Missouri S&T

Missouri University of Science and Technology
Scholars' Mine

Mechanical and Aerospace Engineering Faculty Research & Creative Works

Mechanical and Aerospace Engineering

07 Aug 2002

Object-Oriented Modeling and Fault Detection of a Powder Feeder for a Laser Metal Deposition System

Robert G. Landers
Missouri University of Science and Technology, landersr@mst.edu

Michael Gene Hilgers
Missouri University of Science and Technology, hilgers@mst.edu

Frank W. Liou
Missouri University of Science and Technology, liou@mst.edu

Bruce M. McMillin
Missouri University of Science and Technology, ff@mst.edu

Follow this and additional works at: https://scholarsmine.mst.edu/mec_aereng_facwork

Part of the Business Commons, Computer Sciences Commons, and the Manufacturing Commons

Recommended Citation

R. G. Landers et al., "Object-Oriented Modeling and Fault Detection of a Powder Feeder for a Laser Metal Deposition System," *Proceedings of the 13th Annual Solid Freeform Fabrication Symposium (2002, Austin, TX)*, pp. 271-278, University of Texas at Austin, Aug 2002.

This Article - Conference proceedings is brought to you for free and open access by Scholars' Mine. It has been accepted for inclusion in Mechanical and Aerospace Engineering Faculty Research & Creative Works by an authorized administrator of Scholars' Mine. This work is protected by U. S. Copyright Law. Unauthorized use including reproduction for redistribution requires the permission of the copyright holder. For more information, please contact scholarsmine@mst.edu.

OBJECT–ORIENTED MODELING AND FAULT DETECTION OF A POWDER FEEDER FOR A LASER METAL DEPOSITION SYSTEM

Robert G. Landers*, Michael Hilgers†, Frank W. Liou*, and Bruce M. McMillin†

*Department of Mechanical and Aerospace Engineering and Engineering Mechanics
†Department of Computer Science
University of Missouri at Rolla, Rolla, Missouri 65409

ABSTRACT

One of the most critical components of a laser metal deposition system is the powder feeder unit. Techniques are required to efficiently design and analyze these systems and to ensure fault tolerance during the operation. In this paper, a model of the powder feeder unit, based on object–oriented abstractions of it components, is developed. This model is modular in that specific components can be efficiently updated or replaced with components that perform a similar function. In addition, the model can be used for simulation allowing for efficient design and analysis. Assurance of the correctness of the powder feeder system is obtained from concurrent run–time evaluation of temporal logic expressions. A simulation example is provided.

INTRODUCTION

Laser Metal Deposition System

A novel hybrid manufacturing system consisting of integrated laser metal deposition and machining processes has been developed at the University of Missouri at Rolla [LiCLJBA01]. The system includes a 2.5 *kW* Nd–YAG laser, custom–built powder feeder system capable of 5–15 *g*/*min* continuous flow rate, and a five–axis CNC machining center. Additional sensors have been added to the system to sense the bead profile and melt pool temperature. Monitoring and control is performed on a National Instrument's PXI 8170/850 *MHz* real–time processor. The software performs communication with the machining center, monitoring and control of the powder feeder, laser, and deposition process, and fault detection and correction. The system (hardware, process, and software) is very complicated and has stringent safety and correctness requirements. To gain a greater understanding of the overall system performance, advanced modeling techniques such as object–oriented modeling are required. Object–oriented modeling allows efficient model construction and has the added benefit of modularity (i.e., the model can easily be modified as the system design is modified with new hardware and software components). Since the safety and correctness properties are of paramount concern, advanced fault tolerance techniques must be employed to ensure both the safety and liveness properties of the software system as well as safe and productive operation of the manufacturing system.

Object–Oriented Modeling

Two approaches may be adopted to model the laser metal deposition system: algorithmic and object–oriented. The algorithmic perspective [BOST86] views all software processes as a composition of simple functions in the form of algorithms. While there is nothing inherently

wrong with this approach, software developed using this perspective has proven to be brittle. That is, a small change to a basic sub–function can have a far–reaching impact, even to the point of disabling the entire software system. The object–oriented approach, meanwhile, directly models the way individual components of the system communicate or interact. For example, the rate at which a screw feeder turns depends on the voltage given to the motor. An algorithm approach would write the functional relationship between the rate of turn and the voltage as it depends on a number of physical parameters. In the object–oriented approach, the physical parameters determine the state of the motor and screw feeder objects. The voltage is modeled as a "message" communicated to the motor object. The screw object turns at the "request" of the motor object. While this may be unusual way to view the mechanical interactions, it should be apparent that changing a motor, for example, is very localized in the software system. Hence, the object model approach claims to offer easy reusability not only of individual objects but entire designs [MEYE87]. The object model approach also facilitates utilization of the full potential of powerful languages such as C++ [STRO87] and it leads to software that is built upon stable intermediate forms and is more resilient to change [BOOC91].

