages with state message semantics, rather than using I/O ports or operational interfaces. This feature facilitates system reconfiguration and provides for transparent communication between actors, resulting in flexible and truly open distributed systems. Signal-based communication is also used for internal interactions involving constituent function blocks. That is why system configuration is specified by *data flow models* at all levels of specification. Consequently, actor behaviour is represented as a composition of component functions, and system behaviour – as a composition of actor functions. A synchronous model of computation is applied at the component level. A clocked synchronous model of execution is applied at the actor and system levels, i.e. Distributed Timed Multitasking.

The presented software architecture has important implications for software safety and predictability, as well as the entire software development process. In this case, applications are configured from prefabricated and validated (*trusted*) components, following strict composition rules that are derived from the syntax and static semantics of the framework. The behaviour of software components and applications is rigorously specified via a hierarchy of formal models that constitute the behavioural semantics of the framework. On the other hand, the use of timed multitasking makes it possible to engineer highly predictable systems operating in a flexible, dynamic scheduling environment.

This has been demonstrated in a number of experiments used to validate the framework, e.g. distributed computer control systems involving physical and computer models of plants, such as electric DC motor, production cell, steam-boiler, turntable machine, etc. It has also been applied in an industrial case study - a medical ventilator control system [17]. In all cases, the use of the framework helped reduce development time and increase software quality. This was quite obvious with some of the systems mentioned above, e.g. the production cell control system, which was developed in a relatively short time and became operational without extensive testing and debugging.

However, in order to guarantee that an application is correct by construction, it has to be proven correct with respect to the required functional and timing behaviour. That is only possible if a precise and unambiguous system model is developed, whose particular features would desirably facilitate the process of analysis. In *COMDES-II* that is accomplished through formal design models emphasizing the principle of *separation of concerns*, i.e. separate treatment of computation and communication, functional and timing behaviour, reactive and transformational behaviour, etc. Thus, different aspects of system behaviour can be verified in separation using appropriate techniques and tools. Functional behaviour can be analyzed using tools such as *Simulink* (with continuous systems) and *Uppaal* (with discontinuous systems), following semantics-preserving transformation of system design models into the corresponding analysis models, whereas timing behaviour can be verified through numerical response-time analysis.

In particular, *Simulink* can be used to analyse system behaviour via simulation. That is facilitated by the similarity between *COMDES-II* design models and *Simulink* analysis models representing the controller part of the system, both of which are discrete-time data flow models. Consequently, it is possible to export a *COMDES-II* design model to the *Simulink* environment, by wrapping *COMDES-II* components into S-functions and wiring them together, following the interconnection pattern of the original design model. This analysis method has been successfully experimented with the medical ventilator case study, whereby the *COMDES-II* design of the control system has been exported to *Simulink* and subsequently validated via numerical simulation.

The envisioned development process will make it possible to engineer embedded applications that are *correct by construction*. This will hopefully eliminate design errors, which are difficult and costly to repair. On the other hand, implementation errors will be eliminated through an automated configuration process supported by an integrated toolchain [16], which is based on meta-models that have been derived from the formal design models presented in this paper. Ultimately, the elimination of both design and implementation errors will considerably enhance software safety, which is of paramount importance for the overall safety of embedded applications.

6 References

1. B. Bouyssounouse and J. Sifakis (Eds.), "Embedded Systems Design. The ARTIST Roadmap for Research and Development", *LNCS 3436* (2005)
2. T.A. Henzinger and J. Sifakis, "The Embedded Systems Design Challenge", Proc. of the 14th International Symposium on Formal Methods FM 2006, *LNCS 4085* (2006), pp. 1-15
3. P. Caspi, "Some Issues In Model-Based Development for Embedded Control Systems", Invited Lecture, DIPES'2006, Braga, Portugal, Oct. 2006

4. D.B. Stewart, R.A. Volpe and P.K. Khosla, “Design of Dynamically Reconfigurable Real-Time Software Using Port-Based Objects”, *IEEE Transactions on Software Engineering*, vol. 23, No 12, 1997, pp. 759-776
5. J. Liu and E.A. Lee, “Timed Multitasking for Real-Time Embedded Software”, *IEEE Control Systems Magazine: Advances in Software Enabled Control*, Feb. 2003, pp. 65-75
6. A. Ghosal, T.A. Henzinger, C.M. Kirsch and M.A. Sanvido, “Event-Driven Programming with Logical Execution Times”, Proc. of HSCC 2004, *LNCS 2993* (2004), pp. 357-371
7. D. Isovic and C. Norström, “Components in Real-Time Systems”, Proc. of the 8th International Conference on Real-Time Computing Systems and Applications RTCSA’2002, Tokyo, Japan, March 2002
8. H. Hansson, M. Åkerholm, I. Crnkovic and M. Törngren, “SaveCCM – A Component Model for Safety-Critical Real-Time Systems”, Proc. of the 30th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2004, pp. 627-635
9. A.L. Sangiovanni-Vincentelli and G. Martin, “Platform-Based Design and Software Design Methodology for Embedded Systems”, *IEEE Design and Test of Computers*, vol. 18 (2001), pp. 23-33
10. K.H. John and M. Tiegelkamp, IEC 61131-3: *Programming Industrial Automation Systems*, Springer, 2001
11. R. Lewis, *Modeling Control Systems Using IEC 61499*, Institution of Electrical Engineers (2001)
12. C. Angelov, K. Sierszecki, N. Marian and J. Ma, “A Formal Component Framework for Distributed Embedded Systems”, in I. Gorton et al. (Eds.): Proc. of CBSE 2006, *LNCS 4063* (2006), pp. 206-221
13. C. Angelov, X. Ke and K. Sierszecki, “A Component-Based Framework for Distributed Control Systems”, Proc. of the 32nd EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2006, Cavtat, Dubrovnik, Croatia, Aug.-Sept. 2006, pp. 20-27
14. K. Sierszecki, C. Angelov and X. Ke, “A Run-Time Environment Supporting Real-Time Execution of Embedded Control Applications”, Proc. of the 14th International IEEE Conference on Embedded and Real-Time Computing Systems and Applications RTCSA 2008, Kaohsiung, Taiwan, Aug. 2008
15. C. Angelov, X. Ke, Y. Guo and K. Sierszecki, “Reconfigurable State Machine Components for Embedded Applications”, Proc. of the 34th EUROMICRO Conference on Software Engineering and Advanced Applications SEAA 2008, Parma, Italy, Sept. 2008, pp. 51-58
16. Y. Guo, K. Sierszecki and C. Angelov, “COMDES Development Toolset”, Proc. of the 5th International Workshop on Formal Aspects of Component Software FACS 2008, Malaga, Spain, Sept. 2008, pp. 233-238
17. F. Zhou, W. Guan, K. Sierszecki and C. Angelov, “Component-Based Design of Software for Embedded Control Systems: the Medical Ventilator Case Study”, Proc. of the International Conference on Embedded Software and Systems ICESS 2009, Hanchzhou, China, June 2009
18. W. Henderson, D. Kendall and A. Robson, “Improving the Accuracy of Scheduling Analysis Applied to Distributed Systems”, *Real-Time Systems*, vol. 20, No 1 (2001), pp. 5-25
19. A. Jantsch, *Modeling Embedded Systems and SoCs - Concurrency and Time in Models of Computation*, Morgan Kaufmann, 2003
20. A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic and R. de Simone, “The Synchronous Languages 12 Years Later”, *Proc. of the IEEE*, vol. 91, No 1, Jan. 2003, pp. 64-83

Improving Timing Analysis for Matlab Simulink/Stateflow

Lili Tan, Björn Wachter, Philipp Lucas, Reinhard Wilhelm*

Universität des Saarlandes, Saarbrücken, Germany
{lili,bwachter,phlucas,wilhelm}@cs.uni-sb.de

1 Introduction

Control software in embedded hard real-time systems is subject to stringent timing constraints. To compute the required safe upper bounds on its worst-case execution-time (WCET), static timing analysis is used in industry [1].

Today control software is predominantly developed with model-based design tools such as Matlab Simulink/Stateflow. However, current timing tools lose precision as they consider infeasible executions, e.g., changes between operating modes not admissible in the model. These tools analyze compiled executables where information about the feasibility of executions is hard to derive. We propose systematic methods that make model information available to timing analysis and present promising results with Simulink/Stateflow models.

Static Timing Analysis. Static timing analysis [2] uses abstract interpretation [3] to derive program properties that hold for all executions. A classical static analysis is interval analysis, which determines, for each variable, a range of values for each program point which contains *all* the values of the variable in any program execution. The ranges are guaranteed to be safe, i.e., they can be used to exclude division by zero and array-out-of-bounds accesses at compile time. More generally, static analysis computes provably safe approximations of program states.

Static timing analysis determines execution time bounds for programs. These bounds must be safe, i.e., they must not underestimate the execution time. They should also be tight to avoid unnecessary safety margins.

The established methodology splits the problem into different phases. The input to the analysis is a compiled executable of the program. The first phase reconstructs from the executable a control-flow graph (CFG) over basic blocks. In the next phase, a variation of interval analysis, called value analysis, determines the contents of registers and memory locations. Then a micro-architectural analysis computes execution-time bounds for basic blocks. It accounts for the tremendous hardware-induced execution-time variability: depending on whether a memory access causes a cache hit or a cache miss, the execution time of an instruction may differ by two orders of magnitude. Therefore complex, processor-specific architectural features like cache and pipeline effects are considered [4]. In the final phase, path analysis determines a safe estimate of the WCET. First an

* Supported by ITEA 2 project 06042, EU-FP7 Grant 216008 and SFB/TR 14.

ILP generator models the control flow the program as an integer linear program. Each ILP variable corresponds to the traversal count of a basic block. The value of the objective function in the solution is the predicted execution time bound.

The CFG also describes infeasible program executions if *conditions* are not interpreted. Consider the C code `if(a>0) x=1; else x=2; if(a==0) a=x;`. Conditions `a>0` and `a==0` are clearly *correlated*, more specifically, they mutually exclude each other. Although the control-flow graph contains the path from `x=1` to `a=x`, this is not a feasible program execution. In general, we call a control-flow path an *infeasible path* if it does not correspond to any program execution (this notion is distinct from dead code).

To make path analysis more precise, so-called *flow constraints* can be added to the ILP that eliminate infeasible paths. A salient point of our work is that such constraints can be systematically derived from model information.

Matlab Models and Generated Code. Matlab Simulink/Stateflow is a hierarchical modeling language for control software with a sequential, imperative semantics. The underlying methodology is to design control computation within Simulink and control logic within Stateflow. Simulink offers building blocks for proportional, integral and differential (PID) control computations and estimations, e.g., filters, look-up tables, and arithmetic operators. Stateflow is an automata specification language that can be used to express transitions between different operating modes of the system. Blocks communicate with each other via signals and receive external inputs from the environment.

For deployment, code generators synthesize production C code, in which the internal states of Stateflow and Simulink blocks are encoded by state variables. Signals and internal inputs also map to C variables. The implementation of blocks can be traced in the source code. However this mapping depends on characteristics of different code generators.

2 Model-aware Timing Analysis

In real-time systems, the different tasks run periodically and are triggered by a scheduler. These tasks are commonly implemented with model-based tools like Matlab. A periodic run corresponds to one execution of the Matlab model where inputs are received, the internal state is updated, and outputs are produced. Timing analysis has to determine an execution time bound that is safe for *each* run. It is impossible in practice to know the worst-case inputs or the worst-case internal state, hence the analysis has to cover all possibilities for each run.

To ensure safety, the analysis must not assume that the value of an external input variable remains constant between definition and use, i.e., the variable is 'volatile' in C terminology. For the internal state, timing analysis has to assume all possibilities at task entry, i.e., for a state variable, assume all potential states. Thus, both input and state must be treated specially to obtain a safe execution time bound. In Matlab-generated code, input and state variables can be identified syntactically. This enables an automatic solution that guarantees safe bounds. In the remainder of the section, our goal is to make these bounds tighter.

We investigate where precision is lost due to infeasible paths. To this end, we focus on typical patterns at the level of the model that lead to infeasible paths. As a running example, we consider the fuel-rate controller which is a Matlab demo model that contains typical features of embedded controllers. The controller estimates airflow rate, and calculates the fuel injection rate based on PID control principle.

We analyzed the controller with `aiT` WCET Analyzer, the static timing analysis tool [2] of AbsInt [5]. `aiT` produces a worst-case path to explain the execution time bound it has computed. Without providing flow constraints, the execution time is over-approximated and the computed worst-case path is infeasible, since static timing analysis is not aware of certain dependencies in the model.

For example, like any control software, the fuel-rate controller has operating modes and signals that conditionally exclude each other. Depending on the current mode, signals, and their logical combinations, different look-up tables or computations are triggered. As discussed in the introduction, the timing analyzer generally does not interpret conditions. Hence it has to take the longer branch of a conditional, even if execution history of the path does not admit so. As a result, the worst-case path spuriously 'switches' between operating modes.

For illustration, we consider such spurious resolutions of conditions on the worst-case path. Some resemble the infeasible-path example in the introduction, e.g., they involve conditions like `mode==LOW` and `mode==RICH`. Other conditions are more involved. For example, condition `O2_fail==0 && mode==LOW` checks if the oxygen sensor is valid and the system is in operating mode `LOW`, while condition `pressure_fail==1` checks if the pressure sensor has failed. These conditions do not have shared variables, and, simply by looking at the expressions, they seem not to be related. Yet there is a relation entailed by the model: the conditions are, in fact, mutually exclusive. The conditions are used in a Simulink block, while the variables `mode`, `O2_fail`, `pressure_fail` are set by a Stateflow automaton. However, the Stateflow automaton would not set `mode` to `LOW` if any sensor had sent a failure signal. Such *entailed relations* need to be derived by analyzing the model semantics. In the source code or executable, dependencies are more implicit and even harder to track than in the model. In the following, we show how to construct flow constraints from the model to achieve a more precise timing analysis.

Trigger Conditions. We aim at conditions that determine whether a piece of the model is executed. These conditions on external inputs, internal signals (e.g., mode variables), and states guard signal transformation and control computation. Simulink/Stateflow express this by conditional blocks, similar to conditionals in C, e.g., triggered and enabled subsystems, guarded transitions in Stateflow and switch-blocks. We uniformly refer to the conditions as *trigger conditions.*

Flow Constraints from Definition-Use Dependencies. We formulate flow constraints that relate a definition, e.g., a mode variable, and uses of that variable. Certain definitions always make a trigger condition false. Trivially, a program execution cannot pass through such a condition *and* the branch guarded by the trigger condition. This can be expressed by flow constraints. One example for such constraints in the fuel-rate controller are signals that indicate a failure of a sensor. These signals are set in a Stateflow block and are used in a Simulink block to trigger the evaluation of a lookup table.

Flow Constraints from Correlations between Trigger Conditions. Relations between trigger conditions can be formulated as flow constraints, e.g., independent, equivalence, implication, antivalence, and exhaustion can be expressed. To be effective, entailed relations need to be considered. The analysis of entailed relations requires information about deep semantic properties of Stateflow and Simulink blocks. To this end, we anticipate that relational abstract domains from static analysis may be helpful.

Other relations could be derived purely from Simulink. This includes the common case of a choice between two implementations of an algorithm with directly inverse trigger conditions.

Significant Branches. Eliminating infeasible paths does not per se improve precision. For example, if branches of conditionals have approximately the same execution time, there can be little gain in precision. Therefore, we focus on significant unbalanced branches when giving flow constraints. In our running examples, the invocations of look-up tables and mode-dependent discrete filters give rise to such branches.

Relative to Stateflow, the Simulink blocks typically dominate the execution time, while Stateflow blocks themselves contribute little to the overall execution time. This is because control logic computations consist of conditionals and assignments, while the expensive computations are often in the Simulink part, e.g., lookup tables and discrete filters for estimation and PID control. Thus determination of infeasible paths pays off more in the Simulink part than in Stateflow.

Experimental Results. We used `aiT` for our experiments. For the fuel-rate controller, we have manually applied the described derivation method for flow constraints. Flow constraints from definition-use dependencies alone reduced the execution time bound by 4%. Adding both kinds of flow constraints yields an overall reduction by 19% and a feasible worst-case path. If we compute an execution-time bound for each operating mode, we achieve a reduction from 20% to 48% per operating mode.

3 Related Work

Previous work on flow constraints focused on the executable [6], or C level. In [7], the authors consider timing analysis of code synthesized from Esterel. They identify flow constraints to eliminate feasible paths. The principal ideas concerning the two kinds of flow constraints are related, however Esterel is significantly different from Matlab Simulink, e.g., Esterel does not have automata as a language feature. Hence rules to derive flow constraints differ significantly.

[8] describes early work on timing analysis for Simulink models *without* Stateflow. Model information like loop bounds is passed to the underlying timing analysis tool. They modified the code generator and used their own (uncertified) compiler. Their timing analysis tool lacks value analysis [9] and thus does not discover loop bounds which `aiT` derives from the executable alone. Integrations of `aiT` with ASCET and SCADE are described in [10] and [11]. They pass model information to `aiT`, e.g., variable ranges and loop bounds. Unlike this paper, [8, 10, 11] mainly focus on other aspects than precision.

4 Conclusion

Initial results the benefit of model information in terms of automation and precision of WCET analysis. We propose model-based generation of flow constraints and have evaluated our method using the industrial tool `aiT`. Initial results with the fuel-rate controller are promising. While definition-use flow constraints are relatively easy to apply, relations between trigger conditions are more difficult to automate due to entailed relations. In future work, we will automate the generation of flow constraints and apply our approach to industrial examples.

References

1. Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R., Ferdinand, C.: An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software Systems. In: Proceedings of DSN. (2003)
2. Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S., Wilhelm, R.: Reliable and precise WCET determination for a real-life processor. In: EMSOFT. Volume 2211 of LNCS. (2001) 469 –485
3. Cousot, P., Cousot, R.: Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. In: POPL77, Los Angeles, California (1977) 238–252
4. Heckmann, R., Langenbach, M., Thesing, S., Wilhelm, R.: The influence of processor architecture on the design and the results of WCET tools. Proceedings of the IEEE **91** (2003) 1038–1054
5. AbsInt Angewandte Informatik GmbH: http://www.absint.com/
6. Stein, I., Martin, F.: Analysis of path exclusion at the machine code level. In: Proceedings of WCET. (2007)
7. Ju, L., Huynh, B.K., Roychoudhury, A., Chakraborty, S.: Performance debugging of Esterel specifications. In: CODES+ISSS. (2008) 173–178
8. Kirner, R., Lang, R., Freiberger, G., Puschner, P.: Fully automatic worst-case execution time analysis for Matlab/Simulink models. In: ECRTS. (2002) 31–40
9. Tan, L.: The worst-case execution time tool challenge 2006. International Journal on Software Tools for Technology Transfer (STTT) **11** (2009) 133 – 152
10. Ferdinand, C., Heckmann, R., Wolff, H.J., Renz, C., Parshin, O., Wilhelm, R.: Towards model-driven development of hard real-time systems. In: Proceedings of ASWSD. (2006) 145–160
11. Ferdinand, C., Heckmann, R., Sergent, T.L., Lopes, D., Martin, B., Fornari, X., Martin, F.: Combining a high-level design tool for safety-critical systems with a tool for WCET analysis on executables. In: Proceedings of ERTS. (2008)

Prototyping of Distributed Embedded Systems Using AADL*

Mohamed Yassin Chkouri and Marius Bozga
{Yassin.Chkouri, Marius.Bozga}@imag.fr

Verimag, Centre Equation - 2, avenue de Vignate 38610 GIERES

Abstract. Prototyping distributed applications can be extremely useful in evaluating a design, and also in understanding the effect of different parameters on the performance of an application. Architecture Analysis and Design Language provide adequate syntax and semantics to express and support distributed embedded systems. This paper studies a general methodology and an associated tool for building and translating AADL systems into a distributed application using network communication protocol. This allows runtime analysis to fully asses system viability, to refine and to correct the behavior of the system using BIP. Using our prototype we analyse the case study MPC in a native platform (PC).

1 Introduction

Distributed applications are used in many safety-critical domains such as space and avionics. Designing distributed systems demands more attention and rigour methodology. The produced systems have to conform to many stringent functional and non-functional requirements from multiple contexts.

Ensuring all the requirements and features becomes very hard if the whole system is hand-coded. Thus, the application code should preferably be generated automatically from a verifiable and analyzable model. This makes easier the work of the developer and helps during the stage of code verification. Besides, constructing a verifiable model from the application model using model transformation is simpler and safer than constructing this model from source code.

Architecture Description Languages (ADLs) have been proposed to support the development process of embedded real-time and distributed applications. This paper presents a definition framework for ADLs. The utility of the definition is demonstrated by using it to differentiate and compare several existing ADLs. This will allow us to choose an ADL according to our requirements.

Among the ADLs, AADL [3] is the Architecture Analysis and Design Language that allows the modeling of distributed, real-time applications. AADL was first introduced to model the hardware and software architectures in the avionics domain. An AADL system model consists of components, their interfaces, the connections between them and properties on various entities of the system

* This work is partially supported by ITEA/Spices and OpenEMBeDD projects

model. The AADL standard defines a textual as well as graphical form of the language.

AADL has been designed to build distributed real-time and embedded systems. AADL can be seen as a collection of many requirements covering many domains. System designers and developers need to describe both functional and non-functional requirements. These requirements must then be sorted and enforced at the deployment level. We will presents the set of requirements that must be respected to build distributed systems.

We have shown in [13], how AADL systems can be automatically translated into BIP [8] (Behavior Interaction Priority), and analyzed using the BIP toolset. BIP is a language for the description and composition of components as well as associated tools for analyzing models and generating code on a dedicated middleware. The language provides a powerful mechanism for structuring interactions involving rendezvous and broadcast.

In this paper, we present an extension of our translation to prototype distributed applications using BIP and network communication protocol. We begin with a model built by the application designer, who maps its application entities onto a hardware architecture. Then, we use AADL into BIP tool to generate BIP model conforming to AADL semantics. Finally, we use a code generator to generate an executable model for each systems with communication protocol. This translation allows simulation of distributed systems specified in AADL in addition to the application of formal verification techniques developed for BIP, e.g. deadlock detection, verification of properties, etc.

The translation from distributed AADL systems into BIP is illustrated on a case study: the Multi-Platform Cooperation (MPC) example provided by J. Hugues [18]. Using our tool, we were able to run the case study in a native platform (PC). In order, to debug and evaluate the case study before deploying it on a distributed embedded platform.

Distributed embedded application code generation from models is not limited to AADL. In fact, distributed and high-integrity systems are probably the domain which has the most maturity. OCARINA [17] allows model manipulation, generation of formal models to perform scheduling analysis and generate distributed applications. OCARINA allows code generation from AADL descriptions to Ada. PolyORB [27] is a middleware toolset that provides distribution services through standard programming interfaces and communication protocols. However, the generated code from AADL does not take into account the annex behavior specifications [1].

This paper is organized as follows. Section 2 gives definition and comparaison between existing ADLs. Section 3 gives an overview of AADL. In section 4, we explain how to translate AADL systems into distributed application using network communication protocol. In section 5, we present a MPC case study and it deploylment into a distributed application. Conclusions close the article in Section 6.

