Impact Analysis in Real-time Control Systems

Jun Li
Electrical and Computer Engineering Department
Carnegie Mellon University
Pittsburgh, PA 15213
junli@ece.cmu.edu

Peter H. Feiler
Software Engineering Institute
Carnegie Mellon University
Pittsburgh, PA 15213
phf@sei.cmu.edu

Abstract

Real-time control systems typically receive streams of sampled data signals, process them, and generate output to control devices via actuators. Control system components make assumptions about sampling rate and other characteristics, such as filtering, of data streams. These application-specific semantic dependencies and time-sensitive semantic properties are usually not explicit in the source code implementation. This paper discusses an approach to modeling the system architecture of real-time applications, capturing semantic dependencies, and recording time-sensitive properties. Analysis of the model can identify inconsistencies in a system design and can determine the impact of a system change. This architectural dependency model can evolve incrementally to record hidden side effects, as they are discovered during maintenance, and prevent such errors in the future.

1. Introduction

In this paper, we focus on large and complex real-time control systems, which are implemented on distributed hardware. These systems usually process streams of sensor inputs from the external environment and generate actuator outputs to realize closed-loop control functions. The control signals have to be produced in a timely manner, i.e., to meet real-time requirements. In addition, supervisory control provides control mode switching to handle different operational conditions, fault condition detection and failure recovery, and the like.

Such systems can become quite difficult to maintain due to the multi-dimensional inter-dependencies of components. The logical component interconnections underlie the first dimension of inter-dependencies. These dependencies are imposed by constraints associated with domain-specific application semantics. For example, a controller might be designed to track the environment signal only within a pre-defined bandwidth. Sharing of resources creates a second dimension of interdependencies in the execution environment. Changes in the execution behavior of one process may potentially affect other processes. The third dimension of interdependencies involves timing properties in the execution model and time-sensitive properties in terms of application semantics. For instance, increasing the period of one control component might make all components schedulable but it affects other application properties, such as phase margins of some of the control loops.

Real-time control systems are difficult to maintain because these intricate dependencies are not always fully documented. Hidden side effects result in residual errors that are often not uncovered the system is in operation and their root cause is difficult to identify. Elimination of such hidden side effects through model-based analysis can greatly reduce maintenance cost.

This paper presents an approach to model applications in terms of interconnecting components, capture properties and assumptions in terms of application semantics, as well as relate their time-sensitive nature to timing properties. The benefits of the model are twofold. Consistency analysis allows early identification of semantic inconsistencies as the result of a change. The model also supports impact analysis, i.e., "identifying the potential consequence of a change, or estimating what needs to be modified to accomplish a change." [1]. A precise model can guide the actual system modification effort so that it is proportional to the predicted scope of change.

Consistency analysis and impact analysis can be performed in different levels of granularity. It has been done at program statements level. Type checking in programming languages provides a mechanism of identifying inconsistencies. Program slicing [2, 3] helps to identify the scope of change through data and control flow dependencies reflected in the source code. The second level is that of source code modules. Unix utility

make, determines the recompilation for system building based on the file dependencies explicitly documented inside the makefile. Import/export relationships are used to determine potentially affected modules and possible recompilation from a change [4, 5]. The recompilation may detect syntactic and semantic inconsistencies and generate new object code. The third level is that of architectural models via Architecture Description Languages. This paper focuses on the ADL level.

Current ADLs describe systems as a set of interacting components [6]. They focus on structural dependencies, behavior constraint constructs (state/event based) on components and interactions, and scheduling analysis. Wright [7] models connector types as abstract patterns of interaction using the CSP-like notation. Rapide [8] models computations and interactions as partially-ordered event sets. In [9], the dependency chaining analysis is conducted by deriving component dependencies from the language semantics of Rapide. MetaH [10] models the process and communication architecture of applications, and performs scheduling analysis and executive generation.

We build on these approaches and focus on modeling characteristics relevant to control systems, in particular, the periodic processing of data streams. We proceed by identifying application assumptions and constraints that are likely to reflect otherwise hidden side-effects. We then describe notational constructs in an ADL to reflect application assumptions and constraints. Such constructs can be viewed as extensions to an existing ADL to complement the available notational power.

The paper proceeds as follows. Section 2 illustrates examples of identifying hidden dependencies. Section 3 describes various ways to record hidden dependencies, with the focus on those associated with application semantics and time-sensitivity. Section 4 discusses impact analysis based on the described notation and consistency checking rules in Section 3. Section 5 discusses how system models refined with these semantic constraints can be used during development and maintenance to reduce hidden side effects and residual errors. Section 6 summarizes the contributions in this paper.