Dynamic Fault Detection

Dynamic failure detection is accomplished through run–time predicate detection; if a predicate (a logical expression on the system variables) is violated, the system is halted before any harm can come. Recent work has shown the attractiveness of this approach through fault containment wrappers using safety predicates [SRFA99] and safety and liveness through temporal logic [RoFA00] to create robust microkernels. Moreover, the wrapper can monitor the correctness of Commercial–Off–the–Shelf (COTS) software by wrapping the COTS object as well as provide a standard interface [PDFS01] using languages such as Java. Existing work uses locally–implemented wrappers to ensure that predicates are correct based on the observed state and messages of the wrapped object. In many cases, however, predicates of interest involve state and message information distributed throughout the system. In this case we are interested in collecting and checking an observed global state – that which occurs across multiple objects within the system.

POWDER FEEDER SYSTEM

The powder feeder is shown in Figure 1. The hopper is the physical container that holds the metal powder. A motor turns the screw feeder and powder is delivered via gravity to the splitter that delivers powder to multiple tubes. These tubes deliver the powder to an annulus that directs the powder into the laser beam (not shown in Figure 1) where the metal powder deposits onto the substrate. Note that the powder mass flow rate at the hopper is, in general, different from the powder mass flow onto the part, which is the mass flow rate of interest. The difference is due to the physical transport delay in the tubes. The tubes and annulus may also get clogged. A tachometer generates an electrical signal proportional to the motor speed. In this scenario, the tachometer signal is calibrated via off–line experiments to the actual powder mass flow rate. The tachometer is described by a static gain relating voltage to angular velocity. Tachometers may encounter faults such as electrical noise and a disconnected wire.

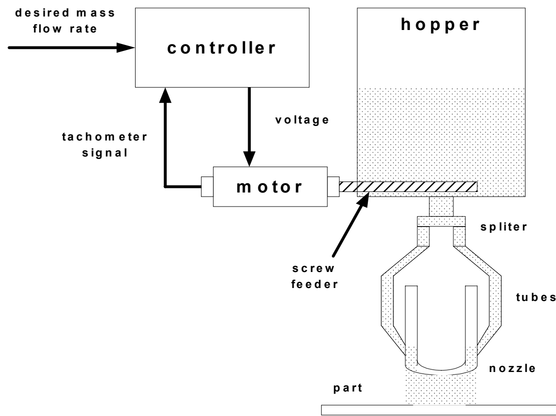

Figure 1: Powder Feeder System Schematic. The screw feeder powder that is gravity feed to the laser beam (not shown) that melts the powder that is deposited on the part and subsequently solidifies. The controller regulates the powder mass flow rate by adjusting the motor voltage. The input to the controller is the desired mass flow rate and the screw feeder tachometer signal that is precalibrated to the actual powder mass flow rate.

The powder feeder controller is part of a hierarchical control system [BoLMARL02] that monitors and regulates operation productivity and part quality. A higher level in this hierarchical controller specifies the desired powder mass flow rate to the powder feeder system controller, which has several responsibilities. First, it must determine the voltage to send to the hopper motor. To do this, it appeals to a mathematical model of the motor in the form of a delay differential equation. Physical parameters included in this model are angular position, speed, acceleration, input voltage, and disturbance torque. The numerical approximation of this delay differential equation is a history dependent difference equation that requires the desired mass flow rate, a finite history of previous measured flow rates, and the physical parameters to estimate the required motor voltage. Other responsibilities of the controller include calculating errors, saturating the control output, and communicating with a digital to analog converter. The raw tachometer signal is processed via a low pass filter and the controller periodically obtains the signal using an analog to digital converter. The controller also monitors faults such as powder clogging and an empty hopper.

Object–Oriented Modeling

In developing an object–oriented model of a system, one begins by simply describing it. Nouns are a good starting place for the basic objects [BRJ99]. For instance, the powder feeder system is composed of a Hopper[1], Motor, Tube, Splitter, DeliveryBarrel, Material, Generator and a screw or, as in this experiment, a `DimpledShaft`[2]. Next, the verbs used in describing the system suggest the ways in which objects interact. For example, the Motor changes the state of the DimpledShaft by turning it. Hence, the DimpledShaft should have a `turn()` operation. Likewise, objects must be able to answer questions concerning themselves, such as the Hopper letting the society of object know the rate at which mass is leaving it through a query operation like getFlowOut().