2 Architecture Description Languages

Architecture Description Languages (ADLs) have been proposed as modeling notations to support architecture-based development. An ADL is a language that provides features for modeling a software system's conceptual architecture, distinguished from the system's implementation. ADLs provide both a concrete syntax and a conceptual framework for characterizing architectures.

The building blocks of an architectural description are (1) components, (2) connectors, and (3) architectural configurations. Here we give a short description of these blocks:

- A component in an architecture is a unit of computation or a data store.
- Connectors are architectural links used to model interactions among components and rules that govern those interactions.
- Architectural configurations, or topologies, are connected graphs of components and connectors that describe architectural structure. This information is needed to determine whether appropriate components are connected, their interfaces match, connectors enable proper communication, and their combined semantics result in desired behavior.

A number of ADLs have been proposed for modeling architectures both within a particular domain and as general-purpose architecture modeling languages. We specifically consider those languages most commonly referred to as ADLs: C2 [21, 20], *Rapide* [15], *Darwin* [19], *UniCon* [24], *SADL* [22, 26], *AADL* [3].

Several researchers have attempted to shed light on these issues, either by surveying what they consider existing ADLs [28, 14] or by classifing and comparing several existing ADLs in some specific areas [25].

Comparisons between the languages (Figures 1, and 2) are given with respect to: components, connections, priorities between components, behavior description and support for distributed embedded system.

All the above languages make distinction between a component interface and an instance of a component that exhibits that interface. All the languages provide syntax and semantics for component interface specification. All the languages view a component interface specification as defining a component type, where there can be multiple instances of components that exhibit that same interface. All languages allow a hierarchical composition that allows architectures to describe software systems at different levels, by using a collection of subcomponents and connections between those subcomponents.

C2, Darwin, SADL, and UniCon share much of their vocabulary and refer to them simply as components; in Rapide they are interfaces; and in AADL component categories.

In this paper, we are interested by ADL which support distributed embedded systems, priority for schedulability analysis, behavior using state machine, and functional and non-functional properties. AADL was first introduced to model the hardware and software architectures in the avionics and automotives domain, and it is backed by several industries.

	Component		
	Interface	**Implementation**	**Non-functional properties**
C2	exported through top and buttom ports; provided and required	component implementation	none
SADL	input and output ports (iports and oports)	component implementation	requires component modification
Rapide	provides, requires, action and service	interface; implementation	none
Darwin	services (provided and required)	component implementation	none
Unicon	players	component implementation	attributes for schedulability analysis
AADL	inputs and outputs ports (event and/or data); provide and require; in and out parameters	component implementation	time constraints schedulability properties safety level

Fig. 1. Comparison between ADLs

Noticeable about the AADL is its strong syntactic and semantic support for architectures consisting of components of a limited number of functional categories. Along with this it allows to add non-functional properties to architectural components, such as timing, memory consumption and safety properties. In this way, the model of a system architecture allows specific tools to predict non-functional properties of the system in early design phases, which makes AADL a particularly interesting notation for distributed embedded software development.

Compared to other modeling languages, AADL defines low-level abstractions including hardware descriptions. These abstractions are more likely to help design a detailed model close to the final product.

3 Architecture Analysis & Design Language

The SAE Architecture Analysis & Design Language (AADL) [3] is a textual and graphical language used to design and analyze the software and hardware architecture of performance-critical real-time systems. It plays a central role in several projects such as Topcased [6], OSATE [4], ASSERT [2], SPICES [5].

A system modelled in AADL v.1 consists of application software mapped to an execution platform. Data, subprograms, threads, and processes collectively represent application software. They are called *software components*. Processor, memory, bus, and device collectively represent the execution platform. They are called *execution platform components*. Execution platform components support the execution of threads, the storage of data and code, and the communication between threads. Systems are called *compositional components*. They permit software and execution platform components to be organized into hierarchical structures with well-defined interfaces. Operating systems may be represented either as properties of the execution platform or can be modelled as software components. Behavior specifications [1] can be attached to AADL model elements

	Connectors	**Priorities**	**Behavior**	**Distributed**
C2	interface with each component via a separate port; interface elementare provided and required	low and high priority	consists of an invariant and a set of operations. The invariant is used to specify properties that must be true of all component states	yes
SADL	specifies the supported data types	scheduling of the process using a static priority	mathematical calculation	none
Rapide	connection; in-line	priority information for schedulability analysis	consists of set of transitions rule	yes
Darwin	binding; in-line; no explicit modeling of component interactions	priority information for schedulability analysis	using CORBA	yes
Unicon	connector	priority information for schedulability analysis	attributes for schedulability analysis	yes
AADL	connector (ports, parameters, data access)	security level	using subprograms; C/C++; ADA; state machine	yes

Fig. 2. Comparison between ADLs

using an annex. The behavioral annex describes a transition system attached to subprograms and threads.

3.1 AADL Components

Software Components AADL has the following categories of software components: subprogram, data, thread and process.

A *subprogram* component represents an execution entry-point in the source text. Subprograms can be called from threads and from other subprograms. These calls are handled sequentially by the threads. The *data* component type represents a data type in the source text that defines a representation and interpretation for instances of data. A *thread* represents a sequential flow of control that executes instructions within a binary image produced from source text. A thread always executes within a process. A scheduler manages the execution of a thread. A *process* represents a virtual address space. Process components are an abstraction of software responsible for executing threads.

Hardware Components Execution platform components represent hardware and software that is capable of scheduling threads, interfacing with an external environment, and performing communication for application system connections.

AADL *processor* components are an abstraction of hardware and software that is responsible for scheduling and executing threads. In other words, a processor may include functionality provided by operating systems. A *device* component represents an execution platform component that interfaces with the external environment. A device can interact with application software components through their ports. A *bus* components are used to describe all kinds of networks, buses, etc. A *Memory* components are used to represent any storage device: RAM, hard disk, etc.

Systems A system is the top-level component of the AADL hierarchy of components. A system component represents a composite component as an assembly of software and execution platform components. All subcomponents of a system are considered to be contained in that system.

3.2 Connections

A *connection* is a linkage that represents communication of data and control between components. This can be the transmission of control and data between ports of different threads or between threads and processor or device components.

4 From AADL to Distributed Implementation Using BIP

4.1 The BIP Component Framework

BIP (Behavior Interaction Priority) is a framework for modeling heterogeneous real-time components [8]. The BIP framework consists of a language and a toolset including a frontend for editing and parsing BIP programs and a dedicated platform for model validation. The platform consists of an Engine and software infrastructure for executing models. It allows state space exploration and provides access to model-checking tools of the IF toolset [12] such as Aldebaran [11], as well as the D-Finder tool [10]. This permits to validate BIP models and ensure that they meet properties such as deadlock-freedom, state invariants and schedulability. The BIP language allows hierarchical construction [16] of composite components from atomic ones by using connectors and priorities. Several case studies were carried out such as an MPEG4 encoder [23], TinyOS [9], and DALA [7].

4.2 Transformation from AADL to BIP

The AADL models are transformed into BIP automatically by using our AADL to BIP translation tool described in [13]. The supported development process is shown in the Figure 3.

The model construction methodology applied to AADL models, opens the way for enhanced analysis and early error detection by using BIP verifications techniques. Once the model has been generated, three model checking techniques for verification can be applied:

D-Finder: is an interactive tool for checking deadlock-freedom for component-based systems by using a static analysis method. It takes as input BIP programs and applies proof strategies to eliminate potential deadlocks by computing increasingly stronger deadlocks.

Model checking by Aldebaran: The second technique of verification is model-checking by using the tool Aldebaran [11]. Exhaustive exploration by the BIP exploration engine generates a Labeled Transition System (LTS) which can be analyzed by model checking. e.g, Aldebaran takes as input the LTS generated from BIP and checks for deadlock-freedom and other temporal properties.

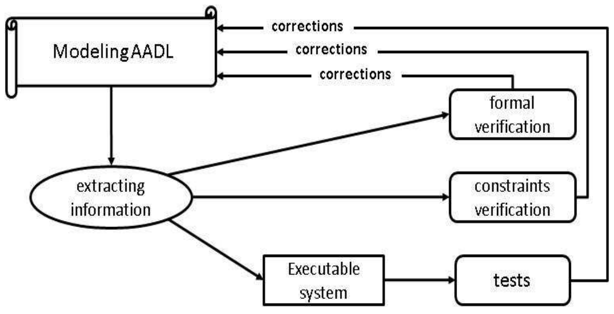


Fig. 3. Verification cycle

Model checking with observers: The third technique of verification is by using BIP observers to express and check requirements. Observers allow us to express in a much simple manner most safety requirements. We apply this technique to verify some properties as verification of communication, and verification of thread deadline.

Simulation & Debugging: In addition to the verifications, we can simulate or tests prototype implementations by creating an executable system. We can use an interactive simulation and debugger to verify each interaction step by step and to know which state or port is activated. These analysis allow to fully asses system viability, to refine and to correct the behavior of system.

Code generator: The code generator takes as input a model, generated by the parser, and transforms it to a C++ application code. The application is an executable model of the original BIP program. Code is generated for each atomic component, connectors and priorities, i.e., the code is modular and preserves the structure of the initial model.

4.3 Prototyping Distributed Implementation

Building distributed systems is a very tedious task since the application has to be verifiable and statically analyzable. The AADL fits these two requirements and allows the designer to describe different aspects of his distributed application (number of processors, number of threads in each processors, connection between threads...).

Requirement: Requirements for prototyping distributed embedded system can be seen as a collection of many requirements covering many domains. System designers and developers need to describe both functional and non-functional requirements. AADL support the different steps of system construction. Supported entities and extensible property sets allow one to build full models and adapt them to the application context. Furthermore, analysis tools can process the models to assess its viability.

Therefore, we list the following requirements for a prototyping process:

1. Data types and related functions to operate on them
2. Supporting runtime entities (threads) and interactions between them (through ports and connections)
3. Association of subprograms to threads
4. Mapping of threads onto processes and binding processes to hardware entities to form the deployed system.
5. Binding connections to buses to form the deployed system.

AADL allows us to refine the description of each entity to detail more precisely its behavior or some non-functional attributes. This allows us to have a library of reusable components and helps in prototyping by refining and extending them.

Deployement: The deployement we describe here supports all of the requirements discussed above. We begin with a model built by the application designer, who maps its application entities onto a hardware architecture. Then, we use AADL into BIP tool to generate BIP model conforming to AADL semantics. Finally, this architecture is tested for soundness, any mismatch in the application is reported by the analysis BIP tool chain.

AADL is expressive enough to detail the deployment view of the application: threads, processors, buses, threads on each process; properties refine the type of tasks (periodicity, priority), and their associated implementation. We defined our distribution model as a set of sender/receiver. It is supported by an AADL architectural model that defines the location of each system and the payload of the message exchanged as a thread-port name plus possible additional data.

Figure 4 shows the steps for generating from a distributed AADL system's description an executable distributed application as follow:

1. Identify each system and a connector's mapped to the bus.
2. Generate for each AADL system its corresponding description in BIP, and for each connector's mapped to the bus a communication protocol.
3. Compile BIP system's and generate an executable for each system with communication protocol.
4. Run and debug the distributed application.

Our protocol supports communication between two or more computers. It provide a full-duplex communication channel between processes that do not necessarily run on the same computer. We consider channels for data exchange among multiple threads in one or more processes are managed by the BIP Engine, if processes are running on one computer. Otherwise, if processes are running on different computers connected by a network, we use a network communication protocol. Before sending data through network to a server, we initially converted into encoded version before being transported (suitable for network transfer). After receiving data (Sever side), it can be converted back.

Most network communication protocols use the client server model. These terms refer to the two machines which will be communicating with each other. One of the two machines, the client, connects to the other machine, the server,

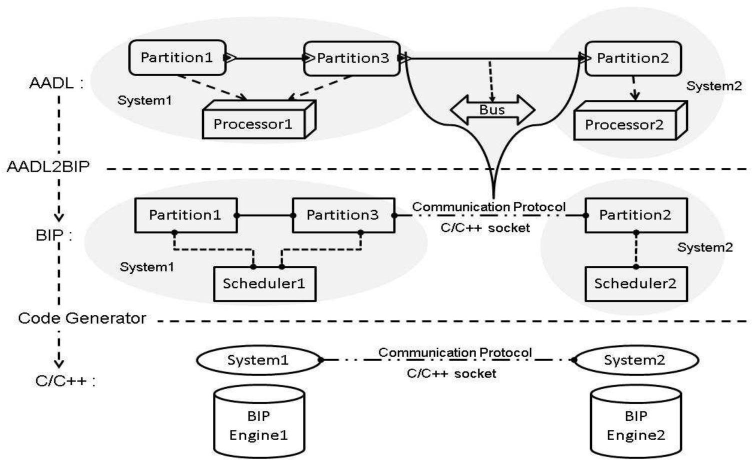


Fig. 4. Deployment

typically to make a request for information. Notice that the client needs to know of the existence of and the address of the server, but the server does not need to know the address of the client prior to the connection being established.

Our protocol use sockets. Sockets are associated with the concept of network communication in the form of client-server programming; a pair of processes of which one will be a client and one a server. The client process will send requests to the server. Of course, when creating a socket, we have to specify the type of communication that will be needed, since different modes of communication requires different protocols.

The steps involved in establishing a communication protocol on the client side are as follows: (1) Create a communication protocol; (2) Connect the communication to the address of the server; (3) Send and receive data.

The steps involved in establishing a communication protocol on the server side are as follows: (1) Create a communication protocol; (2) Bind the communication to an address. For a server, an address consists of a port number on the host machine; (3) Listen for connections; (4) Accept a connection. This call typically blocks until a client connects with the server; (5) Send and receive data.

The generated BIP code provides a framework that will directly call user code when necessary. This allows a rapid and flexible design of the distributed system and does not restrict the user implementations.

5 Case study: MPC (Multi-Platform Cooperation)

This case study has been inspired J. Hugues [18]. Figure 5 shows the software view of our case study. This model holds three system (Partitions); each is a spacecraft with different roles:

– *Spacecraft_1* is a leader spacecraft that contains a periodic thread, which sends its position to *Spacecraft_2* and *Spacecraft_3*.

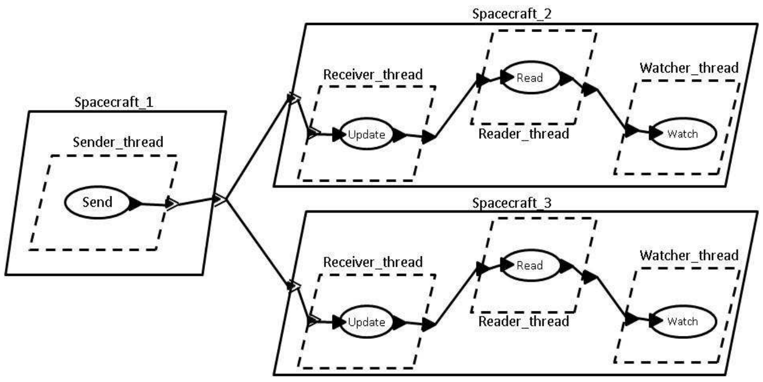


Fig. 5. Software view of the MPC case study

– *Spacecraft_2* and *Spacecraft_3* are follower spacecraft. They receive the position sent by *Spacecraft_1* with a sporadic thread (*Receiver_thread*), update their own position and sends the position to the *Reader_thread*. A *Reader_thread* in these two spacecraft reads periodically the position value from the *Receiver_thread* and store it in a protected object. A third thread "watches and reports" all elements at that position (e.g., earth observation).

This model gathers typical elements from distributed systems, with a set of periodic tasks devoted to the processing of incoming orders (*Watcher_thread*), *Reader_thread* to store these orders (Protected Object), and sporadic threads to exchange data (*Receiver_thread*). These entities work at different rates and should all respect their deadlines so that the *Watcher_thread* can process all observation orders in due time.

The software view only represents how the processing is distributed onto different entities (threads) and gathered as AADL processes to form partitions. The next step is to map this view onto a physical hardware view, so that Processor resources can be associated to each Partition.

Figure 6 is a graphical representation of the deployment view of the system. It only shows the global architecture of the application (number of partition and their mapping to hardware components). It indicates that each partition is bound to a specific Processor and how the communication between partitions occurs, using different buses.

These two views are expressed using the same modeling notation. They can be merged to form the complete system: interacting entities in the software view represent the processing logic of the system, whereas the hardware view completes the system deployment information by allocating resources.

5.1 AADL Models

MPC case study is built by creating software component and mapping entities onto a hardware architecture. The flexibility of AADL allows us to partially

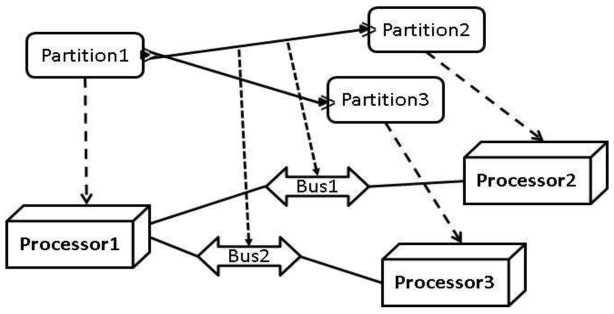


Fig. 6. Hardware view of the MPC case study

define components and use them in other components. This is very useful during the first steps of prototyping where every detail of the system is not yet clear. Details can be added to these components either by means of AADL properties or by component extension, without having to redefine all other components.

Data Types AADL data components model the messages that are exchanged among the Partitions of a distributed application or inside one of these Partitions. To express the kind of a data type, we use AADL data component as shown in the listing 1.1.

Subprograms Subprograms encapsulate the behavioral aspects of a distributed application. They are modeled using the subprogram AADL component. The implementation of a subprogram may be written entirely by the user by indicating the source file or the pre-built libraries that contain the implementation. Listing 1.2 shows the subprogram called *Update*.

```
data Record_Type
end Record_Type;


data implementation Record_Type.Impl
subcomponents
  X : data behavior::integer;
  Y : data behavior::integer;
  Z : data behavior::integer;
end Record_Type.Impl;
```

Listing 1.1. MPC data type

```
subprogram Update
  features
    Data_Sink: in parameter Record_Type;
    Protected : out parameter Record_Type;
end Update;

subprogram implementation Update.impl
  properties
    Source_Language => C;
    Source_Name => "Update";
    Source_Text => "mpc.cpp";
end Update.impl;
```

Listing 1.2. MPC subprogram

AADL subprograms can be modeled in several other ways. AADL2BIP allows three type of subprograms implementation by adding an external source file (C/C++), or by adding annex behavior specification, or by using subprogram calls sequence. All this gives the programmer more flexibility when prototyping his system.

Threads Threads are active parts of a distributed application. A Partition must contain at least one thread. The thread's interface consists of ports. In this case study we use two type of threads:

– Periodic threads, i.e., triggered by a time event (*Period*). Listing 1.3 shows the AADL model of the periodic thread *Sender_thread* that is located in the Partition1. This thread sends a data of type *Record_Type*. The dispatch protocol of the thread and its period are specified using standard AADL properties. In the thread implementation, we describe the behavior of the thread by giving the subprogram that models its activity.

```
thread Sender_Thread
features
  Data_Source : out  event data port Record_Type;
  Data_activate : in event data port Record_Type;
properties
  Dispatch_Protocol => Periodic;
  Period            => 100 Ms;
end Sender_Thread;

thread implementation Sender_Thread.Impl
calls Main: {
  Wrapper : subprogram Sender_Thread_Wrapper.impl;
 };
connections
  parameter Wrapper.Data_Source -> Data_Source;
  parameter Data_activate -> Wrapper.Data_activate;
end Sender_Thread.Impl;
```

Listing 1.3. MPC sender thread

– Sporadic threads. In this case, they are triggered by an incoming event. The AADL model of the sporadic thread *Receiver_thread* is located in *Spacecraft_2* and *Spacecraft_3* and is triggered by the reception of a position sent from *Spacecraft_1* by thread *Sender_thread*.

Processes Processes are the AADL components used to model the Partitions of distributed applications. Listing 1.4 shows the AADL model of the process called *Sender_Process*.

```
process Sender_Process
features
  Data_Source : out event data port Record_Type;
end Sender_Process;

process implementation Sender_Process.Impl
subcomponents
  Sender : thread Sender_Thread.Impl;
connections
  event data port Sender.Data_Source -> Data_Source;
end Sender_Process.Impl;
```

Listing 1.4. MPC Process: Spacecraft_1

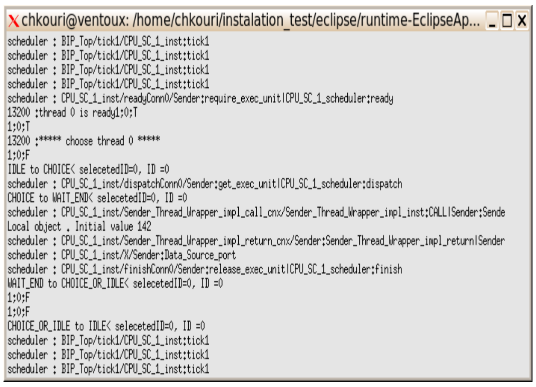
5.2 Deployment

The generation of BIP code helps us to rapidly prototype the MPC case study and make it to a distributed application using our communication protocol between each partition. The prototype helped us to analyse the case study in a native platform (PC) in order to easily debug and evaluate it before running it on an embedded platform.