2. Example System Illustration

Consider the Coordinated Inverted Pendulum system shown in Figure 1. It consists of two inverted pendulums that can independently move to different locations on parallel tracks. Each inverted pendulum is controlled by a Pendulum Controller, which is responsible for balancing it and moving it to desired locations on the track.

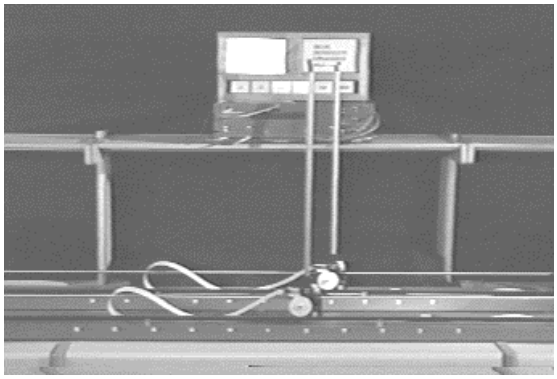

Figure 1 Coordinated Pendulums

A motion coordinator is responsible for coordinated movement, such that the Inverted Pendulums can transport a rod on their tips. This Coordinator is replicated as control of this system is performed through a set of networked computers. Motion commands are sent from a user interface to the Coordinators. They translate these commands into motion trajectories (set-points) for the inverted pendulum controllers, each focusing on control of one device based on status information from both. This control system architecture is shown in Figure 2.

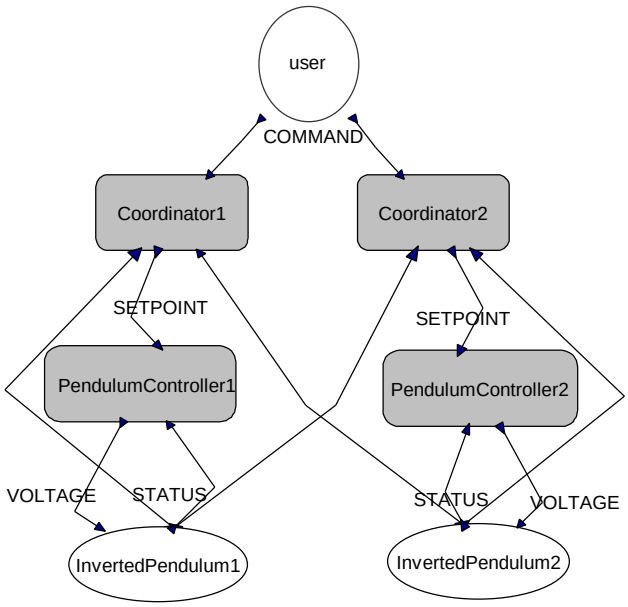

Figure 2 Coordinated Pendulum Architecture

The system of Coordinated Pendulums exhibits the following properties and assumptions.

(1) Data stream consistencies: User commands going to the Coordinators, setpoints going to the Pendulum Controllers, status information from and control information to the Inverted Pendulums are data streams. They have certain rates, either because data is periodically received/generated, or because it has

maximum arrival/departure rates. Similarly, data streams have time latencies. Rate mismatch or latency ignorance potentially result in system malfunction or performance deterioration.

(2) Data representations: Sender and recipient of data along a connection should have the same data interpretation, e.g., use the same measurement unit or coordinate system.

(3) Assumptions about interconnected components: For example, the trajectories issued by the Coordinators are a sequence of set-points at a certain rate. A Pendulum Controller has a certain responsiveness, i.e., is able to respond to set-point requests up to certain limits from the current position.

(4) Assumptions about multiple components: The Coordinators have to make the same assumption about the speed capacity of the Pendulum Controllers.

Examples of undocumented assumptions are:

(1) All components assume that pendulum positions are relative to centers of tracks, and pendulum angle measurement is relative to the upright position.

(2) A change in the rate at which setpoints are received by the Pendulum Controllers affects their speed capacity and their ability to control the device.

3. Dependency Modeling and Inconsistency

This section describes the modeling mechanism to record component dependencies and evaluate inconsistencies reflected in the dependency model. Note that our modeling notation has a graphical and a textual representation. Figure 2 shows the example system as it is displayed by our graphical editing and analysis tool.

The basic component dependencies are based on communicating connections described in Section 3.1. Various ways to attach semantic information along connections to enrich component dependencies are elaborated in Section 3.2.

3.1. Component Dependency Graphs

A system is a hierarchical composition of components and interconnections. Each component has two parts: *specification* and *implementation.* A component specification defines the component input-output interface via *ports.* Components interact through *connections.* A component can be *simple*, i.e., implemented in the source code, or *composite,* i.e., composed of a set of interconnected components. In the latter case, each specification port is *bound* to a port inside the implementation. Figure 3 shows the textual representation of the example system components including their port types.