[1] This font will be used to indicate the name of a class. A class is a set of objects, just like int refers to the objects …, –2, –1, 0, 1, 2, … .

[2] In this experiment, the material powder is moved using a so–called dimpled shaft rather than a screw. The dimple shaft has small holes of various shapes in it. As it rotates, these dimples fill with powder then empty out of the delivery barrel.

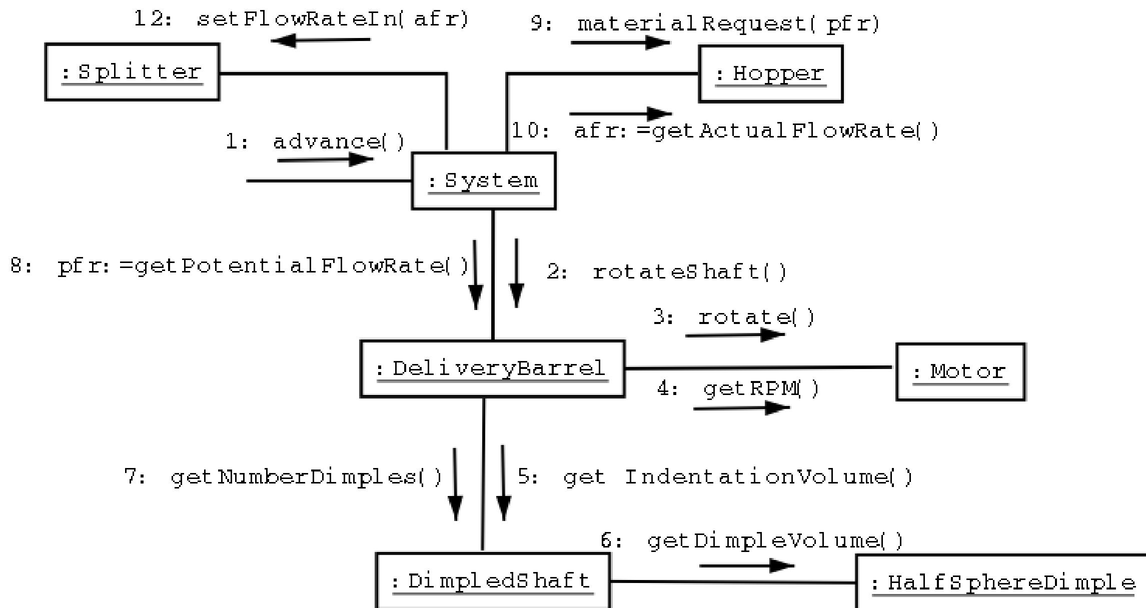

Figure 2: An UML Collaboration Diagram Modeling the Interactions Required to Advance One Time Step.

Once basic the basic structure of the objects is know, the focus turns to a set of objects. Typically, a fundamental task performed by the system is chosen and the necessary interactions between objects are modeled. Tools to do this are supplied by the *Unified Modeling Language* or UML [BRJ99], which is a graphical language. Figure 2 shows how a collection of objects (the rectangles) communicates (the message arrows) along links (the lines) in order to advance the system a time step. The order of the messages is indicated by the numeric prefix. Even a person new to UML and powder feeder systems can follow the sequence of messages and probably ascertain what is happening; however the experienced individual learns much. For example, the HalfSphereDimple class must have an operation called getDimpleVolume(). In order to calculate the dimple volume, the radius must be known. This becomes an attribute of the class and so forth. See Figure 3[3].

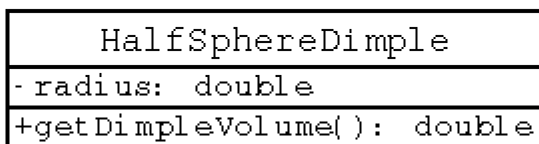

Figure 3: A UML Class Diagram.

Once the basic structure and interactions are modeled, advanced techniques can be employed to improve the vitality of the software. To illustrate, suppose that the engineers are considering three different shafts that vary by the type of dimple each with their own volume calculation. The traditional approach would be to use “if–then–else” to determine the type of the dimple before calculating the volume. This is very error prone. A simple object–oriented approach would be to create a new class for each type of dimple that knows how to calculate the volume for its dimple. Then each time a system is instantiated, the programming must change the code and recompile to accommodate a change in dimple type. This is inefficient.