The separation between software and hardware in AADL allows the programmer to model all the software parts of his application and test it with a native platform (generally a PC). If the tests are successful, the same software part can be reused with the actual hardware AADL. In addition, going from one hardware

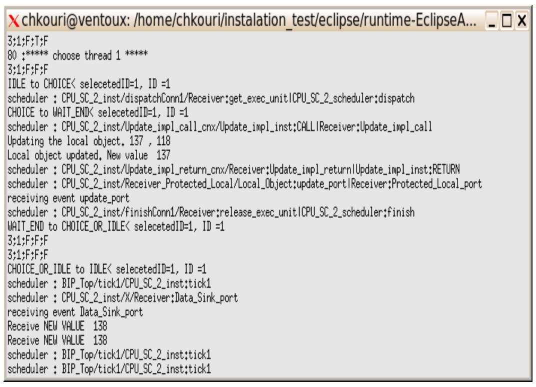
	AADL	BIP		
		Spacecraft_1	Spacecraft_2	Spacecraft_3
Components	20	4	8	8
Connectors	21	8	18	18
Lines of code	350	250	600	600

Fig. 7. Comparison between AADL & BIP



```
chkouri@ventoux: /home/chkouri/instalation_test/eclipse/runtime-EclipseAp...
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
scheduler : CPU_SC_1_inst/readyConn0/Sender:require_exec_unit|CPU_SC_1_scheduler:ready
13200 :thread 0 is ready1:0:T
1:0:T
13200 :***** choose thread 0 *****
1:0:F
IDLE to CHOICE< selecetedID=0, ID =0
scheduler : CPU_SC_1_inst/dispatchConn0/Sender:get_exec_unit|CPU_SC_1_scheduler:dispatch
CHOICE to WAIT_END< selecetedID=0, ID =0
scheduler : CPU_SC_1_inst/Sender_Thread_Wrapper_impl_call_cnx/Sender_Thread_Wrapper_impl_inst:CALL|Sender:Sende
Local object . Initial value 142
scheduler : CPU_SC_1_inst/Sender_Thread_Wrapper_impl_return_cnx/Sender:Sender_Thread_Wrapper_impl_return|Sender
scheduler : CPU_SC_1_inst/X/Sender:Data_Source_port
scheduler : CPU_SC_1_inst/finishConn0/Sender:release_exec_unit|CPU_SC_1_scheduler:finish
WAIT_END to CHOICE_OR_IDLE< selecetedID=0, ID =0
1:0:F
1:0:F
CHOICE_OR_IDLE to IDLE< selecetedID=0, ID =0
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
scheduler : BIP_Top/tick1/CPU_SC_1_inst:tick1
```

Fig. 8. Simulation of Spacecraft_1



```
chkouri@ventoux: /home/chkouri/instalation_test/eclipse/runtime-EclipseA...
3:1:F:T:F
80 :***** choose thread 1 *****
3:1:F:F:F
IDLE to CHOICE< selecetedID=1, ID =1
scheduler : CPU_SC_2_inst/dispatchConn1/Receiver:get_exec_unit|CPU_SC_2_scheduler:dispatch
CHOICE to WAIT_END< selecetedID=1, ID =1
scheduler : CPU_SC_2_inst/Update_impl_call_cnx/Update_impl_inst:CALL|Receiver:Update_impl_call
Updating the local object. 137 , 118
Local object updated. New value  137
scheduler : CPU_SC_2_inst/Update_impl_return_cnx/Receiver:Update_impl_return|Update_impl_inst:RETURN
scheduler : CPU_SC_2_inst/Receiver_Protected_Local/Local_Object:update_port|Receiver:Protected_Local_port
receiving event update_port
scheduler : CPU_SC_2_inst/finishConn1/Receiver:release_exec_unit|CPU_SC_2_scheduler:finish
WAIT_END to CHOICE_OR_IDLE< selecetedID=1, ID =1
3:1:F:F:F
3:1:F:F:F
CHOICE_OR_IDLE to IDLE< selecetedID=1, ID =1
scheduler : BIP_Top/tick1/CPU_SC_2_inst:tick1
scheduler : CPU_SC_2_inst/X/Receiver:Data_Sink_port
receiving event Data_Sink_port
Receive NEW VALUE  138
Receive NEW VALUE  138
scheduler : BIP_Top/tick1/CPU_SC_2_inst:tick1
scheduler : BIP_Top/tick1/CPU_SC_2_inst:tick1
```

Fig. 9. Simulation of Spacecraft_2

architecture to another is reduced (most of the time) to the modification of the values of some few AADL properties.

In the MPC case study, we generate for each AADL partition mapped to the processor, its corresponding description in BIP, and for each connection mapped to the bus a network communication protocol (sender/receiver). We compile BIP partitions and we generate an executable model. Then, we put every executable in the native platform (PC). First, we launch a receiver executable and then the sender executable. When the network protocol communication is initialized between the sender and receiver, the exchange of data is started.

Once the executable model has been launched, interactive simulation and debugging is useful for understanding the working of the distributed application. This helped us to verifies each interaction step by step, to know which state or port is activated, and to see the value of data received/sended. In addition, we use observers which moves to an error state if the period of a thread exceeds its deadline. These analysis allow to fully asses system viability, to refine and to correct the behavior of a system.

Figure 7 summarizes the size of lines of code, number of components and connectors in AADL and respectively the BIP code for the MPC case study. We split the BIP in three parts because we generate for each Spacecraft a corresponding BIP description system. Figures 8 and 9 show a fragment of the simulation of *Spacecraft_1* and *Spacecraft_2* in the distributed platform.

6 Conclusion

In this article, we proposed a prototyping process to model and build distributed embedded systems. We select AADL to implement this prototype. AADL allows a clear modeling structure and provides all the required information to configure a local application as well as distributed application.

We showed the requirements and assessments for prototyping distributed embedded system using our tools chain. In addition, we provide a general methodology for building and translating distributed embedded systems into an executable implementation by using network communication protocol. The executable application is tested for soundness, any mismatch in the application is reported by the analysis BIP tool chain. We provide also MPC case study, which is tested and analysed on a native platform.

In the future we are continuing to work on:

– Communication between processes can have different delay characteristics depending on the underlying communication network. The prototyping environment should support different delay characteristics for communication between different processes so that realistic prototypes can be built.
– Real-time clocks. This will allow real-time distributed algorithms to be implemented, and timing properties to be studied.

References

1. Annex Behavior Specification SAE AS5506.
2. ASSERT: http://www.assert-project.net/.
3. SAE. Architecture Analysis & Design Language (standard SAE AS5506), September 2004, available at http://www.sae.org.
4. SEI. Open Source AADL Tool Environment. http://la.sei.cmu.edu/aadlinfosite/OpenSourceAADLToolEnvironment.html.
5. SPICES: http://www.spices-itea.org/public/news.php.
6. TOPCASED: http://www.topcased.org/.
7. A. Basu, S. Bensalem, M. Gallien, F. Ingrand, C. Lesire, T.H. Nguyen, and J. Sifakis. Incremental component-based construction and verification of a robotic system. In *Proceedings of ECAI'08, Patras, Greece*, 2008.
8. A. Basu, M. Bozga, and J. Sifakis. Modeling heterogeneous real-time components in bip. In *Proceedings of SEFM '06, Pune, India*, pages 3–12. IEEE Computer Society, 2006.
9. A. Basu, L. Mounier, M. Poulhiès, J. Pulou, and J. Sifakis. Using bip for modeling and verification of networked systems – a case study on tinyos-based networks. In *Proceedings of NCA'07, Cambridge, MA USA*, pages 257–260, 2007.
10. S. Bensalem, M. Bozga, J. Sifakis, and T.H. Nguyen. Compositional verification for component-based systems and application. In *Proceedings of ATVA'08, Seoul, South Korea*, 2008.
11. M. Bozga, J-C. Fernandez, A. Kerbrat, and L. Mounier. Protocol verification with the aldebaran toolset. *STTT*, 1:166–183, 1997.
12. M. Bozga, S. Graf, Il. Ober, Iul. Ober, and J. Sifakis. The if toolset. In *Proceedings of SFM'04, Bertinoro, Italy*, volume 3185 of *LNCS*, pages 237–267.

13. M.Y Chkouri, A. Robert, M. Bozga, and J. Sifakis. Translating AADL into BIP - Application to the Verification of Real-Time Systems. In *Models in Software Engineering: Workshops and Symposia at MODELS 2008, Toulouse, France, September 28 - October 3, 2008.*, pages 5–19.
14. P. C. Clements. A survey of architecture description languages. In *In Proceedings of the Eighth International Workshop on Software Specification and Design, Paderborn, Germany*, 1996.
15. L. M. Augustin J. Vera D. Bryan D. C. Luckham, J. J. Kenney and W. Mann. Specification and analysis of system architecture using rapide. In *IEEE Transactions on Software Engineering*, volume 1 no.4, pages 336–335, 1995.
16. J. Sifakis G. Gossler. Composition for component-based modeling. *Science of Computer Programming*, 55:161–183, March 2005.
17. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. Rapid Prototyping of Distributed Real-Time Embedded Systems Using the AADL and Ocarina. In *Proceedings of the 18th IEEE International Workshop on Rapid System Prototyping (RSP'07)*, pages 106–112, Porto Alegre, Brazil, May 2007. IEEE Computer Society Press.
18. J. Hugues, B. Zalila, L. Pautet, and F. Kordon. From the prototype to the final embedded system using the ocarina aadl tool suite. *ACM Trans. Embed. Comput. Syst.*, 7(4):1–25, 2008.
19. J. Magee and J. Kramer. Dynamic structure in software architectures. In *In Proceedings of ACM SIGSOFT'96: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 3–14, 1996.
20. N. Medvidovic. A language and environment for architecture-based software development and evolution. In *In Proceedings of the 1999 International Conference on Software Engineering*, pages 44–53, 1999.
21. N. Medvidovic, P. Oreizy, J.E. Robbins, and R.N. Taylor. Using object-oriented typing to support architectural design in the c2 style. In *In Proceedings of ACM SIGSOFT201996: Fourth Symposium on the Foundations of Software Engineering (FSE4)*, pages 24–32. ACM Press, 1996.
22. M. Moriconi and R. A. Riemenschneider. Introduction to sadl 1.0: A language for specifying software architecture hierarchies. In *Technical Report SRI-CSL-97-01, SRI International*, 1997.
23. M. Poulhiès, J. Pulou, C. Rippert, and J. Sifakis. A methodology and supporting tools for the development of component-based embedded systems. In *13th Monterey Workshop, Paris, France*, volume 4888 of *LNCS*, pages 75–96, 2006.
24. M. Shaw, R. Deline, D.V. Klein, T.L. Ross, D.M. Young, and G. Zelesnik. Abstractions for software architecture and tools to support them. *IEEE Transactions on Software Engineering*, 21:314–335, 1995.
25. R.M. Taylor and N. Medvidovic. A classification and comparison framework for software architecture description languages. *IEEE Transactions on Software Engineering*, 26:70–93, 2000.
26. S. Sendall V. Crettaz, M.M. Kand and A. Strohmeier. Integrating the concernbase approach with sadl. In *In Proceedings 4th International Conference on Modeling Languages, Concepts, and Tools .Toronto, Canada*, pages 166–181, 2001.
27. T. Vergnaud, J. Hugues, L. Pautet, and F. Kordon. PolyORB: a schizophrenic middleware to build versatile reliable distributed applications. In *Proceedings of the 9th International Conference on Reliable Software Techologies Ada-Europe 2004*, volume LNCS 3063, pages 106 – 119, Palma de Mallorca, Spain, Jun.
28. S. Vestal. A cursory overview and comparison of four architecture description languages. In *Technical Report, Honeywell Technology Center*, 1993.

Towards Intelligent Tool-Support for AADL Based Modeling of Embedded Systems

Dries Langsweirdt, Yves Vandewoude and Yolande Berbers

Katholieke Universiteit Leuven, Department of Computer Science, Celestijnenlaan 200A, B-3001 Leuven, Belgium
{dries.langsweirdt, yves.vandewoude, yolande.berbers}@cs.kuleuven.be

Abstract. Model-driven design (MDD) of complex embedded systems is currently based on successive cycles of model changes, analysis and simulation. This iterative process suffers from a delay between applying changes on the model and knowledge about the resulting properties of the system. Current research on Architecture Discription Languages (ADL) in general, and AADL in specific, focuses primarily on tools and support for analysis and simulation, as distinct phases during design. We give an overview of existing work on AADL, and illustrate though a case study the opportunities for a novel, integrating research domain on ADL.[*]

1 Introduction

Model-driven design (MDD) helps system architects master the complexities associated with the development of large and multi-concern embedded systems. Architecture Description Languages (ADL) are hereby a primary way to model the system components and their interactions. ADL specific analysis and simulation tools applied on these models can estimate the properties and the behavior of the final system, and thus predict if the system will meet its requirements. Deviations between analysis and simulation results on the one hand, and expected properties and behavior on the other, lead to model changes. As such, the design follows an iterative scheme. Changing the model implies changing the properties of the system, be it functional or non-functional. However, the relation between a model change and the resulting properties is often unclear until the subsequent analysis and simulation phase, making this iterative cycle unnecessarily long. There is a clear need for tools able to seamlessly integrate the modeling, analysis and simulation phases as to assist the architect in decision making through direct feedback on model changes. Formalization of domain knowledge and architectural patterns are prime concerns in this context. This paper identifies the need for research addressing these concerns, and does so in

[*] This work has been carried out as a part of the Condor project (`http://www.esi.nl` Projects->Condor) at FEI company under the responsibilities of the Embedded Systems Institute (ESI). This project is partially supported by the Dutch Ministry of Economic Affairs under the BSIK program.

the concrete context of the Architecture Analysis and Design Language (AADL) [1, 2] as a prime example of a Real-Time/Embedded (RT/E) ADL.

This paper is organized as follows. Section 2 gives an overview of the current work on AADL. Section 3 introduces an example used to pinpoint the current technical barriers, and is a first incite towards more intelligent tool-support. In section 4, we discuss our current ideas and provide a possible roadmap for future research. Conclusions are drawn in the final section.

2 Overview of current work on AADL

Four research domains applicable to AADL are currently under active development: front-end processing, code generation, analysis and simulation. Figure 1 shows a classification of the most important initiatives, which are discussed next. OSATE [3] targets front-end processing and semantic checking of AADL models, with two possibilities to extend its capabilities into the analysis and simulation domain. First, plugins can be build on top of OSATE's functionality, allowing for custom analysis on OSATE resident models. Second, models can be exported in an XMI schema, which allows for analysis on the models by external tools. The TOPCASED [4] project integrates with OSATE to visualize the AADL models. Ocarina [5] is an Ada tool suite with the ability to generate the infrastructural code in Ada or C of a distributed, real-time and high-integrity application from an AADL specification. Ocarina links the generated applications with the high-integrity middleware libraries PolyORB-HI-Ada and -C, derived from the PolyORB [6] project. STOOD [7] offers the embedded engineer multiple modeling paradigms: UML2.0, HRT HOOD and AADL 1.0. Like Ocarina, STOOD generates Ada and C code from AADL models, but the generated applications are not distributed. Cheddar [8] is a framework for schedulabilty analysis. CPN-AMI [9] is a CASE environment able to analyze and simulate Petri-net based models. Both projects are independent of AADL, but provide a good example of how the aforementioned XMI scheme (together with appropriate model transformations) can bring external analysis tools to the AADL scene. ADeS [10] aims at behavioral simulation of AADL models. The goal is to implement the entire AADL Behavior Model Annex in the simulation kernel Jimex, but currently ADeS only implements a simple behavioral model on threads. In contrast, AADS [11] transforms a subset of AADL to SystemC for simulation. Building on the SCoPE [12] project, AADS is well suited for HW/SW co-design. Finally, [13] proposed execution of AADL models based on a translation to the synchronous languages Scade and Lustre.

Only minor integration between the different domains is noted. Intelligent integration of the available work to assist the system architect during the actual act of modeling is absent, and is why we suggest a fifth domain on design support (see figure 1). Intelligent in this context refers to the availability of formalized knowledge and patterns specific to the embedded domain. Reasoning algorithms could apply this knowledge on the concrete, but potentially incomplete, models to deduct architectural suggestions, warnings and optimal properties. A more concrete discussion can be found under section 4.

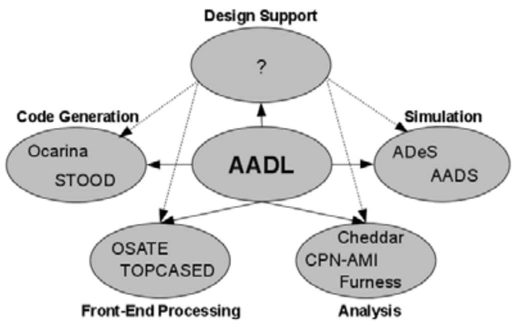


Fig. 1: Overview of the four existing and proposed fifth domain on AADL.

3 Case Study

We present the extension of an existing OSATE plugin as an illustration of more integrated modeling support. It also identifies current problems with analysis on unfinished, declarative AADL models.

3.1 System Specification Versus Instantiation

AADL differentiates between system specification and instantiation. A system is defined as completely instantiable if: “the system implementation being instantiated is completely specified and completely resolved”. The tools presented in section 2 almost exclusively work on instantiated models, and thus complete with respect to the compositional and legality rules of AADL. They are incapable of gathering information from, and act on, declarative models.

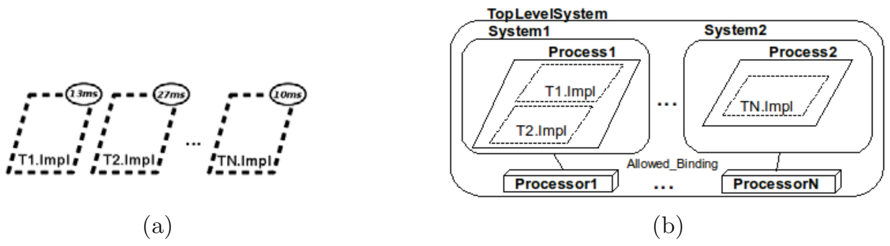


Fig. 2: Automated model completion through bin-packing and scheduling from (a) the isolated thread set, to (b) the instantiable system implementation.

3.2 Automated Bin-packing and Scheduling

The case study extends OSATE's bin-packing and scheduling plugin, which is an implementation of the work done on partitioned bin-packing algorithms by de Niz et al. [14, 15]. The plugin automates the assignment of threads to processors available in an instantiated AADL model. We extended the plugin to work on declarative models as well. Provided with a minimum of information (a thread set and its properties, with optionally a processor and/or bus used as templates), our plugin calculates the amount of needed processors and links to make the threads schedulable. The plugin then extends the declarative model bottom-up, based on the outcome of the analysis, to a completely instantiable model with bounded threads. This model transformation is illustrated in figure 2.

3.3 OSATE Deficiencies and AADL Intricacies

Although the case study is limited, we noted the following four interesting problems. First, the API of OSATE with respect to extensive manipulations of declarative models could be much improved upon. One example is the asymmetry between addition and removal of component types or implementations. For removal, the programmer needs to rely directly on the Eclipse Ecore infrastructure. Second, because of the AADL semantics, information can be scattered throughout the declarative model in complex ways. Instantiated models do not have this problem as such, because the relations between components are fixed and their properties can be referenced directly. Gathering information on the declarative model quickly results in multiple model scans, potentially leading to a scalability issue in the current OSATE implementation. Finally, two forms of ambiguity in the use of the AADL standard (first noted by Delanote in [16]) complicate the automated extension and analysis of an unfinished model. First, the legality rules are different for each component category, making component composition not only complex, but also ambiguous. For example, the model extension in the case study wraps each process in a separate system component. There are however multiple other legal ways with respect to the standard to complete the model, without the advantages of one approach over another being obviously clear. Second, there is no well defined relation between the analysis of certain system properties, and the AADL model properties needed to conduct it. With these relations being unclear, it becomes hard to discover missing information in the model.

4 Roadmap

As mentioned in section 2, key to assisted modeling is formalization and embedding of domain specific knowledge and architectural patterns, together with appropriate reasoning algorithms, in the tools the system architect uses to construct ADL models. Concrete properties and interconnections of model components can as such be linked with corresponding analysis and simulation routines.

The results of these routines, automatically invoked on each incremental model change, can be fed back to the architect in the form of suggestions on architectural changes. If appropriate, changes can be performed automatically by the tool, already illustrated in the case study. Note that, as stated in section 3.1 and 3.3, analyzing an incomplete model is a non-trivial task making analysis and simulation adaptation, or virtual model completion, a necessity. The first challenge, formalization, is in principle ADL-agnostic, and depends on the RT/E domain in general. Harvesting model properties and performing model changes on the other hand, depend on the legality and compositional rules of the concrete ADL. Both challenges are interesting tracks that can be addressed by the proposed domain on design support.

5 Conclusion

This paper discusses the need for more research focusing on support for the embedded systems architect during modeling, compared to analysis of a certain made choice. Through an AADL case study, we identified some of the existing barriers and defined a roadmap for future research.

References

1. Feiler, P., Gluch, D., Hudak, J.: AADL: An Introduction. Tech. rep., SAE (2006)
2. SAE: Architecture Analysis & Design Language (AS5506A), `http://www.sae.org`
3. Open Source AADL Tool Environment (OSATE). Techn. rep., SEI (2006)
4. The Open-Source Toolkit for Critical Systems, `http://www.topcased.org/`
5. Hugues, J., Zalila, B., Pautet, L.: From the Prototype to the Final Embedded System Using the Ocarina AADL Tool Suite. In: ACM TECS 7 No.4, Art.42 (2008)
6. Vergnaud, T., Hugues, J., Pautet, L.: PolyORB: A schizophrenic middleware to build versatile reliable distributed applications. LNCS, vol. 3063, pp 106–119. Springer, Heidelberg (2004)
7. Ellidiss-Software: STOOD, `http://www.ellidiss.com/stood.shtml`
8. Singhoff, F., Legrand., J., Tchamnda, L.: Cheddar: A flexible real time scheduling framework. J. ACM Ada Lett. 24, 1–8 (2004)
9. The CPN-AMI home page, `http://www.lip6.fr/cpn-ami`
10. ADeS: a simulator for AADL, `http://www.axlog.fr/aadl/ades_en.html`
11. Varona-Gmez, R., Villar, E.: AADL Simulation and Performance Analysis in SystemC. In: IEEE ICECCS, pp. 323–328. IEEE Computer Society, Potsdam (2009)
12. SCoPE v1.0.0 UC 2008, `http://www.teisa.unican.es/scope`
13. Jahier, E., Halbwachs, N., Lesens, D.: Virtual execution of AADL models via a translation into synchronous programs. In: Proceedings of the 7th ACM&IEEE ICESS, pp 134–143. ACM, NY (2007)
14. de Niz, D., Rajkumar, R.: Partitioning Bin-Packing Algorithms for Distributed Real-Time Systems. I. J. of Embedded Systems 2 No.3/4, 196–208 (2006)
15. de Niz, D., Bhatia, G., Rajkumar, R.: Model-Based Development of Embedded Systems: The SysWeaver Approach. In: 12th IEEE RTAS, pp. 231–242. IEEE Computer Society, San Jose (2006)
16. Delanote, D., Van Baelen, S., Joosen, W., Berbers, Y.: Using AADL in Model Driven Development. UML&AADL'2007, ICECCS07. IEEE, Auckland (2007)

Model-Based Codesign of Critical Embedded Systems*

Marco Bozzano[1], Alessandro Cimatti[1], Joost-Pieter Katoen[2], Viet Yen Nguyen[2], Thomas Noll[2], and Marco Roveri[1]

[1] Fondazione Bruno Kessler, Italy
{bozzano,cimatti,roveri}@fbk.eu
[2] RWTH Aachen University, Germany
{katoen,nguyen,noll}@cs.rwth-aachen.de

Abstract. We present a comprehensive methodology for the specification and analysis of critical embedded systems. The methodology is based on an architectural design language that enables modeling of both software and hardware components, timed and hybrid behavior, faulty behavior and degraded modes of operation, error propagation and recovery. The methodology is supported by an integrated platform, implemented on top of state-of-the-art tools, that provides verification capabilities ranging from requirements analysis to functional verification, safety assessment, performability evaluation, diagnosis and diagnosability.

1 Introduction

The design of critical embedded systems is a very complex and highly challenging task, for a number of reasons. First, it requires designing and assembling heterogeneous components, implemented either in hardware or in software, and their interactions. Secondly, it has to take into account functional requirements as well as several sorts of non-functional requirements, such as (real-)time constraints, performability and safety requirements.

In this paper we present a comprehensive and tool-supported methodology for the design of critical systems, following the component-based paradigm. Component-based design helps to master design complexity while, at the same time, allowing for reusability. The key principle is a clear distinction between component behavior (implementation) and the interactions between the individual components (interfacing). The internal structure of a component implementation is specified by its decomposition into subcomponents, together with their hardware/software bindings and their interaction via connections over ports.

