Aspect-Oriented UML Modeling for Developing Embedded Systems Product Lines

Workshop on Aspect-Oriented Modeling with UML
International Conference on Aspect-Oriented Software

JORGE L. DÍAZ-HERRERA, JASMIN CHADHA, AND NEIL PITTSLEY
School of Computing & Software Engineering
Southern Polytechnic State University
1100 South Marietta Parkway, Marietta, GA 30060-2896
Tel: +1 770-528-5558 FAX: 1 770-528-5511 E-mail: jdiaz@spsu.edu

1 Introduction

Plummeting hardware costs are pushing for a rapid computing growth in new application areas, and especially in the embedded systems arena. Indeed, computers in systems products and services have become very pervasive, and the market is expected to continue to grow rapidly [1]. Embedded system applications range from "small" scale devices such as smart phones, personal digital assistants (PDAs), set-top boxes, to large enterprise servers and network computers, radar systems, automotive, avionics, etc.

In today's post-PC era, embedded systems are increasingly more software-driven, as more and more of the functionality is placed in software. Although the conventional software industry has been relatively successful, little progress has been observed in the area of development methodologies for embedded systems, and current approaches are not entirely satisfactory. For the most part, embedded software is still written by hand in response to natural language descriptions of functionality, which is custom-written repetitively from scratch each time a system is built for the hardware being considered. Embedded system development life cycle processes are fundamentally waterfall, top-down, single system driven. The results are monolithic (i.e., without interchangeable parts), one-of-a-kind systems that are difficult to change in order to upgrade and customize. Product behavior is inevitably validated late in the development cycle, with typical consequences of costly errors, lower overall quality, delayed delivery, and high overall development costs.

Increasing industry pressures to boost productivity and to reduce time-to-market, point to the need to look for ways to change the current situation. One direction is to move our focus from engineering single systems to engineering families of systems using a component-based, reuse-driven approach rather than the traditional monolithic, one-of-a-kind designs. For software intensive systems, a reuse-driven approach will potentially reduce time-to-market, and improve product quality while reducing uncertainty on cost and schedule estimates [2]. Additional benefits include longer time-IN-market, amortization of up-front development cost, and reduced training and maintenance costs [3].

A factor that holds back reuse in the embedded systems domain is the general *lack of firm standards* in many areas. For example, the complexity of embedded systems has driven a proliferation of specialized modeling notations. There are many languages in used today ranging from a varaiety hardware description languages, continuous modeling languages, protocol specification languages, dynamic synchronization languages, architrecture description languages, programming and module interconnection languages, and a pletora of formal languages and mathematical notations. (For a fairly complete list see [4], and for a good survey see [5]). Nevertheless, current notations are not entirely satisfactory for all aspects of embedded systems development.

Development environments are fragmented, with little interoperability between various levels of tools, and poor portability across environments. Industry-adopted connectivity standards between development environments involving mixing languages, cross-platform compilers, and multilingual debuggers are not readily available. A fundamental problem we are facing is the inability of representing all the design information in an appropriate standard modeling notation, one that is suitable at various levels of abstraction and that can present the design from various view points.

In response to these issues our effort in the Yamacraw Embedded Systems (YES) project [6] is directed at bringing modern software engineering to the embedded software development world. Indeed, modern software engineering, development processes, and construction technology are not commonly found in embedded systems projects. For example, only recently has there been some discussion on the use of *object technology* [7]. Advancements in mainstream software technology, such as aspect-oriented design and separation of concerns, have not found their way into embedded systems development yet.

Our work fundamentally impacts current practice on several fronts, more specifically on the following:

- A broad-spectrum notation and integrated methodology for specifying requirements and designs from high-level behavioral descriptions to the register-transfer logic (RTL) level
- A product line design approach (domain engineering) that promotes IP reuse
- An aspect-oriented, component-based implementation (application engineering) that increases productivity
- Application-specific development environments with support for early validation

2 A broad-spectrum notation

We propose to use a unified notation to address the multiple-language, multiple-analyses problem of embedded systems design. Previous attempts to address this problem have taken two avenues, a *co-simulation* approach based on a common backplane, and a *compositional approach* based on a single, **"**integrated**"** model. In the former the various analysis tools corresponding to the different notations are cognizant of the semantics of an underlying common backplane. The basis for the second, compositional, approach is the translation of all different notations into a single "integrated" model with its own analysis tools. Our approach is a variant of the latter.