The connection topology may be constrained by the number of connections to a port. For example, in a system without queuing, only one incoming connection is allowed to an input port.

```
component PendulumController is
    in setpoint: SETPOINT;
    in status: STATUS;
    out actuator: VOLTAGE;
end PendulumController;

component Coordinator is
    in status1: STATUS;
    in status2: STATUS;
    out setpoint: SETPOINT;
    in command: COMMAND;
end Coordinator;

component User is
    out command1: COMMAND;
    out command2: COMMAND;
end User;

component InvertedPendulum is
    in actuator: VOLTAGE;
    out status: STATUS;
end InvertedPendulum;

system CoordinatedPendulums is
  components
    user: User;
    Coordinator1, Coordinator2: Coordinator;
    PendulumController1: PendulumController;
    PendulumController2: PendulumController;
    InvertedPendulum1 : InvertedPendulum;
    InvertedPendulum2: InvertedPendulum;
  connections
    user.command1 -> Coordinator1.command;
    Coordinator1.setpoint -> PendulumController1.setpoint;
    ……
end CoordinatedPendulums;
```

Figure 3 Component Modeling

A set of interconnected components represent a directed *component dependency graph* with nodes representing components and edges representing connections. *Predecessors (successors)* of a node are nodes reached by an incoming (outgoing) edge. A component change can potentially affect its direct predecessors and successors as well as their transitive closures.

3.2. Semantic Information Constructs

Components make various assumptions about the data coming in and going out through ports. These assumptions may be about the data representation or about the semantic interpretation from application perspectives. Since the data may be processed by multiple components, the assumption may be satisfied by a

component other than the direct predecessor or successor. Assumptions may be made about properties of components that are not communicated through connections.

We proceed by discussing constructs to model assumptions about the data specific to each port connection, i.e., input/output specifications, assumptions about any predecessors or successors, i.e., predicates, as well as constructs on properties of components not communicated by ports, i.e., property constraints and dependencies.

Notice, that although the examples in the next sections are shown in textual representations, this information is associated with components and ports as attributes, and can also be added and modified by the user through the graphical editor.

Input/Output Specifications. An input specification contains expected or required constraints on data. An output port specification contains assertions on provided data. For each pair of connected ports analysis can determine whether the provided data constraints satisfy the expected data constraints.

Communicated data can be of primitive type (int, float, boolean), composite type (record), or user-defined type. User defined types are defined in terms of primitive, record, and other user-defined types. If the data represents measurements, measurement unit and coordinate system can optionally be specified. Data values may be constrained by ranges. The data types, measurement units and coordinates along a connection are expected to match, the provided ranges must be contained in the respective required ranges.

```
in setpoint: SETPOINT (cm) is
      range – (Track.Length/2 – Stability_Range) to
            Track.Length/2 – Stability_Range;
      delta – Max_Step_Size to Max_Step_Size;
      every 200 ms;
end setpoint;
out actuator: VOLTAGE(v) is
      range –4.95 to 4.95;
      every  20 ms;
end actuator;
property Stability_Range = 8 cm;
property Max_Step_Size = 10 cm;
```

Figure 4 Pendulum Controller Port Specifications

A port may handle a data stream arriving with a certain rate and departing with a possibly different rate. For every port, a *delta* specifies how much two consecutive data values can differ. Along every connection, rates should be identical, and deltas follow containment relationship from output ports to input ports.

Figure 4 and Figure 5 illustrate the input/output port specifications for the Pendulum Controllers. Components can have properties are constant values that are associated with primitive types (int, float, boolean) or user defined enumerations. The *Stability_Range* and *Max_Step_Size* are properties of the Pendulum Controllers. Track is a separately declared component with a property *Length*. The range and delta specifications can refer to local properties, e.g., *Max_Step_Size*, or to properties in other components, e.g., *Track.Length*. The referencing of component properties in other components creates definition-use dependencies across multiple components.

Analysis determines whether the types of two connected ports match, whether the measurement units and coordinate systems agree, whether the stream rates match, and whether the provided range and delta constraints are contained in the required constraints.

```
out setpoint: SETPOINT(cm) is
      range – (Track.Length/2 – Stability_Range) to
            Track.Length/2 – Stability_Range;
      delta – Max_Step_Size to Max_Step_Size;
      every 200 ms;
end setpoint;
in command: COMMAND(cm) is
      range –Track.Length/2 to Track.Length/2;
      every 200 ms;
end command;
property Stability_Range = 10 cm;
property Max_Step_Size = 5 cm;
```

Figure 5 Coordinator Port Specifications

Suppose the Pendulum Controller's *Max_Step_Size* is 3 cm instead of 10 cm. The *setpoint* port connection between the Coordinator and the Pendulum Controller is inconsistent, because the delta provided by the Coordinator is too large, which indicates the set-point may change too quickly for the Pendulum Controller to follow successfully.