The design methodology is built on top of the SLIM modeling language, an architectural language inspired by SAE's AADL [9] (Architecture Analysis and Design Language) and the related Error Model Annex [10]. SLIM inherits the most important features of AADL, such as multiway communication, dynamic

* Funded by ESA/ESTEC under Contract No. 21171/07/NL/JD

reconfiguration of components and port connections, and probabilistic error behavior and propagation, while enriching it with constructs to express timed and hybrid behavior. Moreover, the SLIM language is endowed with a formal semantics that cover all of its aspects in a clear and unambiguous way [3].

The methodology proposed in this work is targeted at the architectural design of critical embedded systems, and in particular it covers modeling and verification of the following aspects: requirements analysis, verification of functional correctness, safety assessment and fault tolerance measures, quantitative and performability analysis, and partial observability analysis, including effectiveness of the FDIR (Fault Detection, Identification and Recovery) components.

The proposed approach is being investigated in the COMPASS project[3] (Correctness, Modeling, and Performance of Aerospace Systems) in the aerospace domain, and results as a response to an invitation to tender by the European Space Agency. The techniques described in this work, however, are applicable in general to every domain where design of critical embedded systems is involved.

The paper is structured as follows. In Section 2 we describe the main features of the SLIM language; in Section 3 we give an overview of the methodology; in Section 4 we discuss the COMPASS tool, implementing the methodology, and finally we draw some conclusions and discuss future directions in Section 5.

2 The SLIM Language

The SLIM language follows the component-based paradigm. In SLIM, it is possible to refer to both software (e.g. threads and processes) and hardware components (e.g. memories and processors) as first-class objects. Each component is given via its type, describing the interface, and its implementation, describing the interactions via a finite state automaton. Sets of interacting components can be grouped into composite components, enabling the modeler to manage the system's complexity by introducing a component *hierarchy*. Communication is achieved via exchange of messages on event ports, in a rendez-vous manner. Moreover, components may exchange data through typed data ports (e.g. bool, integer and real data types). Timed and hybrid behavior can be expressed by means of real-valued variables with (linear) time-dependent dynamics.

The resulting hierarchical system model, also referred to as *nominal model*, describes the system behavior under normal operation. This is complemented by an *error model* which expresses how the system can fail. Moreover, a subset of the nominal components may be designated as dealing with error diagnosis and recovery; they are referred to as FDIR (Fault Detection, Identification and Recovery). The error model expresses how faults may affect normal operation and may lead the system into a degraded mode of operation. It is modeled as a probabilistic finite state automaton, where transitions may occur due to error events which may be annotated with a rate that indicates the expected number of occurrences per time unit. Transitions can also occur because of error propagations from other components. The nominal and error models are linked through

[3] http://compass.informatik.rwth-aachen.de

a so-called *fault injection*. A fault injection expresses the effect of the occurrence of the corresponding error on the nominal model. Multiple fault injections are possible. The process of integrating the nominal models with the error models and the fault injections, is called *model extension* [4]. Finally, in order to enable modeling of partial observability and analysis of FDIR components, the SLIM language allows the modeler to explicitly define a set of observables.

We refer to [3] for a more detailed description of the language, a discussion of the similarities and extensions with respect to AADL, and a simple example (a processor failover system). Moreover, [3] presents a formal semantics for all the language constructs, based on networks of event-data automata (NEDA).

3 Methodology

The methodology discussed in this paper is inspired by the framework described in [1], which provides a unifying view of different aspects of system engineering, within the context of model checking. In order of increasing complexity, the first problem that we consider is system functional correctness. Functional requirements are traditionally expressed in temporal logic, e.g. Computation Tree Logic (CTL) or Linear Temporal Logic (LTL). Technologically, model checking techniques are used to exhaustively explore every possible system behavior, providing a formal guarantee that a given requirement is obeyed.

Safety analysis investigates the behavior of a system in degraded conditions, that is, when some parts of the system are not working properly due to malfunctions. It includes hazard analysis, whose goal is to identify all the hazards of the system and ensure that the system meets the safety requirements that are required for its deployment and use. Examples of hazard analysis techniques are Fault Tree Analysis (FTA) and Failure Mode and Effects Analysis (FMEA). Model-based safety analysis is in turn based on model checking techniques [4].

Quantitative analysis and performability aim at evaluating system performance with respect to timed and probabilistic requirements. They also include probabilistic versions of safety and diagnosability measures. The related requirements can be expressed in Continuous Stochastic Logic (CSL). The implementation of these analyses is based on probabilistic model checking techniques.

Diagnosis can be seen as the problem of safety analysis carried out at run-time. It is usually performed on systems which provide limited run-time sensing, and under the hypothesis of partial observability. Diagnosis starts from the observed run time behavior of a system, and tries to provide an explanation (in terms of hidden states). In particular, diagnosis is often the problem of identifying the set of possible causes of a specific unexpected or faulty behavior. Probabilistic information can be taken into account, in order to search for the most likely explanation. Another related problem is diagnosability, i.e., the analysis, at design time, of diagnosis capabilities. Finally, the problem of synthesis consists in the automatic generation of controllers from specifications. The latter problem has been tackled by planning techniques based on model checking.

Finally, requirements validation is used to check correctness and completeness of a set of properties. Requirements validation is performed before the system architectural design starts, and has the goal of ensuring the quality of system requirements. In particular, our approach enables checking for logical consistency, i.e., freedom from contradictions. Moreover, it is possible to check whether a given set of properties is strict enough to rule out unwanted behavior, and not too strict to disallow for certain desirable behavior.

4 Tool Support

The methodology is supported by an integrated toolset, which is built on top of existing state-of-the-art tools for formal verification, based on model checking. In particular, the toolset builds upon the NuSMV [7] symbolic model checker, the MRMC [6] probabilistic model checker, and the RAT [8] requirements analysis tool. The architecture of the tool set is shown in Fig. 1. The toolset takes as input a model written in the SLIM language, and a set of property patterns, used to instantiate formal requirements. Depending on the context, instantiated properties are expressed in CTL, LTL or CSL temporal logics. A few building blocks take care of performing *model extension*, translating the SLIM input model into NuSMV and MRMC formats when needed, and visualize traces and fault trees.

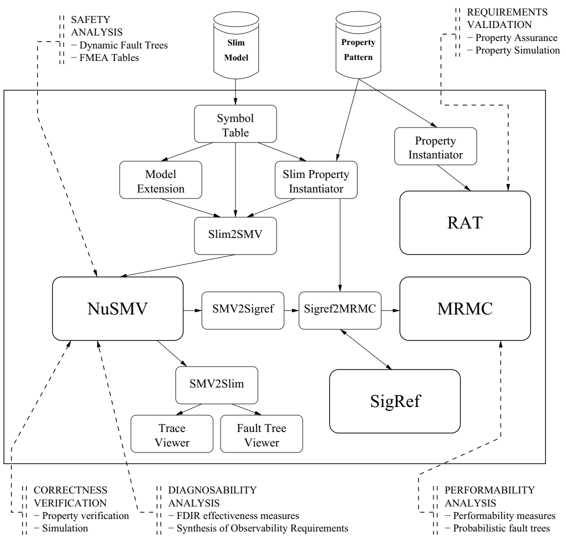


Fig. 1. Architecture of the toolset.

The following analyses are supported. Requirements validation is used to analyse the quality (correctness and completeness) of the requirements, and is carried out by the RAT tool. Correctness verification focuses on verification of functional requirements, and is implemented on top of NuSMV; NuSMV implements standard symbolic model checking techniques such as BDD-based and SAT-based (bounded) model checking, as well as SMT (Satisfiability Modulo Theory)-based techniques to deal with hybrid models. Safety analysis supports two of the most popular hazard analysis techniques, namely FTA and FMEA, that are carried out by FSAP [5], a plugin of NuSMV. Diagnosability analysis focuses on the evaluation of the effectiveness of the FDIR sub-system; these functionalities are built on top of NuSMV and FSAP. Finally, performability analysis evaluates a SLIM model with respect to probabilistic requirements; it is implemented on top of MRMC. For more information on the toolset, its architecture, and the analyses that are supported, we refer to [2].

5 Conclusions

In this paper we have presented a comprehensive methodology and a toolset for the specification and analysis of critical embedded systems, that focuses on system features such as (real-)time and faulty behavior, degraded modes of operation, diagnosis and performability. The methodology and toolset are currently being evaluated on industrial-size case studies from the aerospace domain, that will provide a substantial insight into their applicability and effectiveness.

Our methodology is applicable to any domain where, e.g., timing, system performance and safety are at stake. Examples are avionics, transportation, including railways and automotive, power plants, and the medical domain. Our approach is based on a general purpose architectural language, and it is especially targeted at modeling and analyzing systems designs at the architectural level. It can be complemented by specific implementation-level languages to deal with the most implementation-oriented features of system design.

The toolset is under active development and evaluation. A thorough experimental evaluation is planned, based on a comprehensive set of case studies.

Finally, some of the modifications to the AADL language that have been incorporated into SLIM, have been brought into the AADL standardization bodies for evaluation and proposed as a possible extension of the standard.

References

1. P. Bertoli, M. Bozzano, and A. Cimatti. A Symbolic Model Checking Framework for Safety Analysis, Diagnosis, and Synthesis. In *Model Checking and Artificial Intelligence*, volume 4428 of *LNCS*, pages 1–18. Springer, 2007.
2. M. Bozzano, A. Cimatti, J.-P. Katoen, V. Y. Nguyen, T. Noll, and M. Roveri. The COMPASS Approach: Correctness, Modelling and Performability of Aerospace Systems. In *Proc. SAFECOMP'09*. Springer, 2009.
3. M. Bozzano, A. Cimatti, V. Y. Nguyen, T. Noll, J. P. Katoen, and M. Roveri. Codesign of Dependable Systems: A Component-Based Modeling Language. In *Proc. MEMOCODE '09*, 2009.
4. M. Bozzano and A. Villafiorita. The FSAP/NuSMV-SA Safety Analysis Platform. *International Journal on Software Tools for Technology Transfer*, 9(1):5–24, 2007.
5. The FSAP/NuSMV-SA platform. `http://sra.fbk.eu/tools/FSAP`.
6. The MRMC model checker. http://wwwhome.cs.utwente.nl/ zapreevis/mrmc/.
7. The NuSMV model checker. `http://nusmv.fbk.eu`.
8. RAT: Requirements Analysis Tool. `http://rat.fbk.eu`.
9. Architecture Analysis and Design Language (AADL) V2. SAE Draft Standard AS5506 V2, International Society of Automotive Engineers, Mar. 2008.
10. Architecture Analysis and Design Language Annex (AADL), Volume 1, Annex E: Error Model Annex. SAE Standard AS5506/1, SAE International, June 2006.

Design Complexity Management in Embedded System Design

Johan Ersfolk[1,2], Johan Lilius[2], Jari Muurinen[3], Ari Salomäki[3], Niklas Fors[2], and Johnny Nylund[2]

[1] Turku Centre for Computer Science, Turku, Finland
[2] Department of Information Technologies
Åbo Akademi University, Turku, Finland
FirstName.LastName@abo.fi
[3] Nokia Devices, Finland
FirstName.LastName@nokia.com

Abstract. Research on embedded system design typically focus on design space exploration in the architecture platform space and the goal is to obtain an optimal implementation of the system. In the mobile phone industry the design problem is often quite different. The goal is not to design a new system but to add a use case to an existing product or to a family of products. In this case it is important to be able to quickly find possible performance problems caused by the simultaneous use of the new use case in conjunction with existing use cases on all platforms. In this paper we address this problem by 1. proposing a structure for the design space, 2. an automated algorithm that generates performance models by combining use case models, and 3. an approach for performance optimization by adding flow control elements into the system design.

1 Introduction

Existing embedded system design methodologies focus on design space exploration in the architecture platform space. That is to say, they assume that the set of applications is fixed and a suitable architecture for this set of applications needs to be explored. In the mobile phone industry the design problem is often quite different and the situation is usually that there is a number of fixed platforms for which new applications are being developed using libraries of existing software components. This often leads to a situation where the concurrent execution of a set of applications needs to be simulated on a number of architecture platforms in order to analyse the resource sharing between the applications. In order to make the evaluation of such designs efficient there is a need for exploring how existing design methodologies and tools can be extended with functionality that addresses the problem of efficiently combining software components. In this paper we approach this issue with a model driven approach using our metamodeling tool Coral [1].

The design flow depicted in figure 1 highlights the communication between a system architect and the teams working on the different subsystems. The typical scenario in which this design flow is instantiated is when a set of new use cases needs to be implemented. This would involve for example adding video playback (a use case) and

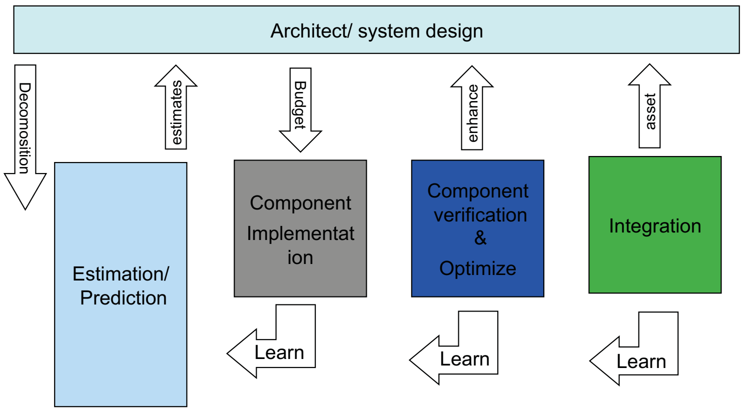


Fig. 1. A design flow and its feedback loops

video recording capabilities (a second use case) to a phone. The design will not proceed by trying to build a new system from scratch, but instead the goal is to find the minimal changes to an existing architecture to implement the new use cases.

The design would proceed approximately like this. The system architect takes the new use case and decomposes the system into subsystems. For the existing system the subsystems are available as *assets* in a library, from which relevant performance data can be obtained. For the required new subsystems, the system architect requests estimates from the designer team responsible for the technical subsystem (e.g. for the video encoding from the media subsystem team). Using these estimates the system architect can start evaluating the system model for its performance. At this point in the design flow it is important to obtain quick results. Therefore the individual elements in the system model are often quite abstract and focus only on the performance characteristics.

It is a key requirement to be able to evaluate different *use case combinations* quickly. Most often all use cases are not used at the same time, e.g. video playback might not be used at the same time with a voice call, if the phone does not support video calls, but video playback might happen at the same time as a file download. Therefore, it is important to know which use cases can be used in combination and to analyze the combinations for potential performance bottlenecks. The performance bottlenecks are typically caused by use cases sharing resources. In some cases it is not possible to run a specific use case combination on a platform which means that the platform needs to be modified. More often the problem of sharing a resource is due to stochastic behavior of data streams and badly tuned mechanisms for handling the resource contentions. In order to resolve resource conflicts and find the optimal parameters for the system the system architect can use different flow control mechanisms. In section 5 we describe different techniques and how these can be used.

When the system architect has found parameters that fullfill the performance requirements the design flow continues based on this validated information. The obtained values are given back to the designers as a *budget*, to use in the implementation of the subsystem. The verification of the implementation will give some feedback to the system architect, which may lead him to make changes to the design. Once the component is deemed ready it is integrated into the final product. At this stage the component becomes an *asset*, which means that the component has been deployed in a finished product, and proven to work in conjunction with the other components in the use cases.

Since the system architect has an overall responsibility over the design process, he needs powerful and efficient (i.e. rapid) tools, based on dedicated analysis models, to support him in his design work. In section 4 we explain how such models can be obtained automatically. The requirement for the automatic use case combination is that the models have a specific structure, and such structuring mechanisms will be presented in section 3. In section 6 we shortly present a front-end tool (EFCO Tool) to CoFluent Studio [2].

Due to space limitations many pictures and details have to be omitted. A more thorough description of the approach can be found in [3, 4].

2 Related Work

There is a large number of tools and methods for design space exploration and it is not possible to mention all of them here. We will focus on the most relevant approaches and we will also describe the approach used in CoFluent Studio and compare it to our. In [5] several different methodologies intended to be used in the field of system-level design are discussed and compared.

The Y-chart approach [6] separates the application models from the architecture instance models. A set of architecture instance models can be evaluated against a set of application models and the models can be reused separately in other projects. For a given architecture instance a performance model needs to be created. Performance analysis for a specific architecture instance can then be done after the set of applications have been mapped to the architecture instance. The results from this performance analysis can be used to make improvements on the architecture instance, the applications themselves or on the mapping between application and architecture instance. This process can then be repeated until an architecture that satisfies all constraints is found. The Y-chart approach does not specifically deal with software reuse in any way. If effectively separates architecture design from application design, but it does not guarantee that the applications can be reused easily for architectures or combined with other applications. What the Y-chart approach does for software reuse is that it specifies a structured method to map a set of applications to an architecture and simulate the results.

Another approach that focus even more on component reuse in the hardware part of the system is Platform-based design [7]. Platform-based design is an approach to embedded system design where refined specifications meet with abstractions of possible architecture implementations [7]. Platform-based design identifies well defined layers in the design process where the abstractions and refinements are done. Each abstraction layer must give enough information about lower levels of abstraction upwards, so that design space exploration can take place. Furthermore, constraints from higher levels of abstraction need to be passed down to lower levels of abstraction so that the refinement process can take place between layers. The difference to our aproach is that the set of applications is assumed to be fixed and that a suitable architecture for this set of applications needs to be explored.

CoFluent Studio [2, 8] is an embedded system design tool that enables performance analysis of hardware/software systems, by using the Y-chart approach. Consistent with the Y-chart approach, CoFluent Studio separates the functional model of the system from the architectural model of the system. By separately describing an application model and a platform model, and then mapping the two models together, an architecture model can be obtained. CoFluent Studio supports simulation of these models through automatically generating a SystemC test bench for performance analysis.

Functional design in CoFluent Studio is done by specifying a functional model of the complete system using a combination of a graphical notations and C code. The functional model can graphically be represented using structural and behavioral components called "functions". Structural functions can contain other structural functions as well as behavioral functions. Behavioral functions specify a set of operations and their temporal ordering for a specific functional behavior. Communication between different functions is described using different communication components, which include communication channels, shared variables and events. The CoFluent functional model describes the systems behavior and timing without platform constraints and can be used to simulate the system without a platform. This can be used to analyze shared resources without being distracted by problems related to mapping. Our approach makes use of the methodology in CoFluent Studio but refines it by hiding details from the designer and by providing tools for use case combination and flow control.

The Architecture Analysis and Design Language (AADL) [9] is used to model the software and hardware architecture of embedded real-time systems. It contains constructs for modeling both software and hardware components and is used for analysis such as schedulability and flow control. Compared to our approach, our models could be exported to AADL and be used for analysis instead of the simulator generated by CoFluent Studio.

Real Time Calculus (RTC) [10] is an interesting approach to investigate schedulability and resource usage of real-time systems. RTC could be used instead of simulation to analyse our models. Multi-mode RTC could be used to analyse flow control. RTC is used to get similar information about the system as we are interested in.

In general, the difference between our approach and other related approaches is that our model describes and focus on the use cases of the system as the main modelling concept. Our model is also designed to allow easy combination and evaluation of use cases and provides an automated method to do this. Another difference is that we are not searching for an optimal platform for the system but investigating how to enable new use cases on an existing platform.

3 Asset management

In order to make the design phase efficient we need support for managing reusable assets and methods for combining these with new components that make up the new system. The goal with asset management is to provide a better way to support the life-cycle management of product families and the strategic decision making by enabling

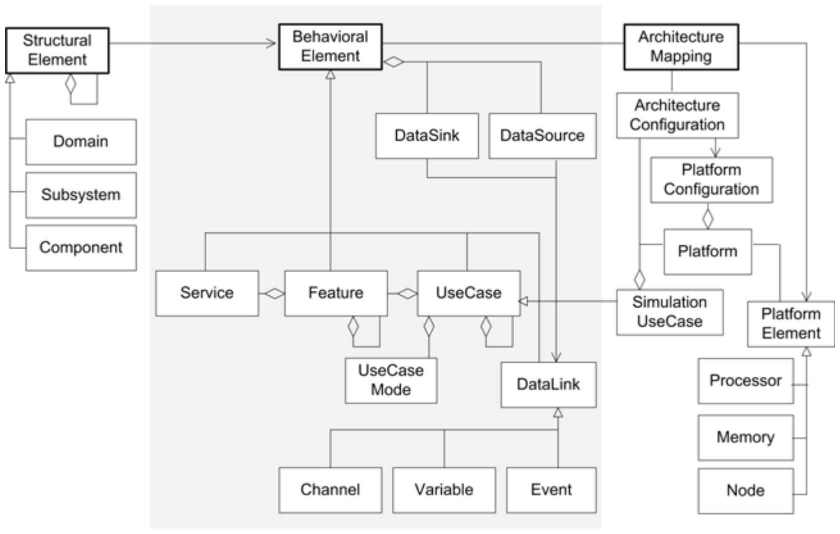


Fig. 2. The structural metamodel (left), the behavioral metamodel (middle) and the platform metamodel (right). The metamodel has been simplified in order to increase readability.

quick evaluation of new product features in order to give feedback for the business decisions.

We provide a set of structuring concepts encoded into a metamodel. The metamodel can be split into 3 parts. The **Structural Model** is used to delineate parts of the system according to particular responsibilities (e.g. the multimedia subsystem), the **Behavioral Model** is used to describe the functionality of the system and the **Platform Model** describes the hardware architecture of the system. These can also be seen as a hierarchy, where the structural model is at the highest level and the platform model at the bottom, but note that a structural relationship of inclusion does not necessarily imply a corresponding relationship on the platform level, since the same subsystem can be mapped onto different platform elements. In this paper we concentrate on describing the behavioral model as it is central to our modeling approach. More detailed information about the models can be found in [3, 4].

Although we use the term behavioral to characterize the part of the metamodel used to structure the functionality of the design, we do not propose a new approach to describe the functionality of atomic elements, but rely on the approach of the underlying simulation tool for this (e.g. the Timed-Functional approach of CoFluent studio is used in our tool).

The behavioral metamodel (c.f. Figure 2) supports a use case driven decomposition approach. The decomposition has as a starting point a use case. A use case describes a general functional scenario of a system, it looks at the system from the point of view of the end-user. Thus a use case is very generic like “video playback”. Typically this is too generic and the use case has to be refined into *use case modes*. A use case mode describes a specific way the use case will be implemented. For the video playback we could have 3 modes: video playback from internal memory, over 3G or over Wireless Lan. Typically a phone might support internal memory and 3G, and the Wireless streaming would be an option for a more high-end phone, depending on which *Features* the radio subsystem contains. The use case mode is decomposed into *Features* and *Services*. Services act as the elements at the lowest level of granularity and are used to compose features. A Feature can then be composed of other Features and Services.

It could be argued that this distinction is ad-hoc and driven by what is possible to do in the CoFluent Studio tool, but there is a however a more abstract characterization of the concepts by looking at the communication characteristics of the concepts.

– A use case defines the structure and the dynamics of the communication with external actors.
– A use case mode defines a particular instance of a use case, and fixes the communication network topology. A use case mode can contain detailed descriptions of the internal structure of the network.
– A feature describes dynamic communication aspects between nodes.
– A service defines the node level routing.