The idea is to use a general modeling language such as UML [8] and to define syntactic isomorphic mappings, and corresponding semantic homomorphism, to existing notations, each supporting various modeling techniques familiar to embedded systems designers. It is worth mentioning that UML itself is an attempt to unify several orthogonal object-oriented modeling elements such as data (classes), behavior (states), and execution flow (actions), each with its own set of notations and semantics. The UML definition, however, as it stands at the time of writing, is not entirely satisfactory for modeling applications in the real-time embedded systems domain. It lacks sufficient semantics in the fundamental concepts that are critical for this domain[1].

Nevertheless, UML includes built-in *extension mechanisms* that allow refinements of the notation to include concepts found in a particular domain. There are several implementations as part of support tools to respond to the real-time, embedded systems challenge [9, 10, 11, 12]. These extensions, known as *profiles,* however, are still fairly low-level, and they do not directly address system-level modeling concepts, nor do they include full support for all the abstractions of an embedded systems methodology that covers partitioning, Hw/Sw co-design/co-simulation, etc.

In this regard, we propose a broad-spectrum notation combining levels of abstraction (vertical integration), and heterogeneous conceptual models (horizontal integration), built as a series of extensions

[1] At the time of writing, there are four OMG's request-for-proposals to deal with the deficiencies, one dealing with Action Semantics, another with Scheduling, Performance, and Time, and the last two related to quality of service (non-time related), and modeling of complex systems.

to UML. This mapping works in two ways, namely at the front-end, analysts deal directly with UML models, and at the back-end designers are able to use their conventional analysis tools which are "seamlessly" connected via UML model interchange technology such as XMI, ready for integration with other models, and/or for further processing by analysis tools such as model-checking programs. This complements other YES projects [13].

We have already implemented an isomorphic mapping of SpecC to UML. SpecC integrates system-level concepts and the C programming language. Although SpecC is not an object-oriented language, it supports the notions of *instantiations*, *interfaces*, and *implementations* without the "complexities" of a full-blown object-oriented language. There is a clear separation between computation and communication where behaviors model computation, and communication is modeled by using shared variables and/or channels. Fundamentally, SpecC specifies a system through the concepts of *behaviors* that interact via *channels* through *ports* and *interfaces*. Figure 1 illustrates the YES-UML SpecC mapping.

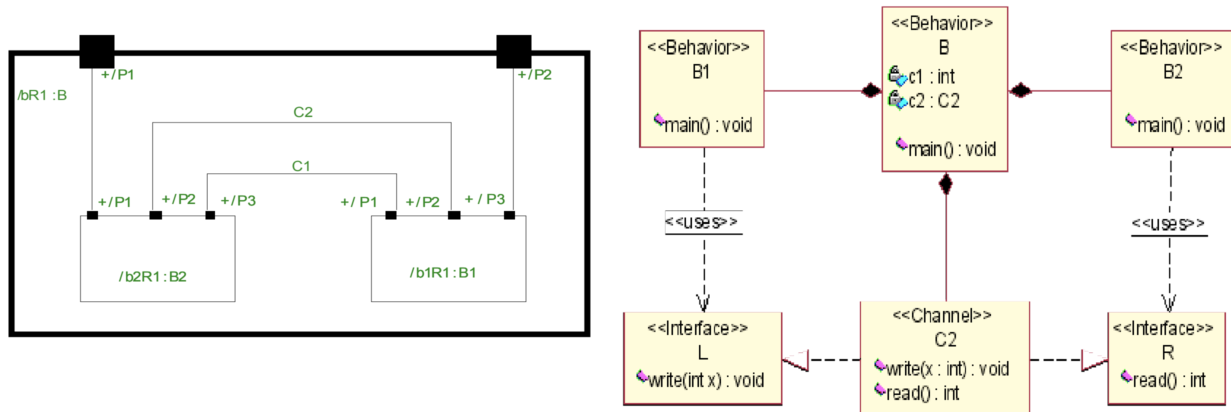


Figure 1: YES-UML SpecC example (automatically generated from SpecC code).