[3] In UML, the second sub rectangle lists the attributes and the bottom the operations. Types are given after the name. A minus sign means the visibility is private whereas the plus sign means public.

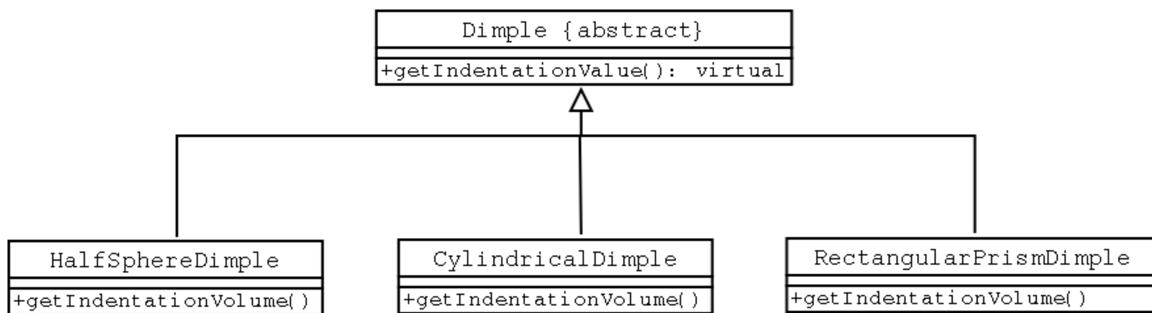

Figure 4: A Diagram Showing the Dimple Class Generalizing the Other Three Classes.

A better approach is indicated in Figure 4. The DimpledShaft object will have a number of generic dimple object pointers. At the time the system is instantiated, these pointers will be directed to one of the children of Dimple; the compiler determines which getIndentationVolume() function to used based upon the type of the child. This structure frees the program from making type–sensitive decisions. This technique is called *polymorphism*.

Polymorphism can also be used effectively in handling faults during the run of the system. From the perspective of error management, every Component of the system has a Status that must be checked. Whether or not a Component is in some form of error state will differ by part type; yet each must respond to a checkStatus() query. Similarly, each Component will respond to an error state differently, but each must be able to throwException(). All of this suggests that an ExceptionHandler could maintain a list of the generic components and use the dynamic binding of polymorphism to query specific objects at run time. This is summarized in Figure 5.

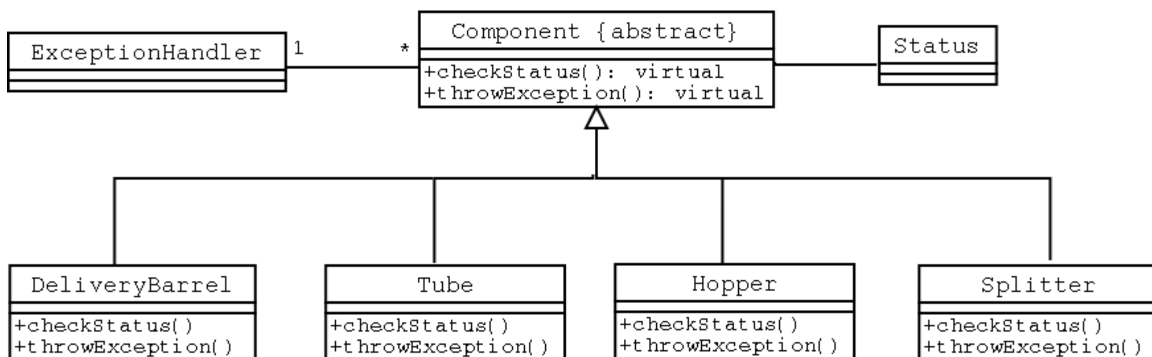

Figure 5: A UML Diagram Illustrating the use of Generalization and Polymorphism for Handling Faults of Component Pieces.

DYNAMIC FAULT DETECTION

Since system correctness and fault tolerance are expressed at a high level, temporal formulae must be expressed within the collaboration diagram. Also, the correctness of the logic expressions in a distributed computation, at run time, in the presence of partial information must be checked. Modified Interval Temporal Logic (MITL) [TsIM95] that is an event–based logic

allowing reasoning about time, if it can be expressed in terms of an event, is used. The primitive type of formula is called eval_term. This can denote a state or transition event in a state diagram, time, a condition that has to be fulfilled or it could alternatively denote a special term that is very specific to the implementation. These can be distinguished by the parentheses that we use, a square bracket indicates a state or time interval. A curly bracket indicates a condition to be satisfied.