We still need 3 other concepts into our meta-model:

DataLinks represent functional communication elements. The types of communication elemenents we use exactly corresponds to the elements in CoFluent Studio. A *channel* is a communication channel which must have a specific type. A *variable* is a shared variable between Features or Services, and it must also have a specific type. An *event* is a trigger for a communication element and has no type definition.

Parameters are values that have to be specified in the use case, but usually are given concrete meaning on the level of feature or service. Typical example parameters would be the frame-size of a picture, frame-rate. Parameters are not separately represented as an entity in the metamodel, but are instead given as attributes.

Actors are used to create external inputs for the simulation model. Currently the tool implements very simple actors, like a file-reader, and random number generators. In the future we plan to include network simulators for TCP/IP and other protocols. A more detailed description of the actor mechanism can be found in [4].

4 Use-case based evaluation

Under the assumption that the design of all systems is structured according to the meta-model presented in the previous section, the design of a new system can now proceed as follows. The new system will consist of an old system structured as a set of use cases and a set of new uses cases. Then the first questions to be answered is whether the new use cases can be run on the given platform. This is standard fare. However the challenge comes when there is a need to support new use cases concurrently with the old ones. It is therefore important to be able to analyze use case combinations using different parameters rapidly. To this end we have developed a simple graph merging algorithm that given two use case modes (or features) creates a new model, that contains the behavior of both use case modes. The new model will contain shared elements and can

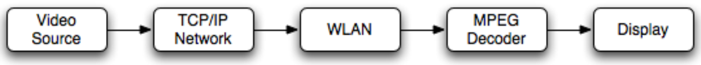


Fig. 3. Video playback use case

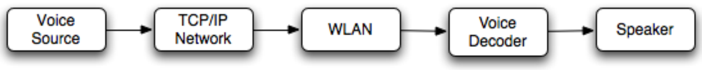


Fig. 4. Voice call use case

therefore be analyzed for problems in resource contention. The detailed description of the algorithm can be found in [3].

As a simple example we use two use cases: video playback and voice call. Figures 3 and 4 show the separate use cases using the notation in the EFCO tool and figure 5 shows the combined use case. The evaluation problem comes down to the sharing of the TCP/IP and WLAN features. This model can now be used to evaluate how the combined use cases can coexist on the given platforms.

As we combine *use cases* such that common functionality is merged there is a need to add mechanisms for routing messages through the system. This is done by adding a header to messages and adding routers to the design before the *simulation use cases* are generated and exported to CoFluent Studio. We need routers in two different situations, 1) for sending data through the choosen use case mode, e.g. transmit over WLAN or GSM and 2) for deciding to which use case a message belongs, e.g. if the message from the WLAN feature belongs to the video decoder or audio decoder use case. The routers analyse the headers and sends the messages in the correct direction without modifying the messages or adding delay. The routers abstract away the implementation of routing messages through the use case as the architect is only interested in analyzing the performance.

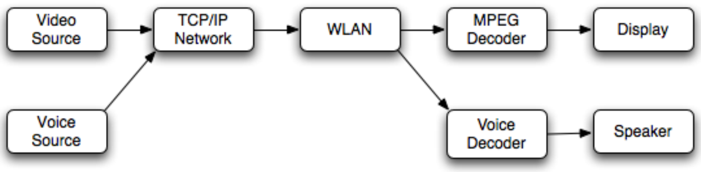


Fig. 5. Combined voice call and video playback use case

5 Flow control

A central observation that can be made is that in the presented approach the traditional distinction between a platform independent and a platform dependent model does not properly exist. Indeed the goal is that all the assets used in the models contain as much platform dependent information as possible. The reason why this is possible is because the hardware dependent values are going to remain stable throughout the life-time of a product family, or their changes can be predicted through discussions with the silicon vendors. The advantage of this is that a single use case can be verified very much on its own. The only real issue that is left and that needs the platform mapping is the resolution of resource contention. This is the topic of this section of the paper.

When several use cases are running concurrently there is often a need to control the resource usage of some critical parts of the system. Problems with resource sharing are typically a result from a task using a resource, such as a processor or a communication channel, for long time intervals or of high buffer levels which lead to long message delays. Buffer delays can be a problem when the buffer is shared between use cases, it is then possible that a critical message gets stuck in the back of the message queue. Such problems could be solved by giving some messages higher priority but this would be a static solution and it would not take the state of the system into account.

In the example use case in the previous section components such as the video decoder might need control mechanisms that prevent its buffers from overflowing or underflowing. This kind of control is needed as most systems contain components with stochastic behavior and therefore adds burstiness or jitter to the system. Examples of such components are communication networks, DMAs, storage systems etc. In the example use case the TCP/IP and WLAN components will shape the data flow and it is essential for the simulation results that the behavior of such components can be modeled and that the impact of these is considered in the simulation.

Furthermore, two use cases might also share a resource such as a processor due to the mapping. In this case flow control can be used to restrict the processor usage of one or both of the tasks, the desition of which tasks are allowed to use the processor can be made based on such properties as buffer level. As an example consider a task with bursty input, the task will alternate between periods of high activity and periods of being idle, this will in turn affect the execution of the other tasks running concurrently on the processor. The other tasks will experience periods when these get more or less processor time. Depending on the length of the periods and on the execution times of the tasks, some tasks might miss deadlines during the periods when the first task is active. This problem can be solved by restricting the processor usage of the first task, e.g. by suspending the task when its output buffer has reached a certain level, as a result the execution of the task will not follow the burstiness of the input stream anymore but instead the periods can be made shorter and the processor usage more even. In order to find the optimal parameters for the system the designer can try different flow control mechanisms.

In our approach the designer can add flow control constructs to the system in order to balance the resource contention. In its simplest form flow control is a sender and a receiver feature where the receiver monitors a buffer and sends feedback messages to the sender when the buffer has reached one of the defined levels. The sender then

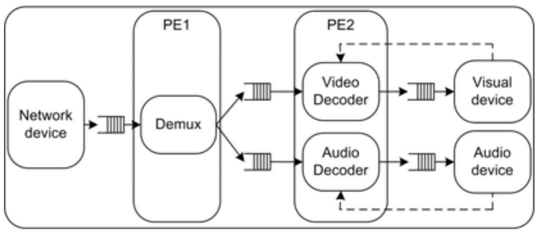


Fig. 6. An example of flow control

suspends or resumes its services depending on the content of the feedback message. Except for tuning the buffer levels, such constructs affect the resource contention as it can give the resource to a task that really needs it at the moment. In general the desision to add flow control to the design is based on the designers experience, therefore, the proposed approach does not consider how to get the optimal system but describes a methodology how to design a system and what kind of tools the designer needs. By using different types of flow control the designer can solve problems related to shared resources which in some cases means that he can avoid to make modifications to the platform.

In figure 6 a system running an audio and a video decoder is presented, the system has a general purpose processor (PE1) which in this case handles the network interface and feeds the streams to the decoders, the general purpose processor also runs the operating system. The actual decoding is performed on a digital signal processor (PE2). As the input streams are received over a network and because the sender serves several client simultaneously, the input streams are bursty. The resource sharing we need to simulate is 1) the sharing of the network and 2) sharing of the processor used for decoding the audio and video streams.

The goal of the system architect is to find the parameters that ensure that the playout buffers of the audio and video decoders does not underflow. If either decoder process is late, i.e. the level of the playout buffer is low, it should get more processor time than the other tasks. For this purpose the buffers are controlled using a flow control mechanism which changes execution rate of the tasks depending on the level of the playout buffer. In figure 6 the feedback channels are illustrated with dashed arrows. What can be ensured with this type of simulation is that if we add a specific flow control, the system will work as long as the input streams are within the limits we have specified regarding burstiness and jitter.

In this example the flow control is simple in that sense that it directly controlls the output buffer of the sender feature. Often this is not the case, in many applications control messages travel in the same channels as other messages and the messages might pass through several features, as an example consider a sender and receiver communicating through a network. This will affect such aspects as the delay of feedback messages and the delay before the impact can be measured after changing the state of a feature. If there are several buffers between a sender and a receiver, the level of the receivers input buffer might continue to rise for some time after the sender has been suspended. Such properties can be analyzed when simulating the system with different parameters and by monitoring buffer levels and message delays.

During the simulation stage buffer levels and message delays at different parameter settings can be measured for the use case combinations. The designer obtains useful information about the control mechanisms needed for specific use case combinations. It is essential that flow control parameters can be easily modified and that the structure of the flow control mechanisms are general enough that the design does not need to be changed when changing type of flow control. This enables the designer to try several setups rapidly and find a solution that satisfies the requirements.

In our metamodel flow control controlls the features by suspending and resuming the services in these. In the generated CoFluent project a controller is added to the output of a feature and the services of the feature is connected to the controller by suspend/resume control signals. A controller is a service that forwards data in zero time if the output channel is not full and it has no buffer space on its input. This is important as adding a controller should not add buffer space or delay to the system. Further, the controller has an input port for feedback messages and an output port for setting the state of the other services of the feature. The feedback messages are produced by an observer located in the receiver feature. The observer records buffer levels and number of messages received and sends feedback messages according to the flow control protocol choosen. The observer has similar features as the controller as it does not add delay of buffer space to the system. Feedback messages are handled as any other message and can in some cases travel on the same channels as the data, it is only at the sender or receiver *features* where the messages need to be routed to the corrects data source/sink. This simple structure of flow control allows different feedback based flow control types without changing the structure of the model.

The basic flow control types we have implemented in the EFCO-tool are Watermark based flow control, Xon/Xoff, Window based flow control and Credit based flow control. These basic flow control mechanisms are based on existing protocols used in computer communication.

Watermark based flow control The Watermark based flow control makes its decisions based on available buffer space at the receiver. The sender can decrease/increase its transmission rate depending on if the buffer has reached its low or high water marks. The transmit rate can be changed for example by changing the priority of the sending process.

The **Xon/Xoff** protocol is similar to watermark based flow control but is simpler as it either signals the sender to stop or continue sending jobs based on the input buffer size of the receiver. The controller in the sender feature receives the feedback message and suspends/resumes its services depending on the content of the message.

A different type of flow control is **Window based flow control**. In this type of flow control the senders is allowed to transmit a given number (window size) of messages before acknowledgments are needed. This means that it is the number of messages between the sender and the receiver that is controlled and not the number of elements in a specific buffer. Window flow control is widely used; one example of a refinement of it is the sliding window protocol used in TCP.

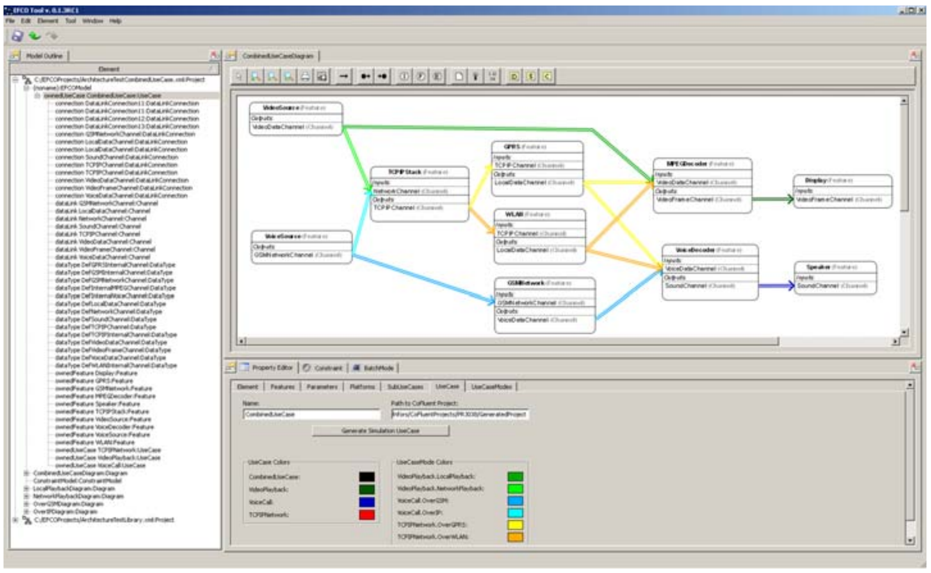


Fig. 7. Overview of the EFCO tool

A similar protocol is the **Credit based flow control**. Here the receiver gives credit to the sender which indicates the number of messages the sender is allowed to send, the sender consumes this credit while sending messages.

The flow control protocols described here are abstract simulation tools that gives the designer guidelines to how parts of the system needs to be controlled in order to make the whole system stable. The approach does not specify how the flow control should be implemented in a real system, instead this is left to the designer. In real systems flow control could be implemented within a single application be having the application suspend/resume itself depending on some criteria, in this case adding flow control would only make local changes to the application and not to the system. Another solution is to implement flow control support in the operating system, it would then be possible to have the scheduler make its decitions based the state of the use cases.

6 EFCO Tool

The EFCO tool is built on the Coral framework [1]. Figure 7 gives an overview of the tool, and shows the three most important parts of the tool. These are all part of the Coral modeler, and only support for the EFCO modeling language was needed to implement in them. The outline editor (to the left) shows the current loaded models in a tree-like hierarchy, both libraries and projects are loaded and shown in this editor. Elements in this tree-view can be selected, copied and pasted, and also dragged and dropped into other parts of the tool. The diagram editor (right top) shows a diagrammatic representation of different parts of the models loaded in the outline editor. Elements from one model in the outline editor can be dragged and dropped into the diagram editor of another model.

For example, a library element can be dragged from a library in the outline editor into a diagram of a project, which effectively combines the instantiated library element into the project. The property editor (right bottom) presents different editable properties of the currently selected element.

The EFCO tool implements the use case based metamodel and provides a set of tools for manipulating it. It supports import from and export to CoFluent Studio projects, this is important as services are the lowest level of detail handled in the EFCO tool and the implementation of these must be done in CoFluent Studio. The EFCO tool does not include any simulation tools and therefore the projects must be exported to CoFluent Studio, this because EFCO tool has only been implemented as a tool for reusing and combining components in an efficient way. What EFCO tool provides to the design is tools for managing use cases. It provides the nodes as a tree which enables easy reuse on any level of the design. It is possible to reuse whole use cases but also features and services. The tool also implements use case combination tools, the combining of components should be done in such a way that common smaller components are recognized and not duplicated, and mappings to architecture elements should also be reused if they have already been created. If a design already contain platform mapping the tool also keeps the mapping as it is usually only new use cases that should be mapped and the existing use cases will not be modified if not neccessary.

Before exporting the project to a CoFluent Project the use case is translated in to a *SimulationUseCase*. A *SimulationUseCase* contains everything needed to simulate different modes and combinations of the system, compared to a *UseCase* it breaks the structure suitable for reuse and implements components needed for simulating the system. To be able to simulate a *UseCase* created in EFCOTool it needs to be exported back to CoFluent Studio, before the exporting of the new *UseCase* can be done, it needs to be transformed into a *SimulationUseCase*. When transforming a *UseCase* into a *SimulationUseCase*, all routers needed for simulating the *UseCase* are automatically added to the model. Additional generic parameters are also added, one for every *SubUseCase* and one for choosing between different *UseCaseModes* in a *SubUseCase* or *UseCase*. Information regarding the architecture part of the model is also gathered to make the exporting mechanism easier.

There are currently two different types of routers, namely *UseCaseRouters* and *UseCaseModeRouters*. An *UseCaseRouter* is needed if, for example, a project element has more than one connection from the same output to project elements in different *SubUseCases*. The generic parameter that is automatically created for each *SubUseCase*, can be used to enable or disable *SubUseCases* when simulating the model. When disabling a *SubUseCase*, all data to that *SubUseCase* is routed to a discard channel in the *UseCaseRouter*. An *UseCaseModeRouter*, on the other hand, is needed if there are more than one connection from the same output in a project element to other project elements in the same *UseCase* or *SubUseCase*. Choosing the active *UseCaseMode* in a simulation is done via a generic parameter, which is automatically created when the *UseCaseModeRouter* is created.

The EFCOTool also supports the ability to automatically run several executable CoFluent Studio simulation models, with certain specific parameter values. This is called a *BatchMode* run, and it is located in the *BatchMode* tab in the *property edi-*

tor area of the EFCOTool. To be able to run a *BatchMode* in EFCOTool, a XML-file (that follows a XML-schema) needs to be created. This XML-file contains all information describing the different simulations, along with their parameter and simulation configurations. The batchmode tool is important for the design space exploration as the designer can analyse a large number of parameters and use case modes and combination without interacting with the tools. The designer can then after the batchmode run check the simulation results for feasible solutions.

7 Conclusions and Future Work

In this paper we have introduced an approach that uses a hierarchy of concepts that help structure the components/assets that are needed for composing the new product and map these to requirements on the business level. This approach uses the use case as the fundamental high-level design concept and structures the design so that it is easy to store design assets into libraries. This also enables use of the automated algorithm for creating new performance analysis models based on a set of use cases which allows for automatic combination of features and their evaluation of different platforms. Further, an approach for finding parameters to control the resource contention in the system by allowing features to have flow control constructs was also introduced. Such properties are important for exploring what kind of control a set of use cases need in order to work properly. Except from optimizing performance such construct can solve resource sharing problems and it might be possible to avoid to modify a platform.

Currently the approach is tied to the CoFluent Studio tool, but in principle any SystemC evaluation framework can be used, since the algorithms are all implemented on the level of the meta-model. As future work we will experiment with how real-time calculus [10] could be used to calculate bounds of the system. This is useful in case of realtime systems as the simulation can never show every possible special case; if the bound can be calculated it is possible to show that no task will miss a dealine. Simulation will still be useful for studying the performance of the system. Flow control can also be studied using RTC, one suitable method that can be used is multimode RTC [11]. It is also possible to directly describe the flow control mechanisms using RTC as long as we are only interested in the best case or the worst case, examples of this can be found in literature concerning network calculus [12] which has been used to analyze networks that are based on window buffer protocols.

References

1. Lundkvist, T., Porres, I.: Coordination of Model Transformation Engines and Visual Editors. In Peltonen, J., ed.: Proceedings of NW-MODE'09. (2009) 269–283
2. Cofluent design homepage, available at http://www.cofluentdesign.com (2009)
3. Fors, N.: Efficient combination of reusable components in embedded system design. Master's thesis, Åbo Akademi University, Faculty of Technology (2008) http://research.it.abo.fi/research/ese/projects/efco/fors.pdf.
4. Nylund, J.: Efcotool - a tool to efficiently combine and reuse components in embedded system design. Master's thesis, Åbo Akademi University, Faculty of Technology (2008) http://research.it.abo.fi/research/ese/projects/efco/nylund.pdf.

5. Živković, V.D., Lieverse, P.: An overview of methodologies and tools in the field of system-level design. In: Embedded processor design challenges: systems, architectures, modeling, and simulation-SAMOS, New York, NY, USA, Springer-Verlag New York, Inc. (2002) 74–88
6. Kienhuis, B., Deprettere, E.F., van der Wolf, P., Vissers, K.A.: A methodology to design programmable embedded systems - the y-chart approach. In: Embedded Processor Design Challenges: Systems, Architectures, Modeling, and Simulation - SAMOS, London, UK, Springer-Verlag (2002) 18–37
7. Sangiovanni-Vincentelli, A.: Defining platform-based design. EE Design (2002)
8. Calvez, J.P.: Embedded Real-Time Systems. A Specification and Design Methodology. John Wiley and Sons (1993)
9. Feiler, P.H., Gluch, D.P., Hudak, J.J.: The architecture analysis and design language (aadl): An introduction. Technical report, CMU/SEI (2006)
10. Chakraborty, S., Kunzli, S., Thiele, L.: A general framework for analysing system properties in platform-based embedded system designs. In: DATE '03: Proceedings of the conference on Design, Automation and Test in Europe, Washington, DC, USA, IEEE Computer Society (2003) 10190
11. Phan, L.T.X., Chakraborty, S., Thiagarajan, P.S.: A multi-mode real-time calculus. In: RTSS '08: Proceedings of the 2008 Real-Time Systems Symposium, Washington, DC, USA, IEEE Computer Society (2008) 59–69
12. Le Boudec, J.Y., Thiran, P.: Network calculus: a theory of deterministic queuing systems for the internet. Springer-Verlag New York, Inc., New York, NY, USA (2001)

Using Higher-order Transformations to Derive Variability Mechanism for Embedded Systems

Goetz Botterweck[1], Andreas Polzer[2], and Stefan Kowalewski[2]

[1] Lero, University of Limerick
Limerick, Ireland
goetz.botterweck@lero.ie
[2] Embedded Software Laboratory
RWTH Aachen University
Aachen, Germany
{polzer|kowalewski}@embedded.rwth-aachen.de

Abstract. One approach to handle the complexity of embedded systems is the use of models and domain-specific languages (DSLs) like Matlab / Simulink. If we want to apply such techniques to families of similar systems we have to describe their variability, i.e., commonalities and differences between the similar systems. Here, approaches from Software Product Lines (SPL) and variability modeling might be helpful. In this paper, we discuss three challenges which arise in this context: (1) We have to integrate mechanisms for describing variability into the DSL. (2) To efficiently derive products, we require techniques and tool-support that allow us to configure a particular product and resolve variability in the DSL. (3) When resolving variability, we have to take into account dependencies between elements, e.g., when removing Simulink blocks we have to remove the signals between these blocks as well. The approach presented here uses higher-order transformations (HOT), which derive the variability mechanisms (as a generated model transformation) from the meta-model of the DSL.

1 Introduction

Embedded systems are present in our everyday life. For instance, they are integrated into washing machines (to save energy and water), mobile devices (to simplify our lives) and in cars (to guarantee our safety).

In many cases, the engineering of embedded systems has to fulfill conflicting goals, such as reliability and safety on the one hand and the need for cost reductions and economic efficiency on the other hand. Moreover, the complexity of such systems is increasing due to the extension of functionality and increased communication with the environment. One possibility to deal with the complexity, requirements and the cost pressure is to use model-based development techniques like Matlab / Simulink. The advantages of such an approach are that connections between system components are expressed in an intuitive way on a higher abstraction level, which hides implementation details. Other benefits are support for simulation and increased reuse due to the component-oriented approach.

In the context of this paper we regard a "system" as a Matlab / Simulink model that contains components (Blocks) and provides a certain functionality. Nowadays, such systems are reused with small, but significant changes between different applications. Such variability causes additional complexity, which has to be handled. Some well-known techniques for this are suggested by Software Product Lines (SPL) [1,2]. In the research presented here, we discuss how these SPL techniques can be applied and adapted for the domain of model-based development of embedded systems.

The main contributions of this paper are (1) an analysis of Matlab / Simulink mechanism for variability, (2) concepts for managing realizing this variability with model transformations, (3) a mapping mechanism which adjusts the model according to configuration decisions (extension of [3]), and (4) concepts for “pruning”, i.e., the cleanup of components that are indirectly influenced by configuration decisions.

The remainder of the paper is structured as follows: First, we will give an overview of our approach (in Section 2). After this we will explain methods of modeling variability with Matlab / Simulink (Section 3) and how the suggested variability concepts are managed (Section 4). Subsequently, we explain the additional pruning methods (Section 5) and the implementation with model transformations (Section 6).