3 Embedded Systems Product Lines

Our development approach incorporates concepts in support of embedded systems development based on a fundamentally product line-driven process. Product lines capture the commonality across a given domain so that new products can be created easily by interchanging components, changing design parameters, leveraging established synthesis and testing suites, or other development artifacts. The focus is thus one of analysis and design of variability within a product line and the analysis and design of commonality across product lines [14]. This takes special consideration of contextual differences to allow an asset to cater to the variability found in the various products, while designing for commonality across products. Optional subsystems (or parts) are very important for component-based development within a product line. Possible combinations of optional features must be supported by two complementary elements: (1) the derivation of variants from the product line architecture, and (2) the flexibility incorporated in the component design, otherwise it may be impossible to reuse an asset "as designed" because commonality and specificity are mixed. This would make it necessary to modify the asset when reused in the new context, and this should obviously be avoided whenever feasible.

The various concerns are illustrated in Figure 2 as different axes in a multidimensional space. ***Domain models*** describe typical systems features, including functional and non-functional, as well as mandatory and optional features. ***Design models*** describe the generic solutions that are the result of domain design and implementation. Solution models represent both software and hardware architectures (i.e., components and their interfaces) suitable for solving typical problems in the domain. Product line

architectures depict the structure for the design of related products and provide models for integrating optional/alternative components. This domain analysis identifies the high-level architecture along with points of variation, "hot spots," for versioning flexibility. Component designs specify the structure for the explicit variability of components across products and product lines; they serve as models for specifying and encapsulating commonality.

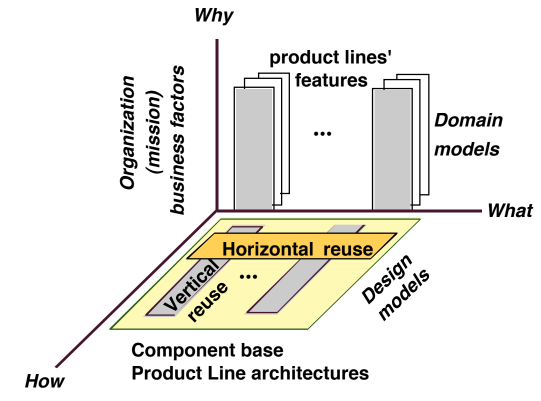


*Figure 2: **Software Product Lines***

Design models also include configuration models with specific information to support adaptation. A ***configuration model*** maps between the problem and solution models in terms of product construction rules, which translate capabilities into implementation components. These rules describe allowable *feature* combinations, default settings, etc. For example, certain combinations of features may not make sense; also, if a product does specify certain features, some reasonable defaults may be assumed and other defaults can be computed, while on some other features may be disallowed. The configuration model isolates abstract requirements into specific configurations of components in a product line architecture.

Figure 3 illustrates the process of generating domain models and actual products.

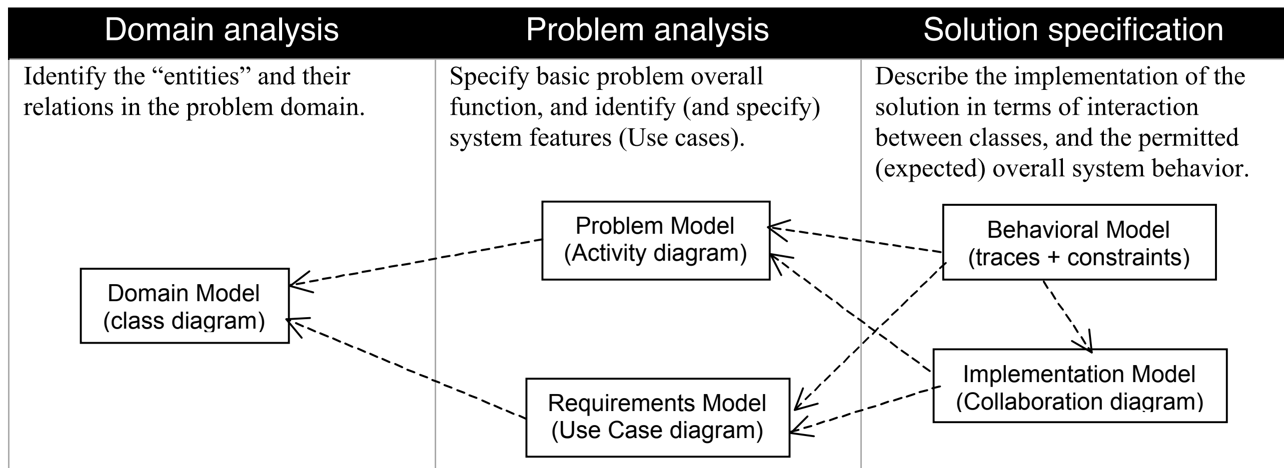


Figure 3: High-level workflow linking domain engineering and application engineering activities

4 An aspect-oriented, component-based implementation

Reusable models are likened to partially completed applications that are later customized and/or specialized, when completed for a specific set of requirements. Given generic solutions, a specific solution is an adaptation of this base solution.