AG[eval_term1, eval_term2]{eval_term3} Over all possible causalities of log events on the interval [eval_term1, eval_term2], {eval_term3} holds during the interval.
AF[eval_term1, eval_term2]{eval_term3} Over all possible causalities of log events on the interval [eval_term1, eval_term2], {eval_term3} holds some time during the interval.
EF[eval_term1, eval_term2]{eval_term3} Over some possible causality of log events on the interval [eval_term1, eval_term2], {eval_term3} holds some time during the interval.

Using MITL for the powder feeder example, some properties of correctness (not an exhaustive list), referencing Figure 2, are:

Safety: AG[time=T,time=T+delta]`{afr:=getActualFlowRate()>0}` – Over a particular time interval delta, the actual flow rate is positive (in other words the hopper isn't clogged or empty). Since flow is a continuous phenomena, it is expected to hold continuously over the delta time interval.
Liveness: AF[`10:afr:=getActualFlowRate()` at time T, T+.1 second]{ mass flow rate at the splitter `12:setflowRateIn(afr)=afr`}} – The time delay between a set of the mass flow rate message to the `:Hopper` and an observed change is bounded by .1 second (physically the delay until the result of the change appears at the :Splitter in the material tubes).

These properties are checked at Hopper and Splitter. Note that Splitter needs values both from itself and Hopper to check its liveness property. These are available in the simulation, but in an actual implementation, a distributed structure is used [HLML02].

SIMULATION EXAMPLE

In this section, a simulation example of the object–oriented powder feeder model is provided. The object–oriented model simulates the open–loop transient response (i.e., the controller is not included in the model) and typical faults of the powder feeder system. To make the model user–friendly, a graphical user interface (GUI) was constructed in Pearl. The main GUI screen is shown in Figure 6. From this screen, the simulation can be run in one of three modes: the motor input voltage is specified, the motor angular speed is specified, or the powder flowrate is specified. Also, faults in various components can be injected during the simulation to analyze the effects of the fault on the performance of the powder feeder system and the software. In the simulation example, a "dimpled" shaft is used in the powder feeder. This shaft has indentations machined into it. In the model, prismatic, cylindrical, and spherical are assumed. As the shaft rotates, powder is caught in the indentations and then falls into the delivery tubes. For this example, the powder flow rate is simulated for all three types of indentations. Each indentation has a different volume that can be specified by the user. There are 50 indentations in each shaft, and during the operation, each one fills up with powder at the top of the rotation and emptys the

powder at the bottom of the rotation. The simulation results are shown in Figure 7. The transient response is the same for each type of shaft since the shafts are driven by the same motor and have approximately the same inertia. Although the number of indentations and shaft angular speed are identical, the steady–state flowrates are different for each shaft since each type of indentation contains a different volume of powder. The technique of *polymorphism* allows one to easily select the indentation type and conduct the simulation for the purpose of design and simulation. In future work, multiple shaft, motor, controller, etc. types will be available to the designer.

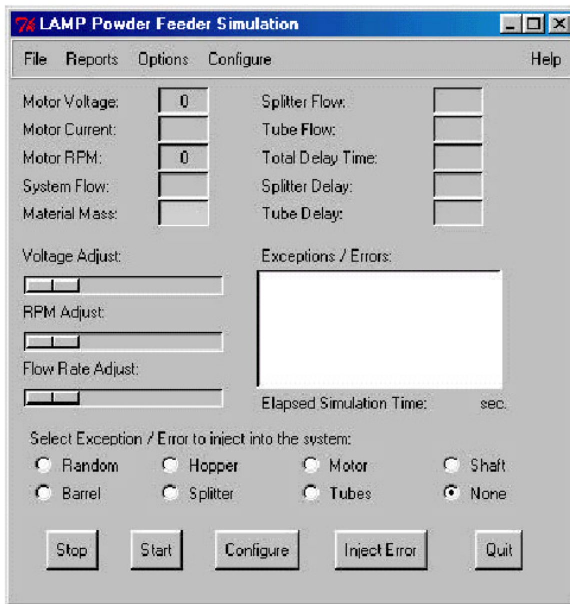

Figure 6: Main Simulator GUI.