2 Overview of the Approach

We address the challenges described in the introduction with a model-driven approach that relies on higher-order transformations. Before we go into more details, we will give a first overview of our approach (see Figure 1). The approach is structured into two levels (1) *Domain Engineering* and *Application Engineering*, similar to other SPL frameworks [2,1]. Please note that we mark processes with numbers (❶ to ❼) and artefacts with uppercase letters ($\boxed{A}$ and $\boxed{C}$, $\boxed{B}$ will be used in later figures). In addition, we use indexes (e.g., $\boxed{A_d}$ and $\boxed{A_a}$) to distinguish artefacts on Domain Engineering and Application Engineering level.

2.1 Domain Engineering

Domain Engineering starts with the consideration of the context and requirements of the product line during *Feature Analysis* ❶ leading to the creation of a *Domain Feature Model* $\boxed{A_d}$, which defines the scope and available configuration options of the product line. Subsequently, in *Feature Implementation* ❷ a corresponding implementation is created. Initially, this implementation is given in the native Simulink format (`*.mdl` files). To access this native implementation in our model-based approach, we have to convert it into a model. For this we use techniques based on Xtext [4]. (see [5,6] for more details). As a result we get the corresponding *Domain Implementation Model* $\boxed{C_d}$.

To prepare the derivation of products a higher-order transformation (HOT) ❸ is executed, which reads the DSL meta-model and generates the derivation transformation ❻, which will later be used during *Application Engineering*.

2.2 Application Engineering

The first step in *Application Engineering* is *Product Configuration* ❹, where, based on the constraints given by the *Domain Feature Model* $\boxed{A_d}$, configuration decisions are

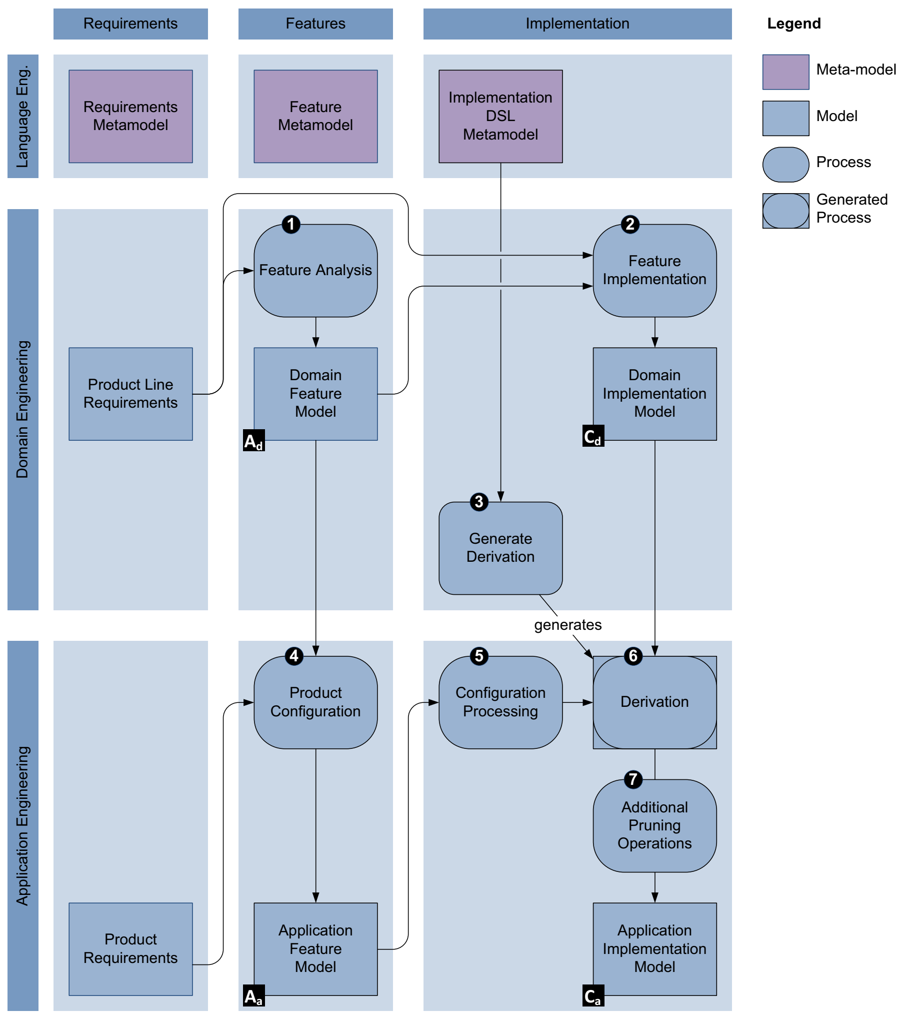


Fig. 1. Overview of the approach.

made, which defines the particular product in terms of selected or eliminated features. This results in a product-specific configuration which is saved in the *Application Feature Model* A_a. After some further processing ❺, this configuration is used in the *Product Derivation* ❻ transformation generated earlier by the HOT. This derivation reads the *Domain Implementation Model* C_d and – based on the product-specific configuration – derives a product-specific implementation. After additional pruning operations ❼, the result is saved as the *Application Implementation Model* C_a, which can be used in further processing steps (e.g., Simulink code generation) to create the final executable product.

We will now describe these processes in more detail. We start with the process of *Feature Implementation* (❷ in Section 3). We will then explain the required adaptation

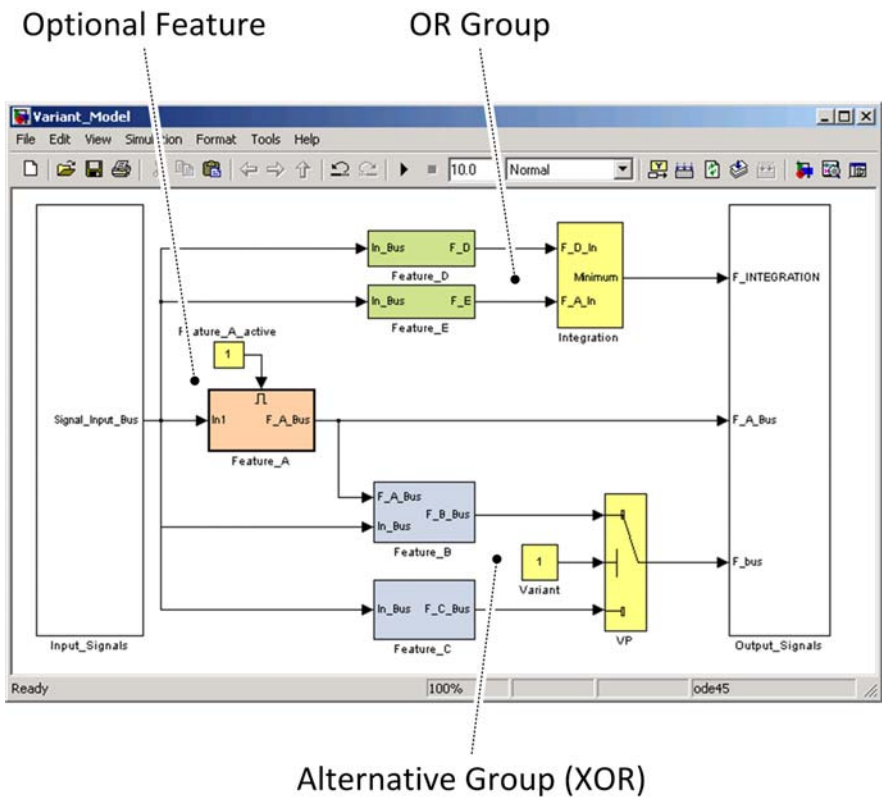


Fig. 2. Implementing Variability within a Matlab / Simulink model.

techniques including the *Derivation* ❻ (Section 4) and the subsequent *Pruning Operations* ❼ (Section 5).

3 Variability with Matlab / Simulink

Matlab / Simulink is a modeling and simulation tool provided by Mathworks. The tool provides *Blocks* which can be represented graphically and *Lines* which indicate communication. Blocks can contain other blocks, which allows to abstract functionality in special blocks called *Subsystems*. A similar technique is used by summarizing multiple lines into *Buses*.

In many cases, a feature can be implemented in Matlab / Simulink by a subsystem which contains the functionality for this particular feature. All necessary information consumed by the subsystem has to be provided by an interface, called *Input ports*. These input ports are normally buses. The information provided by a component are made available via *Output ports*, which are again organized buses and some additional signals.

We identified different possibilities to introduce variability in Matlab / Simulink for the required *Feature Implementation* ❷. Predominantly, we use the technique called *negative variability*. When applying this technique a model is created, which contains the *union* of all features across all possible configurations. When deriving a product-specific model this "union" model is modified such that features that were selected are activated and features that were not selected are deactivated or deleted.

To realize such a model in Simulink we can use two different approaches: (1) embedding the variability mechanisms *internally* (i.e., within the Simulink model) or (2) manipulating the model, based on variability information described *externally* (i.e., outside of the model).

We will now describe these two different techniques by explaining how common structures of feature models (Optional feature, Or Group, Alternative Group) can be implemented. For more information on these structures and feature models see [7,8]. An overview of feature diagram techniques and their formal semantics can be found in [9].

3.1 Variability mechanisms within the Simulink model

The first option is to realize variability by inserting artificial elements *into* the Simulink model. See the example model in Figure 2 where the blocks that implement features (*Feature_A*, *Feature_B*, *Feature_C*, ...) have been augmented with additional elements for variability (*Feature_A_active*, *Integration*, *Variant*, *VP*)

Mandatory features are – by definition – present in all variants. Hence, there is nothing to do for the implementation of mandatory features when deriving a product. There is no mandatory feature in the example.

Optional features can be realized in Matlab / Simulink models as a *triggered subsystem*, which is a special block that can be activated using a boolean signal (see *Feature A* in Figure 2). By using these mechanisms we are able to activate or deactivate a feature implementation.

When modelling *Alternative (XOR) group* and *Or group* we have to realize similar variability mechanisms. However, in addition we have to take care of the resulting signals and how they are fed into subsequent blocks. For alternative features we apply a Switch block (see the block *VP*in Figure 2) to choose the right output signal.

For *OR-related features* the integration of the results cannot be described for the general case, but has to be implemented depending on the particular case. In particular we have to take into the account the case when more than one feature of the group is selected and present in the implementation. We can implement an integration block for such cases in different ways. One example, is the case where limit is calculated for a certain dimension. Then the integration can be done by using a minimum (or maximum) function. In doing so, the lowest (highest) value is used by the system. As an example for this see *Feature D* and *Feature E* which are combined using an *Integration* block.

3.2 Variability mechanisms outside of the Simulink model

A second possibility, besides variability within the model, is the direct manipulation of the model with an *external* tool. To this end, during product derivation it is necessary to analyze the structure of a model file and delete the blocks representing deactivated features in such a way that a desired configuration is obtained.

When creating the union model it might be necessary to create a model, which could not be executed as-is within Simulink since the execution semantics of the created structure is undefined. This is because the variability decisions are still to be made and we have not yet removed certain elements.

As an example consider the case when we want to combine signals of two alternative features in an XOR group. In the union model, we might create the blocks for these two features side-by-side and feed their results into the same inport of the next block. As long as we configure this model (i.e., when the variability is applied), some parts will be removed, such that we get a valid and executable Simulink model. However, if we leave the union model (i.e., no variability applied) we do not realize the exclusion of the two features ("the X in XOR"). This leads to two signals feeding into on inport, which is an invalid Simulink model with undefined execution semantics.

In the example in figure Figure 2 this would correspond to connecting the *F_B_Bus* outport of Block *Feature_B* directly to the port *F_bus* of the bus *Output_Signals*, while at the same time connecting the *F_C_Bus* outport of Block *Feature_C* directly to the same port *F_bus*. If we would try to execute such a model, Simulink would respond with an error message.

The advantage of this kind of external method is that we do not have to pollute the domain model with artificial variability mechanisms.

Analyzing both possibilities, we came to the conclusion that a combination of both is an appropriate way of introducing variability. There are two major requirements which have to be fulfilled introducing variability methods. On the one hand we have to keep the characteristics of model based development (e.g., easy testing and simulation, capturing of dependencies possible), on the other hand the derived product should no longer contain any variability mechanisms. The mechanisms, which we introduced to realize this are explained in the next section.

4 Managing Variability

In this section we introduce the variability mechanisms we used in Simulink models and how they are influenced by configuring a corresponding feature tree. For instance the feature tree shown in Figure 3. This feature tree has an optional `Feature A`, XOR-grouped `Feature B` and `Feature C`, and OR-grouped `Feature D` and `Feature E`. Additionally the `requires` relation indicates that `Feature B` needs `Feature A`, i.e., whenever B is selected A has to be selected as well. This structure defines a set of legal configurations. Each of these configurations contains a list of selected features.

The mechanism that implements the structure of the feature tree has to fulfill certain requirements. In particular, it is important to keep the ability of simulating and testing the model. Therefore, it is necessary to build a model which has correct syntax, even after the variability mechanisms have been introduced. Additionally, the developer must have the possibility to introduce new features directly in the model. But due to the fact

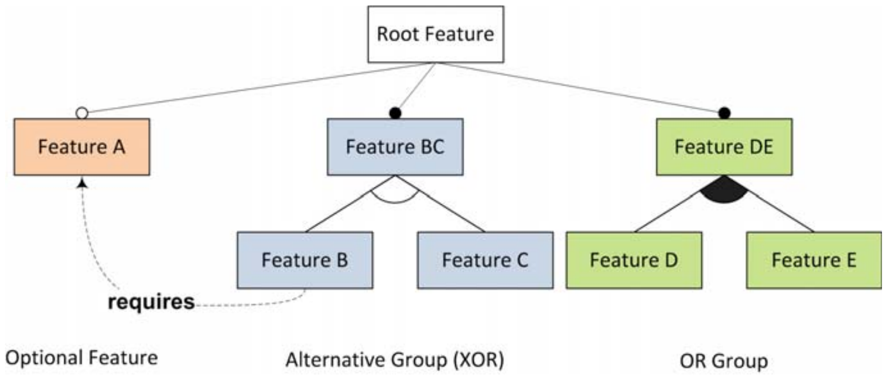


Fig. 3. Feature Tree for the variant Matlab / Simulink model shown in Figure 2.

that there are space and performance requirements the mechanisms have to be removed if the model is converted into a program executable on an embedded hardware.

To this end we used in general the approach of modeling variability *within* Simulink but with blocks which are specially marked as variability blocks. These special blocks are adopted afterwards according to the desired configuration. This means for obtaining a configuration, features which are not necessary will be deleted. Additionally signals between blocks are rerouted to be able to eliminate the variability blocks, which are not necessary in the derived product. The exact methods for a *switch*-block, *triggered subsystem* and, arbitrary integration mechanism is given in the following paragraphs.

The *switch*-block is used to express the relation between alternative grouped features. Therefore only one of them will give their contribution to the system. Using this, the developer of the implementation is able to simulate all features simply by choosing a configuration of the switch which selects to corresponding feature implementation block.

When deriving the executable product, it is necessary to delete those features that are not selected in the feature configuration. The output of each selected feature has to be connected with the port the switch block points to. All other corresponding signals have to be removed. In the end, no variability mechanism should be left in the model.

The situation is a bit simpler for triggered subsystems, which implement optional features. During simulation these blocks can be selected easily using *trigger*-signals. This will take effect on the simulation. If the corresponding feature is selected then the *trigger*-port has to be activated. When deriving an executable program, whenever the corresponding feature is deactivated, the subsystem has to be deleted. If it is activated nothing has to be done.

The mechanism to join the signals of OR-grouped features are not so easy to adopt. In the case of simulating the system the developer has to activate the desired features. This can be done either by using triggered subsystems to implement the features or just by disconnecting their signals with the block which joins the signals.

In case of deriving a real product two cases have to be distinguished. If only one feature is selected the other feature and the block joining the signals can be deleted. If more than one feature is selected the feature which are not selected can be deleted. But in this case the integration mechanism is still necessary.

5 Pruning

Dealing with blocks implementing features and the mechanisms to express variability is not the only adaptation which has to be done. There are additional components in the Simulink model which have to be changed as well, depending on the choices in the configuration

Especially, the interfaces for input or output signals which provide the information and supply the result from and to other systems have to be considered here. These blocks are interrelated with the configuration. They are necessary to define access to the signals. Therefore all signals provided from other systems are listed and treated in a way that the information is available in a common way. For instance the information for an input port is stored, renamed and mapped to internal representations which realizes the concrete representation.

Most of the signals, which are provided in the interface blocks are only needed for one feature. If this feature is not present the corresponding signals can be cleared out to optimize the implementation.

Buses that contains more than one signal have to be adopted in a similar fashion, i.e., only the required signals should remain within the buses. Hence, we have to prune out signals which are no longer needed because the corresponding features are to be removed.

These adoptions are not only necessary on the highest level of the model but also for subsystems. In general, it is possible that features are not visible at the highest level, for instance when subsystems are used to implement features. Since these subsystems use the same techniques as the main model of the whole system, it is necessary to adopt their interfaces and buses recursively in a similar fashion.

6 Implementation

The implementation discussed here (see Figure 4) is a technical realization of the approach shown earlier (see Figure 1).

The technical implementation follows the same structure, with Domain Engineering (Processes ❶ to ❸) and Application Engineering (processes ❹ to ❼). We will now discuss the model transformations in more detail.

6.1 Generating the derivation transformation

The higher-order transformation (HOT) `Metamodel2Derivation.atl` ❸ reads the meta-model of the DSL and generates a model transformation ❻, which is able to copy instances of this DSL.

Some excerpts of `Metamodel2Derivation.atl` are shown in Listing 1. The transformation is generated as an ATL `query` which concatenates the output as one

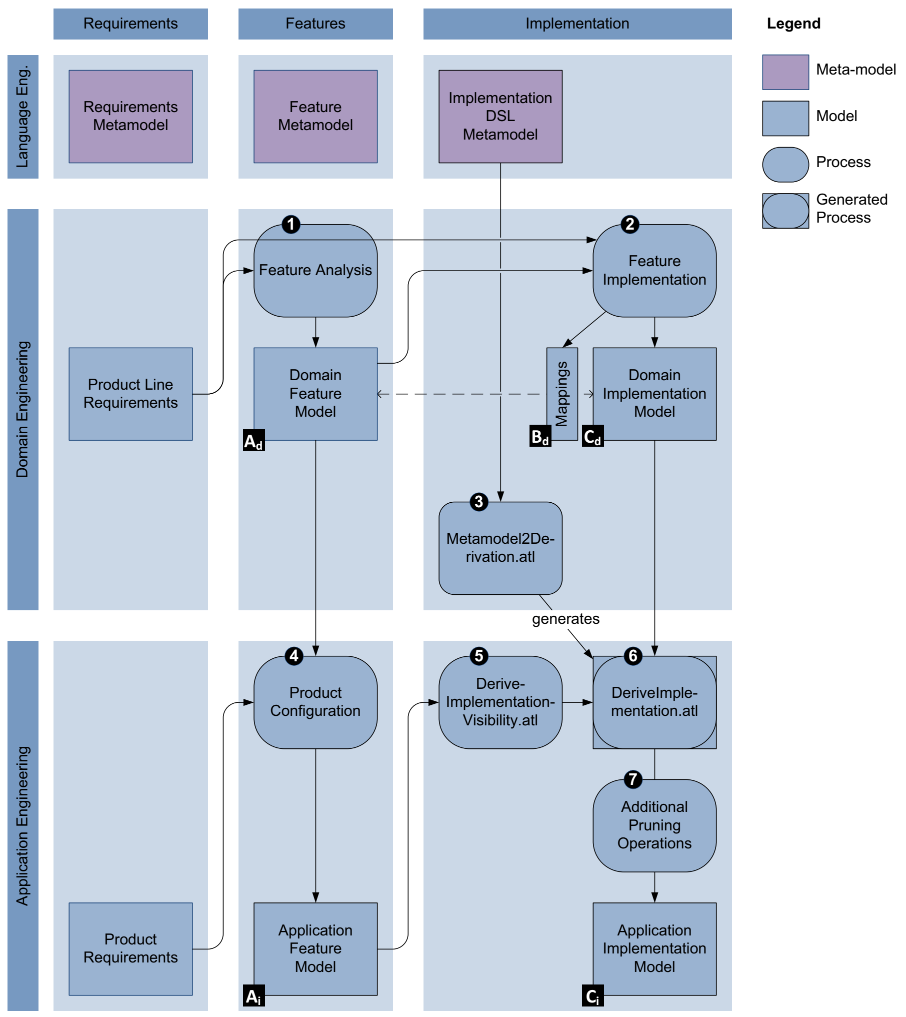


Fig. 4. Technical model workflow.

large string. For instance, the helper function `generateCopyRules()` (see Listing 1, lines 4–8) generates copy rules for all meta-classes in the meta-model. Details of each copy rule (e.g., the `rule`, `from` and to `part`) are generated by the function `Class.toRuleString()` (see Listing 1, lines 10–21) and other functions which were omitted for space reasons.

Examples of how the resulting transformation looks like, will be discussed in the next section.

```
query Metamodel2Derivation =
[..]

helper def: generateCopyRules() : String =
  ECORE!EClass
    ->allInstancesFrom('IN')
    ->sortedBy(o|o.cname())
    ->iterate(e; acc : String = '' | acc + e.toRuleString());

helper context ECORE!EClass def : toRuleString() : String =
  if not self."abstract" and self->inclusionCondition() then
    'rule ' + self->cname() + ' {\n' +
    '  from s : SOURCEMETA!' + self->cname() +
            self->inputConstraint() + '\n' +
    '  to t : TARGETMETA!' + self->cname() +
            ' mapsTo s (' +
            self->contentsToString() + ')\n' +
    '}\n\n'
  else
    ''
  endif;
[..]
```

Listing 1. Metamodel2Derivation.atl, transformation (3), excerpt.

6.2 Executing the derivation transformation

The generated derivation transformation `DeriveImplementation.atl` realizes a principle called "negative variability" [10] (also known as a "150% model"). With negative variability the domain model, here the *Domain Implementation Model* contains the *union* of all potential product-specific models.

Hence, the derivation transformation has to *selectively* copy the elements, filtering out those that should not be included, copying only those which will become part of the product-specific model, here the *Application Implementation Model* .

This selective copying is realized using the following mechanisms:

– For each meta-class in the DSL there is one copy rule. For instance, the rule `Block` (see Listing 2, lines 9–19) will copy instances of the meta-class `Block`.
– Each copy rule contains a condition that refers to an `.isVisible()` helper function, which controls whether an element is "visible" for the particular product and, hence, is copied or not. For instance, when processing the source element `s` the rule `Block` checks whether `s.isVisible()` (see Listing 2, line 12).
– To avoid inconsistent references, the following check is performed: Whenever references are processed, it is checked if the referenced elements are visible, as well. For instance, when copying references to `.generalization`, `.ownedAttribute`, and `.ownedOperation` the visibility is checked (see Listing 2, lines 16–18).

```
 1  module DeriveImplementation;
 2
 3  create TARGET : TARGETMETA from SOURCE : SOURCEMETA, CONFIG :
        CONFIGMETA;
 4
 5  helper context SOURCEMETA!Block def : isVisible() : Boolean =
 6    true;
 7  [..]
 8
 9  rule Block {
10      from s : SOURCEMETA!Block (
11          thisModule.inElements->includes(s) and
12          s.isVisible()
13      )
14      to t : TARGETMETA!Block mapsTo s (
15          name <- s.name,
16          subsystemblockbody <- s.subsystemblockbody->
                select(o|o.isVisible()),
17          normalblockbody <- s.normalblockbody->
                select(o|o.isVisible()),
18          scopeblockbody <- s.scopeblockbody->
                select(o|o.isVisible())
19    )
20  }
21  [..]
```

Listing 2. DeriveImplementation.atl, transformation (6), excerpt.