Integration of concerns is fundamental for "building" software by instantiating models. For embedded systems, the notions of separation of concerns and aspect-oriented development provide a simple method to decouple structural (physical) decomposition aspects from purely behavioral (logical) concerns. The "scattering and tangling effect across the object-oriented design" [15] is also relevant for embedded systems.

Informal analysis and design methods are prone to ambiguity and inconsistency. Our notational "technique" is augmented with formal specifications of behavior as a separate concern.

Let us explain. Assume that units of reuse can be "slices" of behavior as specified by use cases. We treat this reusable functionality as a generic, composable artifact that encapsulates collaborations, since use cases typically affect more than one class. Thus, systems' features are specified as particular collaborations among objects. The services produced and required by a class can be specified by regular expressions of service requests statements as a set of possible "traces" (following Coleman approach [16] with Objectchrats). Collaborations, in turn, specify traces for a set of objects. A specific feature of a system can thus be represented by a sequence of objects' service requests (the set of traces that are possible from the point of view of the objects participating in the interaction). Please, refer to the appendix for a complete example.

Checks can thus be made to ensure collaborations are possible as per the domain model, and that the sequence of collaborations is sound for implementing the stated requirement. The former can be done using Coleman's trace functions. The latter requires the orthogonal specification of system behavior as a set of "conditions" for the instantiations of the various collaborations. We can also use these "conditions" over method invocation history in order to satisfy separate concerns like non-functional requirements such as synchronization, mutual exclusion, liveliness, etc., and perhaps other Quality-of-Service concerns. We use the approach originally proposed by Atkinson [17], connected by " deontic" operators (" obl" and "per" for obligation and permission). The combination of concerns is illustrated in Figure 3. Again, refer to the appendix for a complete example.

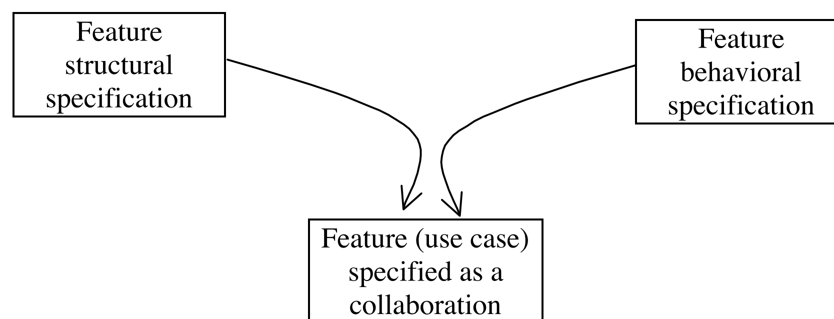


Figure 4: Integration of concerns

5 Application-specific environments

Designing-for-commonality and control-of-variability, the two needed aspects for product line development, require the establishment of a reuse infrastructure. We are identifying support for product line development in the form of ***domain-specific development environments*** that incorporate specialized design representations for which domain-specific tools would be available. The resulting products can be derived from a product line architectural design by populating generic models with domain and application-specific information.

Particularly important is early error detection of both design flaws and unexpected interactions between different aspects of the design; these can be caught before costly and restrictive implementation commitments have been made. The focus here is to develop an executable specification capable of expressing concurrency and other critical aspects in the system being described such as timely communication and synchronization. Components/systems must not be specified with design information tightly coupled, e.g., for a specific technology. Specifications must be "portable" and independent of specific simulators, capable of being verified in different environments (or chips). To aid in the validation and verification processes, specification languages must be either formal or executable (or both). Specific issues that need addressing for system-level validation include behavior of mixed signals computations (analog, digital, wireless), real-time related behavior, size, weight, area, and power (SWAP) constraints, hardware/software interfaces, system integration and co-simulation, etc. There will also be a prototyping,

design exploration facility to allow rapid evaluation of designs at various levels of abstraction and detail, ranging from high-level to RTL specifications.