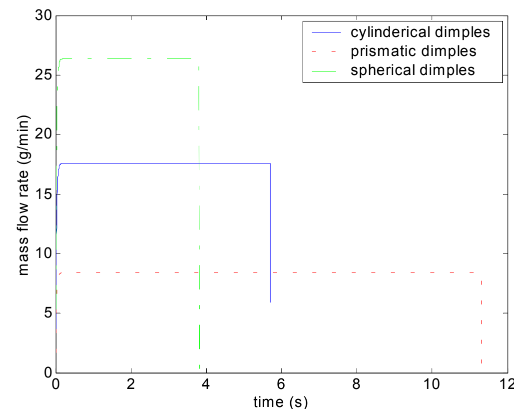

Figure 7: Mass Flow Rates for Different Shafts.

The simulation runs until the powder is exhausted at which point the **Safety** property is violated and an exception is thrown.

SUMMARY, CONCLUSIONS, AND FUTURE WORK

An object–oriented model of a powder feeder unit for the laser metal deposition process has been developed. The model can be utilized for the design and analysis of the open–loop response of the powder feeder unit. In addition, the effect of common faults can be analyzed. The strength of this model is its modularity that allows specific components to be efficiently updated or replaced with components that perform a similar function and that common faults can be easily incorporated. A simulation example was provided. The object–oriented paradigm allows the designer to easily understand the function of the entire powder feeder system and to make incremental changes when necessary. Assurance of the correctness of the powder feeder system is obtained from concurrent run–time evaluation of temporal logic expressions. In future work, the powder feeder controller will be incorporated into the object–oriented model and the model will be validated experimentally.

ACKNOWLEDGEMENTS

The authors gratefully acknowledge their students' contributions to this paper and the financial support of the National Science Foundation (DMI–9871185), Missouri Research Board, Society of Manufacturing Engineers, and Missouri Department of Economic Development.

REFERENCES

[BoLMARL02] Boddu, M.R., Landers, R.G., Musti, S., Agarwal, S., Ruan, J., and Liou, F.W., 2002, "System Integration and Real–Time Control Architecture of a Laser Aided Manufacturing Process," to appear in *Thirteenth Annual Solid Freeform Fabrication Symposium*, Austin, Texas, August 5–7.

[BOST86] Bobrow, D., and Stefik, M. "Perspectives on Artificial Intelligence Programming," *Science*, Vol. 231, p. 951.

[BOOC91] Booch, 1991, *Object Oriented Design with Applications*.

[BRJ99] Booch, G., Rumbaugh, J., Jacobson, I. *The Unified Modeling Language User Guide,* Addison–Wesley, 1999.

[LiCLJBA01] Liou, F.W., Choi, J., Landers, R.G., Janardhan, V., Balakrishnan, S.N., and Agarwal, S., 2001, "Research and Development of a Hybrid Rapid Manufacturing Process," *Twelfth Annual Solid Freeform Fabrication Symposium*, Austin, Texas, August 6–8, pp. 138–145.

[MEYE87] Meyer, B., 1987, "Reusability: the Case for Object–Oriented Design," *IEEE Software*, March, 50–64.

[PDFS01] Pawlek, R., Duchien, L., Florin, G., Seinturier, L., 2001, "Dynamic Wrappers: Handling the Composition Issue with JAC," *39th International Conference and Exhibition on Technology of Object–Oriented Languages and Systems, TOOLS39*, pp. 56–65.

[RoFA00] Rodriguez, M., Fabre, J.–C., and Arlat, J., 2000, "Formal Specification for Building Robust Real–Time Microkernels," *Proceedings 21st IEEE Real–Time Systems Symposium*, November 27–30, Orlando, Florida, pp. 119–128.

[SRFA99] Salles, F., Rodriguez, M., Fabre, J–C., Arlat, J., 1999, "Metakernels and Fault Containment Wrappers," *Twentieth Annual Symposium on Fault–Tolerant Computing*, pp. 22–29.

[TsIM95] Tsai, G., Insall, M., and McMillin, B, 1995, "Ensuring the Satisfaction of a Temporal Specification at Run–Time," *Proceedings 1st IEEE International Conference on Engineering of Complex Computer Systems*, Nov. 6–10, Fort Lauderdale, Florida, pp. 397–404.

[HLML02] Hilgers, M., Landers, R.G., McMillin, B., and Liou, F.W., 2002, "Object-Oriented and Fault–Tolerant Embedded Hybrid Systems for Manufacturing," Technical Report 02-01, University of Missouri–Rolla, Department of Computer Science.