- The visibility functions determine whether instances of a certain meta-class will be copied. For instance, `Block.isVisible()` (see Listing 2, lines 5–6) calculates this for each instance of `Block`. In the initial version of `DeriveImplementation.atl` which is automatically generated from the meta-model all visibility functions default to true.
- In a second transformation, `DeriveImplementationVisibility.atl` ❺ Listing 3 these visibility functions are manually redefined. These functions access the product configuration and determine, which elements go into the product and which do not. Later on, these will be overloaded over the default visibility functions, by using ATL's superimpose mechanisms.

The selective copying is controlled by the function `s.isVisible()` defined in `DeriveImplementationVisibility.atl` ❺. This function reads the `Application Feature Model` $\boxed{A_a}$ and decides how this influences the filtering of elements in the `Domain Implementation Model`.

For this decision to be made, it is necessary to know how the various features in the feature model ($\boxed{A_d}$ and $\boxed{A_a}$) are related to the corresponding elements in the Matlab / Simulink implementation model.

```
module DeriveArchitectureDetermineVisibility;

create TARGET : TARGETMETA from SOURCE : SOURCEMETA, CONFIG :
    CONFIGMETA;

-- true if Block is referenced by a selected feature
helper context SOURCEMETA!Block def : isSelected() : Boolean =
[..]

-- true if Block is referenced by an eliminated feature
helper context SOURCEMETA!Block def : isDeselected() : Boolean =
[..]

helper context SOURCEMETA!Block def : isVisible() : Boolean =
  if self.isSelected() then
    if self.isDeselected() then
      true.debug('feature conflict for block' + self.name)
    else
      true
    endif
  else
    if self.isDeselected() then
      false
    else
      true -- default to visible
    endif
  endif;
[..]
```

Listing 3. DeriveImplementationVisibility.atl, transformation (5), excerpt.

This is represented by the *Mapping* B_d between the *Domain Feature Model* A_d and the *Domain Implementation Model* C_d implemented as a model which contains as elements dependencies. These dependencies relate features and the corresponding implementation, with a link mechanisms available from the meta-model. Using this meta-model we are able to map a feature given in the configuration model to a block given in the implementation model.

To implement pruning operations we are currently experimenting with and implementing new user-defined methods. These methods will adopt the copy rules in such a way that the methods given in Section 5 are realized. These pruning operations are influenced by the configuration and the mapping of features. However, they affect model components which are only *indirectly* referenced by the mapping model.

7 Related Work

Several projects deal with Product Derivation. The ConIPF project provides a methodology for product derivation [11]. ConIPF concentrates on the formalization of derivation knowledge into a configuration model. Deelstra et al. provide a conceptual framework for product derivation [12].

When dealing with variability in domain-specific languages a typical challenge is the mapping of features to their implementations. Here, Czarnecki and Antkiewicz [13] used a template-based approach where visibility conditions for model elements are described in OCL. In earlier work [14,15], we used mapping models and model transformations in ATL [16] to implement similar mappings. Heidenreich et al. [17] present FeatureMapper, a tool-supported approach which can map features to arbitrary EMF-based models [18].

Voelter and Groher [10] used aspect-oriented and model-driven techniques to implement product lines. Their approach is based on variability mechanisms in openArchitectureWare [19] (e.g., XVar and XWeave) and demonstrated with a sample SPL of home automation applications.

In earlier work [20,5], the authors have experimented with other mechanisms for negative variability (pure::variants Simulink connector [21] and openArchitectureWare's Xvar mechanism [19]) to realize variability in Embedded Systems. The mechanisms were applied to microcontroller-based control systems and evaluated with a product line based on the Vemac Rapid Control Prototyping (RCP) system.

The approach presented here can be seen as an integration and extension of work from Weiland [22] and Kubica [23]. Both presented mechanisms to adopt Matlab / Simulink-models based on feature trees. Weiland implemented a tool which influences certain variability points in a Simulink model. However, variability mechanisms are not removed during variability resolution. The approach given by Kubica constructs a new Simulink model for a derived product.

Tisi et al. provide an literature review on higher-order transformations [24] including a classification of different types of HOT. Oldevik and Haugen [25] use higher-order transformations to implement variability in product lines. Wagelaar [26] reports on composition techniques for model transformations.

8 Conclusion

In this paper we presented an approach to introduce and adopt variability in a model-based domain specific language (Matlab / Simulink) for developing embedded systems.

With our approach we are able to simulate and test variable Simulink-models by introducing mechanisms to manage variability and additionally derive models which contains only the product specific components. This provides us with memory and computation time efficient models.

All model transformations were implemented in the ATLAS Transformation Language (ATL) [16]. The original version was developed with ATL 2.0. We are currently experimenting with ATL 3.0 and its improved support for higher-order transformations.

Using this technique we are able to reuse previous work which implements the transformation from a domain specific language and a abstract variability mechanism. This approach is expanded by new methods to decided whether a feature is active and new domain specific methods are needed to adopt the implementation model.

9 Acknowledgements

This work was supported, in part, by Science Foundation Ireland grant 03/CE2/I303_1 to Lero – the Irish Software Engineering Research Centre, http://www.lero.ie/. The higher-order transformation (❸ in the overview) was inspired by an ATL case study by Dennis Wagelaar.

References

1. Clements, P., Northrop, L.M.: Software Product Lines: Practices and Patterns. The SEI series in software engineering. Addison-Wesley, Boston, MA, USA (2002)
2. Pohl, K., Boeckle, G., van der Linden, F.: Software Product Line Engineering : Foundations, Principles, and Techniques. Springer, New York, NY (2005)
3. Beuche, D., Weiland, J.: Managing flexibility: Modeling binding-times in simulink. [27] 289–300
4. Eclipse-Foundation: Xtext http://www.eclipse.org/Xtext/.
5. Polzer, A., Botterweck, G., Wangerin, I., Kowalewski, S.: Variabilitt im modellbasierten engineering von eingebetteten systemen. In: 7. Workshop Automotive Software Engineering, collocated with Informatik 2009, Luebeck, Germany (September 2009)
6. Botterweck, G., Polzer, A., Kowalewski, S.: Interactive configuration of embedded systems product lines. In: International Workshop on Model-driven Approaches in Product Line Engineering (MAPLE 2009), colocated with the 12th International Software Product Line Conference (SPLC 2008). (2009)
7. Kang, K., Cohen, S., Hess, J., Novak, W., Peterson, S.: Feature oriented domain analysis (FODA) feasibility study. SEI Technical Report CMU/SEI-90-TR-21, ADA 235785, Software Engineering Institute (1990)
8. Czarnecki, K., Eisenecker, U.W.: Generative Programming. Addison Wesley, Reading, MA, USA (2000)
9. Schobbens, P.Y., Heymans, P., Trigaux, J.C.: Feature diagrams: A survey and a formal semantics. In: Requirements Engineering Conference, 2006. RE 2006. 14th IEEE International. (2006) 136–145
10. Voelter, M., Groher, I.: Product line implementation using aspect-oriented and model-driven software development. In: 11th International Software Product Line Conference (SPLC 2007), Kyoto, Japan (September 2007)
11. Hotz, L., Wolter, K., Krebs, T., Nijhuis, J., Deelstra, S., Sinnema, M., MacGregor, J.: Configuration in Industrial Product Families - The ConIPF Methodology. IOS Press (2006)
12. Deelstra, S., Sinnema, M., Bosch, J.: Product derivation in software product families: a case study. The Journal of Systems and Software **74** (2005) 173–194
13. Czarnecki, K., Antkiewicz, M.: Mapping features to models: A template approach based on superimposed variants. In: GPCE'05, Tallinn, Estonia (September 29 - October 1 2005)
14. Botterweck, G., Lee, K., Thiel, S.: Automating product derivation in software product line engineering. In: Proceedings of Software Engineering 2009 (SE09), Kaiserslautern, Germany (March 2009)

15. Botterweck, G., O'Brien, L., Thiel, S.: Model-driven derivation of product architectures. In: Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering (ASE 2007), Atlanta, GA, USA (2007) 469–472
16. Eclipse-Foundation: ATL (ATLAS Transformation Language) http://www.eclipse.org/m2m/atl/.
17. Heidenreich, F., Kopcsek, J., Wende, C.: Featuremapper: Mapping features to models. In: ICSE Companion '08: Companion of the 13th international conference on Software engineering, New York, NY, USA, ACM (2008) 943–944
18. Eclipse-Foundation: EMF - Eclipse Modelling Framework http://www.eclipse.org/modeling/emf/.
19. openarchitectureware.org: Official open architecture ware homepage http://www.openarchitectureware.org/.
20. Polzer, A., Kowalewski, S., Botterweck, G.: Applying software product line techniques in model-based embedded systems engineering. In: 6th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2009), Workshop at the 31st International Conference on Software Engineering (ICSE 2009), Vancouver, Canada (May 2009)
21. Pure::systems: pure::variants Connector for Simulink http://www.mathworks.com/products/connections/product_main.html?prod_id=732.
22. Weiland, J., Richter, E.: Konfigurationsmanagement variantenreicher simulink-modelle. In: Informatik 2005 - Informatik LIVE!, Band 2, Koellen Druck+Verlag GmbH, Bonn (September 2005)
23. Kubica, S.: Variantenmanagement modellbasierter Funktionssoftware mit Software-Produktlinien. PhD thesis, Universität Erlangen-Nürnberg, Institut für Informatik (2007) Arbeitsberichte des Instituts für Informatik, Friedrich-Alexander-Universität Erlangen Nürnberg.
24. Tisi, M., Jouault, F., Fraternali, P., Ceri, S., Bézivin, J.: On the use of higher-order model transformations. [27] 18–33
25. Oldevik, J., Haugen, O.: Higher-order transformations for product lines. In: 11th International Software Product Line Conference (SPLC 2007), Washington, DC, USA, IEEE Computer Society (2007) 243–254
26. Wagelaar, D.: Composition techniques for rule-based model transformation languages. In Vallecillo, A., Gray, J., Pierantonio, A., eds.: ICMT. Volume 5063 of Lecture Notes in Computer Science., Springer (2008) 152–167
27. Paige, R.F., Hartman, A., Rensink, A., eds.: Model Driven Architecture - Foundations and Applications, 5th European Conference, ECMDA-FA 2009, Enschede, The Netherlands, June 23-26, 2009. Proceedings. In Paige, R.F., Hartman, A., Rensink, A., eds.: ECMDA-FA. Volume 5562 of Lecture Notes in Computer Science., Springer (2009)

Model-Based Extension of AUTOSAR for Architectural Online Reconfiguration

Basil Becker[1], Holger Giese[1], Stefan Neumann[1],
Martin Schenck[2] and Arian Treffer[2]

Hasso-Plattner-Institute at the University of Potsdam
Prof.-Dr.-Helmert-Str. 2-3
14482 Potsdam, Germany
[1]forename.surname@hpi.uni-potsdam.de
[2]forename.surname@student.hpi.uni-potsdam.de

Abstract. In the last years innovations in the automotive domain have more and more been realized by software leading to a dramatically increased complexity of such systems. Additionally automotive systems have to be flexible and robust, e.g., to be able to deal with failures of sensors, actuators or other constituents of an automotive system. One possibility to achieve robustness and flexibility in automotive systems is the usage of reconfiguration capabilities. However, adding such capabilities introduces even higher degree of complexity. To avoid this drawback we propose to integrate reconfiguration capabilities into AUTOSAR, an existing framework supporting the management of such complex system at the architectural level. Elaborated and expensive tools and toolchains assist during the development of automotive systems. Hence we present how our reconfiguration solution has been seamlessly integrated into such a toolchain.

1 Introduction

Today most innovations in the automotive domain are realized by software. This results in a dramatically increasing complexity of the developed software systems[1]. The objective of the AUTOSAR framework is to deal with this complexity at the architectural level. Additionally these systems need to deal with diverse situations concerning the context in which the software is operating. Such systems and especially the software, which is realizing essential functionalities of the overall system, need to be flexible to react on changes of its context. Regardless if such a system need to react on failures or on other contextual situations[2], flexibility and robustness plays an important role in today's automotive applications.

[1] The complexity concerning the size of the developed software, the functionality realized by the software system and so on.

[2] An example for such a situation, which is not related to a failure is in case the car is connected to diagnostic devices.

Reconfiguration is one possibility to facilitate the flexibility and robustness of such systems. There exist different possibilities to realize reconfiguration within automotive software. One is to realize reconfiguration mechanisms at the functional level. Because the AUTOSAR framework primarily provides mechanisms to deal with the complexity at the architectural level also the reconfiguration aspects should be available at the same level. Because deriving architectural information from the functional level could be difficult or even impossible we propose to specify reconfiguration aspects at the architectural level and to automatically derive the needed functionality based on the architectural information.

Further in a typical development scenario one has to deal with black-box components provided by third parties and elaborated information about the included functionality is not available, what also hampers the management of reconfiguration aspects at the functional level. Another possible solution is to introduce a new approach inherently facilitating reconfiguration aspects in the context of automotive systems. Today standard methods and tools already exist for supporting the development process of AUTOSAR. Because adapting existing tools or developing new once is very costly the propagation of such a new approach would be hardly suitable in practice. Summarizing we have identified the need for an development approach that is able to provide reconfiguration capabilities at the architectural level, can be seamlessly integrated into an exiting development solution and can also include third party components into the reconfigurable architecture. In this work we show how reconfiguration capabilities, which are currently not included in the existing AUTOSAR approach can be supported at the architectural level without degrading existing development solutions, tools or the standard itself. We further show how the needed functionality for realizing the reconfiguration logic can be automatically generated based on the architectural information describing the reconfiguration. The used application example for our evaluation is related to the field of fault tolerant systems and from our perspective such systems are one possible field to which reconfiguration like discussed in the remainder of this work can be applied.

The remainder of this paper is organized as follows. In Section 2 we discuss existing approaches supporting reconfiguration relevant for automotive systems and especially those approaches providing reconfiguration capabilities at the architectural level. In Section 3 we briefly introduce the existing toolchain, which builds the technological foundation for our investigation concerning the developed extension for on-line reconfiguration within the AUTOSAR framework. Subsequently in Section 4 we show how such a system is usually modeled with the given tools and how the additional reconfiguration aspects could be formulated based on the input/output of the existing toolchain. In Section 5 we show how these created additional reconfiguration aspects are automatically merged back into the original architecture and how the merged result fits into the existing tools without discarding or degrading parts of the original toolchain. Finally we give short discussion concerning the current results of our work in Section 6.

2 Related Work

In several different areas of computer science ideas have been presented, which are related to the approach we are going to present in this paper. In the field of software product lines and especially dynamic software product lines the topic of variable software has been addressed. The software architecture community has presented some work on the reconfigurability of robotic systems. Work, tailored to the automotive domain, has been done in the DySCAS project. We did some research on self-optimizing mechatronic systems.

In previous work we have presented a modeling technique called Mechatronic UML (mUML), which is suitable for the modeling of reconfigurable and self-optimizing mechatronic systems [1, 2]. However, the mUML approach differs from the one, which will be presented in this paper, in the fact that mUML uses an own code generation mechanisms and thus could hardly be integrated into existing development tool chains.

In the DySCAS[3] project dynamically self-configuring automotive systems have been studied [3, 4]. DySCAS does not provide a model based development approach, tailored to the specification of reconfiguration. Reconfiguration is specified with policy scripts, which are then evaluated by an engine at run-time (cf. [5]).

Software Product Line Engineering (SPLE) [6] aims at bringing the assembly line paradigm to software engineering. Typically a software product line is used to develop multiple variants of the same product. However, as the classical SPLE approach targets the design-time variability of software it is not comparable to the approach we are going to present in this paper. Recently a new research branch has emerged from SPLE called Dynamic Software Product Line Engineering [7]. In Dynamic Software Product Lines the decision, which variant to run, has moved from design- to run-time. Such an approach is presented in [8], where the authors describe a dynamic software product line, which is suitable for the reconfiguration of embedded systems. In contrast to our approach this one is restricted to the reconfiguration of pipe-and-filter architectures and the reconfiguration has to be given in a textual form.

In [9] a framework for the development of a reconfigurable robotic system has been presented. But the presented approach does in contrast to ours not support the model-driven development of reconfiguration. A policy-based reconfiguration mechanism is described in [10]. The authors present a powerful and expressive modeling notation for the specification of self-adaptive (i.e. reconfigurable) systems but their approach requires too much computational power and is thus only remotely applicable to embedded systems. In [11] an approach based on mode automata has been presented. However, mode automata only support switching between different behaviors internal to a component and do not cover architectural reconfiguration.

[3] http://www.dyscas.org

3 Existing Development Approach

For the development of embedded systems – especially in the automotive domain – several tools exist that provide capabilities for model-based development of such systems. Tools used by companies typically are mature, provide reliable and optimized code generation mechanisms and are as expensive as complex. Hence, any technique that claims being usable in the domain of embedded / automotive systems must be integrated into the existing toolchain. We will use this section to exemplary describe a toolchain, which might be used in the context of the AUTOSAR domain specific language.

3.1 AUTOSAR

The AUtomotive Open System ARchitecture (AUTOSAR) is a framework for the development of complex electronic automotive systems. AUTOSAR provides a layered software architecture consisting of the Application layer, the Runtime Environment and the Basic Software layer. Figure 1[4] shows the different layer of the architecture. The Basic Software layer provides services concerning HW access, communication and Operating System (OS) functionality (cf. [12]). The Basic Software provides several interfaces in a standardized form to allow the interaction between the Basic Software layer and the application layer routed through the Runtime Environment. The Runtime Environment handles the communication between different constituents of the application layer and between the application layer and the Basic Software layer (e.g., for accessing Hardware via the Basic Software, cf. [13]). The Application layer consists of Software Components, which can be hierarchically structured and composed to so called Compositions. Software Components and Compositions can have ports and these ports can be connected via Connectors (see [14] for more details). The real communication is realized through the Runtime Environment in case of local communication between Software Components (Compositions) on the same node (Electronic Control Unit) or through the Runtime Environment in combination with the Basic Software in case of communication between different nodes.

The main focus of AUTOSAR is the modeling of architectural aspects and of structural aspects. The behavior modeling (e.g., needed control functionality for reading sensor values and setting actuators) is not the main focus of the AUTOSAR framework. For modeling such behavior existing approaches and tools can be integrated into the development process of AUTOSAR. One commonly used tool for the model based development of behavior is MATLAB/Simulink (like described in Section 3.2). For executing such functionality AUTOSAR provides the concept of Runnables, which are added as a part of the internal behavior of a Software Component. Developed functionality could be mapped to Runnables and these Runnables are mapped to OS tasks. Additionally events

[4] Picture taken from http://www.autosar.org/gfx/media_pictures/AUTOSAR-components-and-inte.jpg.

can be used to decide inside an OS task if specific runnables are executed at run-time (e.g., runnables could be triggered by events if new data has been received via a port of the surrounding Software Component). For more details about the OS provided by the AUTOSAR framework see [15].

Once the modeling and configuration is done, in the current release version of AUTOSAR[5] changes at run-time concerning the structure of the application layer (e.g., restructuring connectors) are not facilitated by the framework.

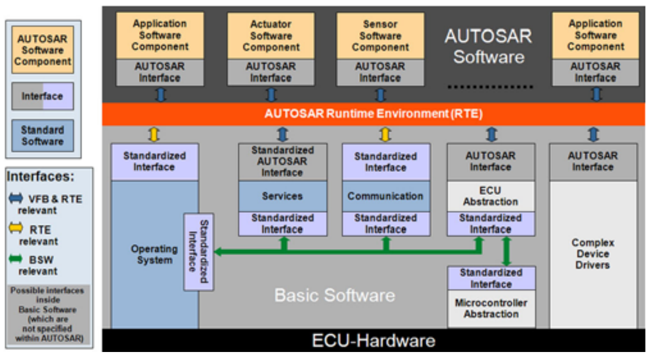


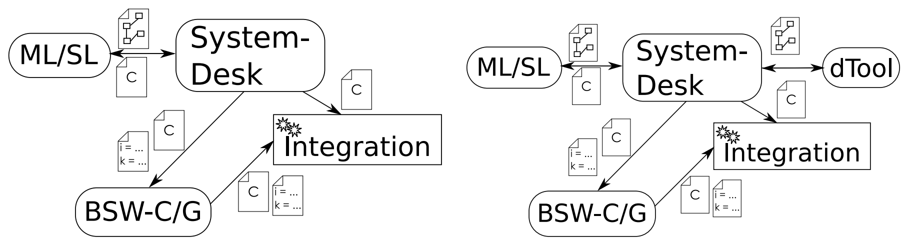
Fig. 1. The AUTOSAR layered architecture

3.2 Existing Toolchain

The scheme in Figure 2(a) shows one possible toolchain for the development of AUTOSAR systems. Rectangles with rounded corners represent programs, rectangles with cogwheels stand for processes. The arrows indicate exchange of documents, the type of the document (i.e. models, C-code or parameters) is annotated to the arrows. The system's architecture (i.e. components, ports and connectors) is modeled in SystemDesk[6]. Together with the architecture SystemDesk also supports the modeling of the system's deployment to several ECUs. The components behavior is specified using Matlab with the extension Simulink. For Matlab/Simulink (ML/SL) special AUTOSAR block sets exist, which allow the import of components specified in SystemDesk into Matlab and following the development of the component's functionality.

Further SystemDesk supports the generation of optimized C-Code, which conforms to the AUTOSAR standard concerning the Runtime Environment (cf. Subsection 3.1). Together with the C implementation of the software components modeled in SystemDesk the generated output also contains a configuration for the basic software layer. This layer is generated from specialized tools (e.g. Tresos

[5] Release 3.1
[6] http://www.dspace.de



(a) Exemplary toolchain for development with AUTOSAR (b) Tool chain for modeling reconfigurable AUTOSAR architectures

Fig. 2. The current and the extended toolchain for the development with AUTOSAR

by ElectroBit, abbreviated as BSW-C/G in Figure 2) and is specific to the system modeled in SystemDesk and the available hardware.

At the integration step a build environment compiles the generated C-Code and builds the software running on each ECU.

3.3 Evaluation Example

The used application example for showing the reconfiguration capabilities that are supplemented to the existing AUTOSAR framework in our approach is the reconfiguration of a set of adjacent aligned distance sensors. The discussed evaluation example allows reacting on sensor failures in the manner that the failure of individual sensor instances is compensated.[7]

Such adjacent aligned sensors are commonly used in a modern car, e.g., in case of a parking distance control. Such a parking distance control uses sensors (e.g., ultrasonic sensors) embedded in the front or rear bumper for measuring the distance to nearby obstacles.

Additionally in Section 5.3 we discuss the evaluation results of experiments we have made on an evaluation platform using the techniques described in Section 4.