To further demonstrate the validity of our approach, we did a complete isomorphic UML mapping of SpecC syntax and its semantic preserving transformation, and built the corresponding processing tools. We have a parser to convert textual SpecC specifications into XMI files, and another to convert XMI files into SpecC modules. Thus, this transformation tool works in both directions, SpecC-to-UML and vice versa, using XMI as an intermediate representation and repository. SpecC specifications are submitted to the various tools for compilation and execution (simulation). Figure 5 illustrates this concept. We have tested our approach by completely modeling a JPEG encoder. Currently, we are developing and building an integrated framework, SxUxS (pronounced "Zeus"), which incorporates the transformation tools, as well as, model validation and simulation tools.

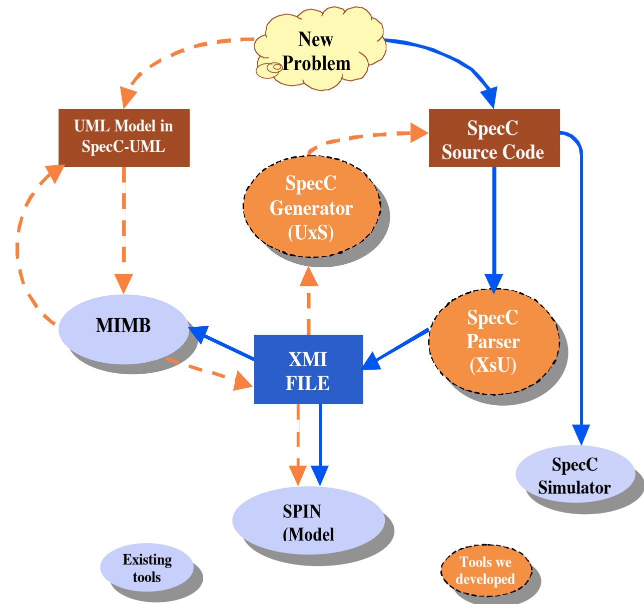


Figure 1: The SpecC-XMI-UML toolset (SxUxS, pronounced "Zeus")

We are currently experimenting with Tengger and Hyper/J to specify products by selecting features form the domain model using traces (collaborations) constrained by corresponding behavioral specification (using deontic operators).

6 Conclusions

Ideally, embedded systems should be portable over a variety of platforms, but these applications are highly device and platform-dependent. Product lines can greatly help here since they raise reuse above the single component level to the design level, and even to the requirements level.

We have developed the Yamacraw Embedded Software Methodology (YES-M) framework that leverages UML; for this, we have created an extended form of the UML, called YES-UML, which includes recent advances in system-level modeling. We have formalized mappings from UML to state-of-the-practice hardware/software specification languages, specifically SpecC, that can be used directly as input to hardware/software codesign techniques and smart compilers. YES-M uses domain modeling and analysis to create software product lines tailored to the Yamacraw target areas. We have implemented a concrete embedded system demonstration as an example to use as a concrete instantiation of the YES-M framework and to also validate it. The demonstration scenario is the design of an embedded system for real-time distributed traffic monitoring, analysis, and reporting to embedded GPS-based displays in automotive vehicles. We have modeled a variety of different views of this application using UML to experiment with and assess the current support for modeling embedded systems designs.

We are in the process of designing a prototype application-specific development environment, YES-E, which would take UML views of an embedded system product line and would allow the engineer to map certain aspects of the views to specialized more formal representations. Associated formal model checkers can check these representations. This mapping is achieved via a parser to extract features from UML diagrams (XMI) constraints and export them to external tools for formal analysis (e.g., model checking, schedulability analysis, or analog simulation). This will allow rapid evaluation of designs across a product line, without costly design implementation. It also captures a wide-range of non-homogenous constraints (SWAP, environmental effects, QoS, safety factors, etc.) that can be verified/

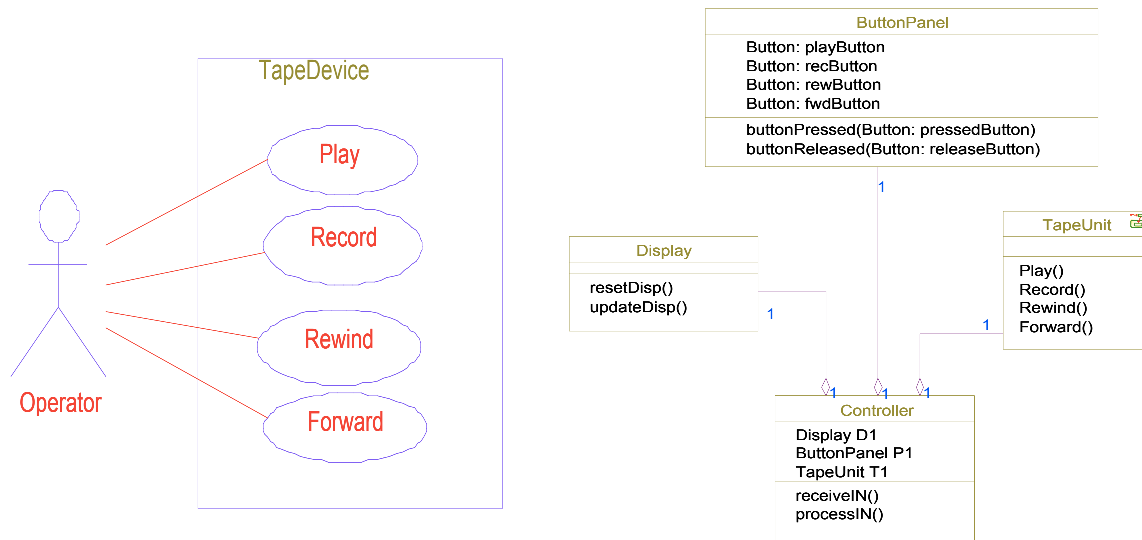
validated. Finally, we are assessing implementation technology and issues that have reached the programming level with the notion of "adapters" and aspect-oriented programming as in AspecJ (from Xerox), Hyper/J (from IBM), and Demeter.

Some of the specific challenges and open questions we are currently addressing are:

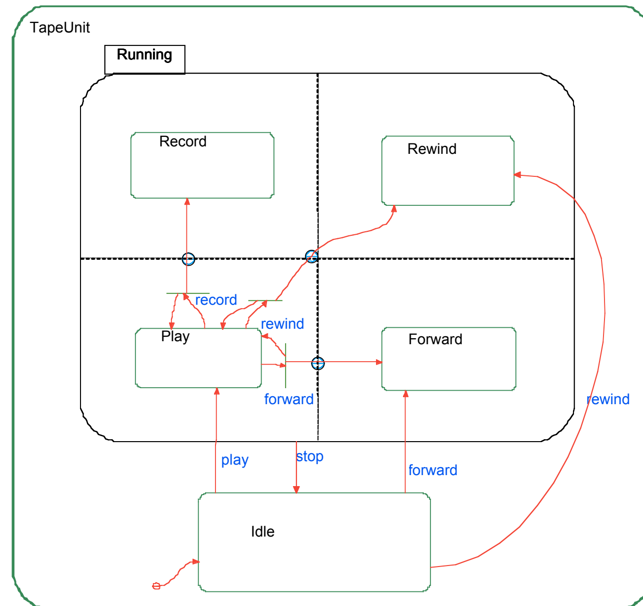
- Develop a unified, integrative methodology for embedded system design and development and assess UML's suitability for modeling embedded systems.
- Explore the implication of aspect-orientation to the design of embedded systems and heterogeneous architectures; what elements should an aspect-based design model incorporate for embedded systems? How can this be described in UML?
- What properties should aspect-oriented designs have to facilitate integration of aspects and components for evolution and reuse? What does IP reuse mean in the context of aspect-oriented development.
- Manipulating UML collaborations as architectural components subject to crosscutting concerns. What is a suitable join-point in a system-level modeling language such as YES-UML? How can the weaving process be described?
- Is UML a sufficiently general and flexible notation to capture syntactic and semantic information of design models for embedded heterogeneous systems?
- Can XMI files of substantially complex models be processed effectively and efficiently? (XMI is our intermediate notation between tools.) Can we maintain interoperability between different levels of abstractions and notations using XMI?
- Can we extend model checking to cover not only behavioral requirements but also virtual system integration and testing of interacting features?

Appendix: Example

In this example, we look at a set of related products (VCRs, answering machines, etc.) and show how to specify the various features.