4 Modeling Reconfiguration

In order to make an AUTOSAR system architecture reconfigurable, some additional concepts are needed. The toolchain needs to be extended in a certain way that extensions do not make the existing toolchain invalid. From our perspective the best way is to integrate an optional tool that can be plugged into the existing toolchain.

[7] For our application example we assume that a sensor failure can be observed at the level of Software Components.

4.1 Extended Toolchain

Our modeling approach is currently restricted to the modeling of AUTOSAR software architectures. The toolchain in Figure 2(b) shows our approach of extending the existing toolchain by another tool without degrading existing ones. By using this proposal the developer is free to choose, whether he wants to use our given enhancement or not. He can either model an architecture, that does not provide any reconfiguration or he can use our tool in addition and empower himself to specify and realize reconfiguration aspects. The advantages are obvious: better control and overview due to the diagrammatic depiction.

SystemDesk SystemDesk is a tool provided by *dSPACE*[8] supporting the modeling of AUTOSAR conform systems. Among other things it supports the modeling of the AUTOSAR HW and SW architectures. For modeling the SW architecture Software Components, compositions as well as ports, interfaces and connectors are provided as modeling artifacts. These artifacts can be used to describe the architectural aspects of a concrete SW architecture for a specific system like shown in Figure 3.[9] Besides modeling the architecture in SystemDesk, the tool also allows the linking of the Software Components to their behavior, written in C-Code or given in form of MATLAB/Simulink models.

Additionally the HW architecture including the used types of ECUs (Electronic Control Unit), the deployment of Software Components to these ECUs as well as additional information concerning the configuration (e.g., configuration concerning communication and the OS) can be specified. Based on this information SystemDesk automatically generates code, which can be compiled for the specified platform. Besides the code for the application layer SystemDesk also generates source code realizing the Runtime Environment functionality.

Figure 3 shows the relevant part of the SW architecture concerning our application example modeled in SystemDesk. Like depicted on the right side of Figure 3 the composition consists of four Software Components representing the distance sensors[10] connected to another composition *SensorLogic* evaluating the sensor values to a single value provided by the port *ShowDistanceOut*.[11]

The above mentioned elements (Software Components, ports and connectors) are used to describe the software when no reconfiguration is intended. Some additional elements shown in Figure 3 are described in more detail in the following section. These elements (*Interpolation*, *Reconfiguration* and the unused ports of the sensors) are used later to realize the reconfiguration functionality.

[8] www.dspace.de

[9] For the realization of control functionality other constituents can be imported into SystemDesk, e.g. in form of C-Code or Matlab/Simulink models, to realize the implementation of internal behavior of Software Components.

[10] The ports accessing the HW via the Runtime Environment and Basic Software are not shown here because they are not object of reconfiguration.

[11] To allow a better understanding *SensorLogic* calculates a single output value based on the different input values. Potentially also several output values can be computed.

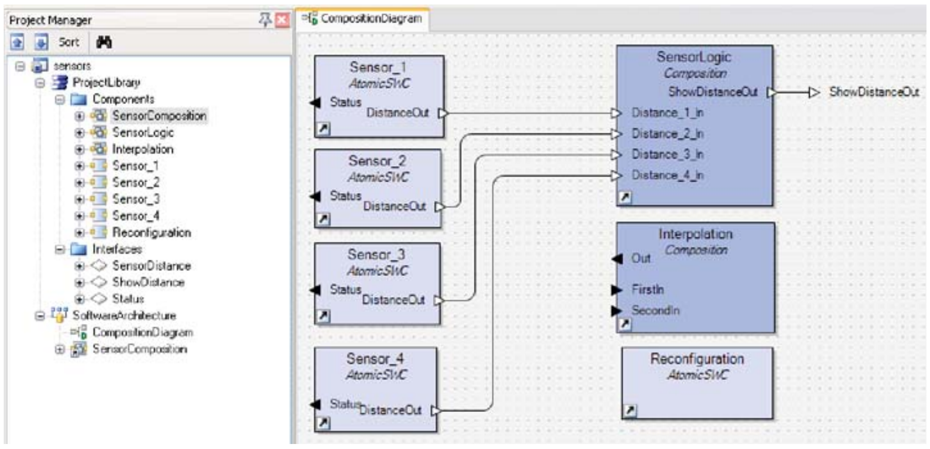


Fig. 3. Configuration in SystemDesk

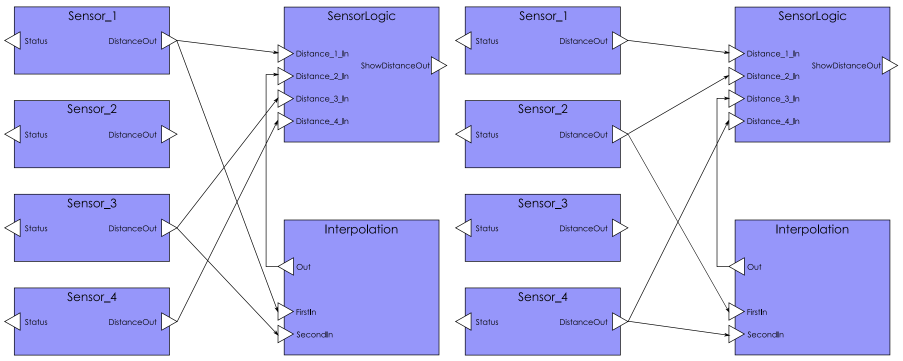
dTool The usual modeling procedure is not altered until the modeling in SystemDesk[12] is initially done like described above. After the model from SystemDesk is exported in form of an XML file[13] and loaded into the *dTool* the constituents concerning the reconfiguration could be specified. Using the dTool we are now able to model two different aspects, relevant for the reconfiguration. On the one hand our tool allows creating new configurations, which differ from the initial one. Such differences are alternative connections (in form of connectors) between components and/or compositions. Which parts of the architecture are relevant concerning reconfiguration is indicated by the Software Component *Reconfiguration* included in the original SystemDesk model. Alternatively the dTool allows to manually choosing relevant parts of the imported architecture. On the other hand our dTool allows to model an automaton, which specifies how to switch between the modeled configurations.

Figure 4(a) depicts the configuration (modeled in the dTool) associated with the state that sensor two is broken. In the shown configuration the value of the port *DistanceOut* from the broken sensor *Sensor_2* is not available. Consequently the value sent to the port *Distance_2_In* of the composition *SensorLogic* is interpolated from the to sensor values of the first and the third sensor via the additional composition *Interpolation*.

Figure 4(b) shows the configuration associated with the state that sensor four is broken and the value sent to the port *Distance_3_In* of the composition *SensorLogic* is interpolated based on the sensor values of the second and the fourth sensor.

[12] http://www.dspace.de/ww/en/ltd/home/products/sw/system_architecture _software/systemdesk.cfm

[13] The AUTOSAR framework specifies XML-Schemes for exchanging AUTOSAR models in a standardized form.



(a) Configuration in case Sensor_2 is broken (b) Configuration in case Sensor_3 is broken

Fig. 4. Two configurations of the architecture for two different scenarios

The composition *Interpolation* used here provides some functionality for interpolating two different sensor values. This functionality has been added specifically for our application example.[14] This interpolation functionality is used to approximate the value of a broken sensor based on the values of two adjacent sensors. It is potentially possible to integrate this functionality into an existing Software Component, but for a better understanding, we decided to introduce a new Software Component for this purpose.

The second part, which could be modeled in the dTool relevant for the reconfiguration is the automaton shown in Figure 5 specifying how to switch between different configurations. The automaton consist of the initial state *initial*, where all four sensors work correctly, the state *sensor2broke* where the second sensor is broken, the state *sensor3broke* where the third sensor is broken and state *allfail* where the first or the fourth sensor or more than one sensor is broken. Transitions between these states specify which reconfiguration is applied at runtime. The transitions are further augmented with guards. These guards are expressions over the values provided by components within the reconfigurable composition, which provide information relevant for the reconfiguration (in our case these information are provided via the *Status*-ports of the four Sensor-Software Components). An example for such a guard is shown at the transition from state *initial* to state *sensor2broke* requiring that the status port of the Software Component *Sensor_2* provides the value 0 (indicating a broken sensor).

For the application example we assume that such status ports of the Software Components representing the sensors exist as we otherwise were not able to observe each sensors' status.[15]

[14] In our application example this functionality has been realized using Matlab/Simulink.

[15] Alternatively an observer could be realized in form of an additional Software Component evaluating the sensor values over time and providing the status ports. If the

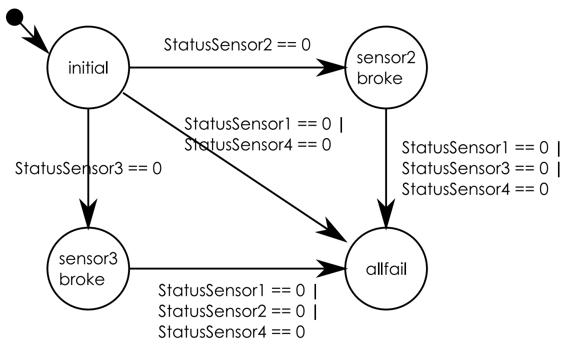


Fig. 5. Reconfiguration automaton in the dTool

5 Merge

In its current version the AUTOSAR standard does not support reconfiguration as a first class modeling element. Thus, SystemDesk also does not support modeling of diagrams that represent different variations of one composition. Hence the direct import of the reconfiguration, we have modeled in the *dTool*, is impossible. Nevertheless we want to make use of SystemDesk's elaborated and AUTOSAR standard conform code generation capabilities. We had to find a way to translate the reconfiguration behavior into a SystemDesk/AUTOSAR model. This is done by merging all configurations to one final model. In the final model, the reconfiguration logic will be encapsulated by two components, the RoutingComponent and the StateManager.

5.1 Merging configurations

Our modeling approach only allows the reconfiguration of connections between components but is not suitable for the addition and removal of components at run-time[16]. Hence, a merged configuration consists of all components, which have been modeled in SystemDesk at the early stages (cf. Subsection 4.1). Connections, which do not exist in all configurations, are redirected via a special component, called RoutingComponent. Therefore, the first step is to build the intersection of all configurations. Connections found here are directly inserted into the merged model. Next the RoutingComponent is added.

Generating the RoutingComponent The RoutingComponent intersects every connection, which is not invariant to the reconfigurable composition. Follow-

measured values of consecutive points in time repeatedly have improper values (too big differences) a malfunction can be deduced.

[16] Please note that the dTool allows to modeling configurations, which do not contain all components. The semantic is that the components are hidden, a dynamic loading of components is not supported by AUTOSAR.

ing the RoutingComponent has to know at each point in time, which configuration is currently the active one. Which configuration is active, is determined by the evaluation of the current configuration and the valuation of the variables used in the guards of the reconfiguration automaton (cf. Figure 5). As an evaluation of the automaton at each point in time a value is sent to the RoutingComponent, is much too expensive we have implemented a different strategy.

```
38 switch (configuration_0) {
39   // Routing for configuration initial:
40   case 0:
41     Rte_IWrite_Distance_2_In_0_1_sndr_Distance(distance_1);
42     break;
43   // Routing for configuration allfail:
44   case 3:
45     break;
46   // Routing for configuration sensor2broke:
47   case 1:
48     break;
49   // Routing for configuration sensor3broke:
50   case 2:
51     Rte_IWrite_Distance_2_In_0_1_sndr_Distance(distance_1);
52     Rte_IWrite_FirstIn_1_0_sndr_Distance(distance_1);
53     break;
54 }
```

Listing 1. Excerpt of the RoutingComponent's code

The configurations modeled in the dTool get a unique number each. The RoutingComponent receives the number of the currently active configuration via a special input port. Using this information the RoutingComponent can be implemented as a sequence of switch statements. The computation of the current active configuration is done in a second component – the StateManager. The dTool automatically generates a runnable for the RoutingComponent containing the described behavior. An excerpt of the RoutingComponent's implementation is shown in Listing 1. The variables configuration_0 and distance_1 hold the values of the current configuration and the second sensor's distance respectively. The excerpt is responsible for routing the value provided by the second distance sensor. In configuration allfail (cf. line 44) and sensor2broke (cf. line 47) no routing takes place. In the initial configuration the sensor's distance value is simply forwarded (cf. line 41) and in case the third distance sensor broke down, the value is forwarded as in initial (cf. line 51) but it is also sent to the Interpolation component (cf. line 52).

StateManager The StateManager – as briefly mentioned above – is responsible for the computation of the currently active configuration. Therefore, it has to be connected with all ports that provide values, which are used in the guards of reconfiguration automaton. Each time the StateManager receives an update on its ports, it has to evaluate the automaton again and change the value of the currently active configuration accordingly.

```
49 configuration_0 = Rte_IrvRead_configuration();
50 switch (configuration_0) {
51   // State change logic for configuration
52   //      initial
53   case 0:
54     // Transition to CGConfiguration#sensor2broke(id: 1,
          name: sensor2broke)
55     if (StatusSensor2_10 == 0) {
56       configuration_0 = 1;
57       Rte_IrvWrite_configuration(configuration_0);
58       Rte_IWrite_conf_out_configuration(configuration_0);
59     }
60   break;
```

Listing 2. Excerpt from the StateManager's implementation

Updates to the StateManager's ports are signaled by events, which then trigger the StateManager's evaluation function.[17] A small part of this evaluation function is shown in Listing 2. At line 49 of the listing the currently active configuration is read, which then is used as input for the switch statement in the following line. In case the second distance sensor is broken (identified by StatusSensor2_10 equals zero) the configuration is changed (cf. line 56). Then the changed configuration is written to the StateManager's internal configuration variable (cf. line 57) and provided to other components through the conf_out port (cf. line 58).[18]

5.2 Final SystemDesk project

Figure 6 shows the Sensor-Composition after exporting the merged model to SystemDesk again. The components for the distance sensors are all connected to the RoutingComponent, which is named *Reconf* in this diagram. The system modeled in our application example does not allow an interpolation for the sensor components one and four. Following these components are always directly connected with the SensorLogic component and are not handled by the RoutingComponent. Nevertheless they also have to be connected to the RoutingComponent as the sensor values are used to interpolate the second respective third sensor in case of a failure.

The StateManager is depicted below the RoutingComponent and is connected to the RoutingComponent through the *Conf* ports, which provide information about the currently active configuration. As defined in the reconfiguration automaton (cf. Figure 5) the decision which configuration to use, depends on the values of the sensor components' status ports. Following the StateManager is connected to those ports. As the reconfiguration automaton does not

[17] Event mechanisms in form Runtime Environment events provided by the AUTOSAR framework have been used to trigger the runnable realizing the functionality of the StateManager. More information about Runtime Environment events can be found in [13].

[18] E.g., provided to the RoutingComponent.

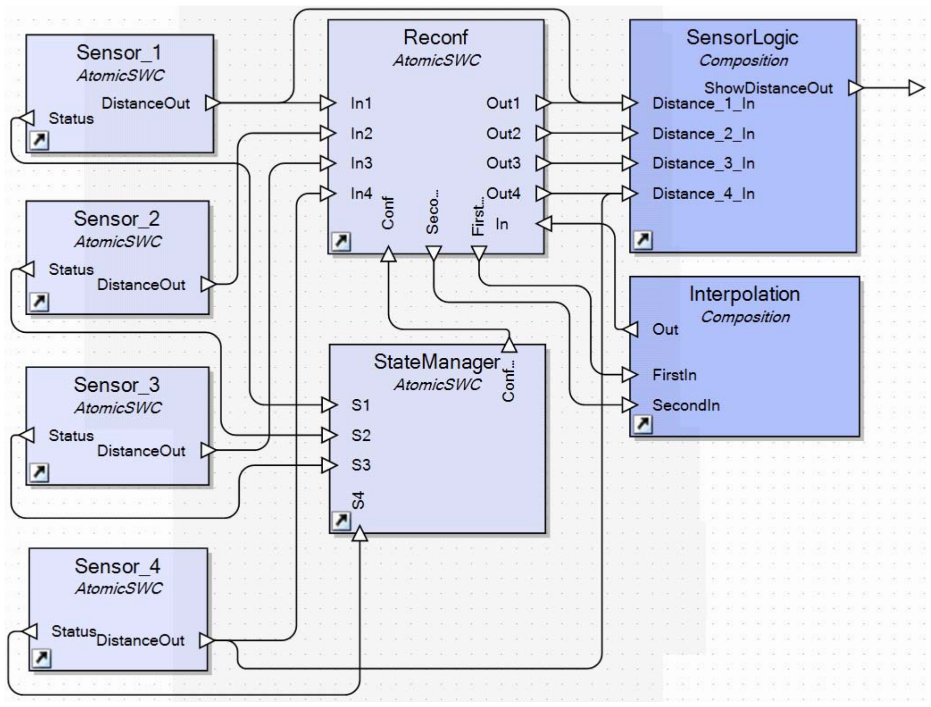


Fig. 6. Resulting merged SW Architecture in SystemDesk

rely on any values provided by the Interpolation or SensorLogic component the StateManager is not connected with them.

5.3 Evaluation Results

The above described approach for the modeling and realization of reconfiguration aspects has been evaluated within a project arranged at Hasso-Plattner-Institute in collaboration with the dSPACE GmbH.

As an evaluation platform for the shown approach the Robotino robot[19] has been used, which provides an open platform for running C/C++ programs (among others) on a Real-Time Operating System (RTOS). The RTOS is provided in form of RTAI[20], which is a real-time extension for the Linux operating system. To be able to evaluate the developed concepts on this platform an execution environment has been realized based on the existing RTAI Linux, which allows to compile and execute the outcome of the above described extended toolchain including the resulting parts of the reconfiguration functionality.

The robot provides nine distance sensors uniformly distributed around its chassis. In the context of our evaluation experiments we modeled the reconfig-

[19] http://www.festo-didactic.com/int-en/news/learning-with-robots.htm
[20] For more details see https://www.rtai.org.

uration of distance sensors accordingly to the above used evaluation example using nine instead of four sensors.[21]

The generated source code of the different tools has been compiled and executed on the platform to show the applicability of our approach. In addition we analyzed the overhead resulting from the reconfiguration functionality added by our approach in comparison to the original functionality without any reconfiguration capabilities. For this purpose we measured the execution time of the generated reconfiguration automaton included in the added StateManager in combination with the parts resulting from the routing functionality realized in the additional RoutingComponent (both components are shown in Figure 6).

In case of the nine sensors provided by the robot we measured execution times of the relevant parts concerning the reconfiguration functionality between 20 and 100 microseconds depending on the type of reconfiguration (react on the defect of one or several sensors at the same point in time). The tests have been realized on the equivalent execution platform on which the real functionality has been executed when running the application example on the robot.[22] While the robot provides a more powerful processor like it is the case for the most Electronic-Control-Units (ECUs) used within a modern car, even by using a platform or processor, which has only a tenth of the computation power we will not reach an overhead concerning the reconfiguration leading to an execution time much greater than one millisecond.

6 Conclusion

In this paper we have presented an approach to extend AUTOSAR architectures with reconfiguration capabilities. The approach fits into existing toolchains for the development of AUTOSAR systems and allows reusing tools, which where currently used. The overhead added to the resulting reconfigurable architecture has been shown to be minimal but the developer rewards an easier development of reconfiguration logic, which otherwise has to be done manually at the functional / implementation level. We have successfully shown that it is possible to use high-level architectural modeling techniques without generating massive run-time overhead.

Although our approach has only been evaluated in the context of AUTOSAR it should be applicable to almost any component based development approach.

For the future we plan to also support the reconfiguration of distributed compositions. From an architectural point of view a distributed composition does not differ from a local one, as AUTOSAR completely hides the communication details in the Runtime Environment-layer from perspective of the application layer. Anyway, a distributed scenario contains enough challenges such as timing delays, Basic Software configuration, deployment decisions concerning Routing-Components, just to name a few. Further the high-level architectural modeling

[21] For a better understanding we decided to only show four sensors in the previous sections.

[22] The robot is equipped with 300 MHz processor.

we have introduced in this paper also allows the verification of the modeled systems. First attempts in these directions have been very promising and we are looking forward to look into the details.

Acknowledgment We thank the dSPACE GmbH and especially Dirk Stichling and Petra Nawratil for their support in setting up the project. We want to thank the participants of the student project “A run-time environment for reconfigurable automotive software” : Christian Lück, Johannnes Dyck, Matthias Richly, Nico Rehwaldt, Thomas Beyhl, Thomas Schulz and Robert Gurol.

References

1. Burmester, S., Giese, H., Münch, E., Oberschelp, O., Klein, F., Scheideler, P.: Tool Support for the Design of Self-Optimizing Mechatronic Multi-Agent Systems. International Journal on Software Tools for Technology Transfer **10**(3) (2008) 207–222
2. Giese, H., Burmester, S., Schäfer, W., Oberschelp, O.: Modular design and verification of component-based mechatronic systems with online-reconfiguration. In: Proc. SIGSOFT ’04/FSE-12, New York, NY, USA, ACM Press (2004) 179–188
3. Feng, L., Chen, D., Törngren, M.: Self configuration of dependent tasks for dynamically reconfigurable automotive embedded systems. In: Proc. of 47th IEEE Conference on Decision and Control. (2008) 3737–3742
4. Anthony, R., Ekeling, C.: Policy-driven self-management for an automotive middleware. In: HotAC II: Hot Topics in Autonomic Computing on Hot Topics in Autonomic Computing, Berkeley, CA, USA, USENIX Association (2007)
5. DySCAS Project: Guidelines and Examples on Algorithm and Policy Design in the DySCAS Middleware System, Deliverable D2.3 Part III. (February 2009) Available online: http://www.dyscas.org/doc/DySCAS_D2.3_part_III.pdf.
6. Pohl, K., Böckl, G., van der Linden, F.: Software Product Line Engineering. Foundations, Principles, and Techniques. Springer, Berlin Heidelberg New York (2005)
7. Hallsteinsen, S., Hinchey, M., Park, S., Schmid, K.: Dynamic Software Product Lines. Computer **41**(4) (2008) 93–95
8. Kim, M., Jeong, J., Park, S.: From product lines to self-managed systems: an architecture-based runtime reconfiguration framework. In: DEAS ””05: Proc. of the 2005 workshop on Design and evolution of autonomic application software, New York, NY, USA, ACM (2005) 1–7
9. Kim, D., Park, S., Jin, Y., Chang, H., Park, Y.S., Ko, I.Y., Lee, K., Lee, J., Park, Y.C., Lee, S.: SHAGE: a framework for self-managed robot software. In: Proc. SEAMS ’06, Shanghai, China, ACM (2006) 79–85
10. Georgas, J.C., Taylor, R.N.: Policy-based self-adaptive architectures: a feasibility study in the robotics domain. In: Proc. SEAMS ’08, New York, NY, USA, ACM (2008) 105–112
11. Talpin, J.P., Brunette, C., Gautier, T., Gamatié, A.: Polychronous mode automata. In: EMSOFT ’06: Proc. of the 6th ACM & IEEE International conference on Embedded software, New York, NY, USA, ACM (2006) 83–92
12. AUTOSAR GbR: List of Basic Software Modules. Version 1.3.0.
13. AUTOSAR GbR: Specification of RTE. Version 2.1.0.
14. AUTOSAR GbR: Specification of the Virtual Functional Bus. (2008) Version 1.0.2.
15. AUTOSAR GbR: Specification of Operating System. (2009) Version 3.1.1.