Use Case model: (product features) and Domain model (class diagram)



Object Diagram for Tape Unit:

The valid traces for the TapeUnit are listed below:

```
Tr1(TapeUnit)
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>

Tr2(TapeUnit)
   <Controller, rewind, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>

Tr3(TapeUnit)
   <Controller, forward, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>

Tr4(TapeUnit)
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, rewind, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>

Tr5(TapeUnit)
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, forward, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>

Tr6(TapeUnit)
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, record, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>

Tr7(TapeUnit)
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, rewind, self>
   <self, updateDisp, Display>
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
```

```
<self, updateDisp, Display>

Tr8(TapeUnit)
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, forward, self>
   <self, updateDisp, Display>
   <Controller, play, self>
   <self, updateDisp, Display>
   <Controller, stop, self>
   <self, updateDisp, Display>
```

The rules written in Deontic logic (using DRAGOON syntax) below ensure the following:

- Record can be executed only if play is pressed
- Forward, rewind, record can be executed if one of them is already executing.

```
Obl (record) ⇔ active(play) = 1;
Per (INDIVIDUAL)⇔active(INDIVIDUAL)=0;
```

where

```
   INDIVIDUAL => record, rewind, forward;
```

Trace for the "play " use case would be:

```
Tr (Play)
   <*,buttonPressed(play), buttonPanel>
   <buttonPanel, receiveIN(),controller>
   <controller, play, tapeUnit>
   <tapeUnit, updateDisp, display>
   <controller, play, self>
   <*, buttonPressed(stop), buttonPanel>
   <buttonPanel, receiveIN(),controller>
   <controller, stop, tapeUnit>
   <tapeUnit, updateDisp, display>
```

References

[1] Chiang, S-Y. "Foundries and the Dawn of an Open IP Era." IEEE Computer, April 2001, pp. 43-36.

[2] Díaz-Herrera, J. L. and V. Madisetti "Embedded Systems Product Lines." Software product lines, ICSE Workshop. Limerick, Ireland. June 2000.

[3] Díaz-Herrera, J. L., Peter Knauber and Giancarlo Succi. "Issues and Models in Software Product Lines" (International Journal of Software Engineering & Knowledge Engineering, 8, 2001).

[4] http://www.inmet.com/SLDL/notations.html

[5] Mermet, J. "System on Chip Specification and Design Languages Standardizaiton." proceedings of *Ada-Europe'99: Reliable Sofwtare Technologies*, M. González-Harbour and J. A. de la Puente, Eds. (Lecture Notes in Computer Sceince, no. 1622, Springer-Verlag, 1999).

[6] Díaz-Herrera, J. L. and V. Madisetti "The Yamacraw Embedded Software (YES) Methodology: A Technical Analysis", CSIP TR-00-01, Electrical and Computer Engineering, Georgia Tech. 31 January 2000." (http://users.ece.gatech.edu:80/~vkm/TR/2000/yesmeth.pdf)

[7] ISOR-2002. Washington DC, USA April 2002.

[8] OMG *Unified Modeling Language Specification*, Version 1.3 First Edition: March 2000 (http://www.omg.org/)

[9] Selic, B. and J. Rumbaugh. *Using UML for Modeling Complex Real-Time Systems*. ObjectTime Limited, 1998.

[10]Moore, A. and N. Cooling. *Real-Time Perspective – Overview*. (Version 1.0), ARTiSAN Software, 1998.

[11]Douglass, B. P. *Doing Hard-Time: Real-Time Systems with UML*. 1999.

[12]LeBlanc, P. and V. Encontre. *ObjectGeode: Method guidelines*. Verilog SA, 1996.

[13]Bobbie, P., Buggineni, V., and Ji, Y., "Model Checking with sf2smv/SMV and Simulation of Parallel Systems in Matlab," Huntsville Simulation Conf. (Society of Computer Simulation), Huntsville, AL, October 3-4, 2001.

[14]Díaz-Herrera, J. L. and R. Schroeder. "Product Line Software Development for Embedded Systems: a Model-Based, Component-Driven approach" (OOPSLA'99 workshop on Object Technology & Product Lines, Denver Oct. 1999).

[15]Clark, S. "Extending standard UML with model composition semantics" Technical report, trinity College, Dublin, Ireland. 2001.

[16]Coleman, D. F. hayes, and S. Bear. "Introducing Objectcharts or How to Use Statecharts in Object-Oriented Design." IEEE Trans. On Software Engineering, Vol. 18, no. 1, Jan 1992, pp 9-18.

[17]Atkinson, C. *Object-Oriented reuse, Concurrency, and Distribution*. Addison-Wesley; 1991.