In case of composite components, the port specifications involved in a binding can also be checked for inconsistencies. Again, data types, measurement units, coordinates must be matched, and provided ranges and deltas must be contained in required ranges and deltas.

Predicate Specifications. We introduce the propositional predicate notation developed in [11] to capture additional assumptions and assertions about the data communicated via ports. Predicates represent preconditions, postconditions and obligations, and we associate them with ports. Preconditions reflect assumptions or requirements to the predecessors along data streams, while obligations are assumptions or requirements on the successors. Postconditions are assertions by components. As illustrated in Figure 6, the obligation SetpointReached indicates that set-point coming through the setpoint port of a Coordinator is expected to be followed correctly. Another component, a Pendulum Controller, may assert

in a postcondition SetpointReached onto its setpoint port to indicate that the data coming into this port are the set-point data and they are handled properly.

```
//inside component Coordinator
out setpoint: SETPOINT(cm) is
    obligation SetpointReached ;
end setpoint;

//inside component PendulumController
in setpoint: SETPOINT(cm) is
    postcondition  SetpointReached;
end setpoint;
```

Figure 6 Use of Predicates

To determine whether a precondition (obligation) is satisfied, we check whether the direct predecessor can provide the matching postcondition. If the names match, the predicate is satisfied. If the predicate is in negated form and the names are matched, the predicate is not satisfied. Otherwise, we continue the search by checking their predecessors (successors). If we exhaust all the predecessors (successors), we continue to check whether the predicate is declared in the component specification – to be satisfied externally. Only if it is not declared, it is considered unsatisfied.

```
//inside component User
out command1:COMMAND(cm) is
  obligation ConductCalibration;
end command1;

//inside component InvertedPendulum
out status: STATUS is
  postcondition ConductCalibration;
end status;

//inside CoordinatedPendulums system implementation
path path1is
   user.command1→Coordinator1.command→
  Coordinator1.setpoint→PendulumController1.setpoint→
  PendulumController1.actuator→InvertedPendulum1.actua
  tor→InvertedPendulum1.status;
obligation user.command1.ConductCalibration en route
path1;
```

Figure 7 Predicate Satisfaction through Paths

The predecessors or successors are determined as follows:

(1) by considering all input (output) ports as predecessors (successors);
(2) by taking into account data flow between input ports and output ports in the source code, either by program slicing techniques [3], by explicit declarations from the user, or by deriving the flow from the dependency graphs of composite components;
(3) by the designation along an explicitly declared path representing a flow of a data stream through multiple components.

Paths are specified in component implementations. The addressing of predicates at the scope of component level is through the hierarchical association, i.e., component.port.predicate. For example, the Inverted Pendulum routinely requires a calibration procedure in order to get the correct sensor angle measurement. This requirement is recorded as an obligation on one of the user output ports, as shown in Figure 7.

Properties and Property Constraints. Although propositional predicates can capture some semantic assumptions, they capture relationships between predicates sharing identical predicate names, i.e., they do not model multiple values. A Pendulum Controller may have three speeds. A Coordinator may make the assumption that the speed is "not zero". If expressed in predicates, the analysis does not recognize that "not zero" can be satisfied by "fast". By introducing properties with values and constraints, we can model the example as:

```
property type speed = enum { zero, slow, fast };
property Speed-Capacity: speed = fast;
```

Notice that the user-defined enumeration of values is ordered, i.e., comparison operators can be applied to properties with such values.

A property constraint documents an assumption a component makes about the properties of other components. Inside constraint expressions, associated properties can be combined together by means of arithmetic operators, comparison operators, and boolean operators.

Properties are constrained in the following ways.

(1) The constraint can be stated in the specification and be associated with different properties within the same specification.
(2) The constraint can be stated in the specification and associated with the properties of connected components. For example, the first constraint in Figure 8 is associated with *Speed_Capacity* in the Coordinator. The Coordinator always requires the connected components, namely, the Pendulum Controllers, to have *Speed_Capacity* of at least *slow*.
(3) The constraint can be stated in the implementation and associated with multiple properties belonging to component instances in the implementation. For example, the second constraint in Figure 8 describes the coordination objective between the two inverted pendulums. If this constraint is not satisfied, the coordinators can potentially drive the two pendulums to be misaligned.
(4) The constraint can be stated in the implementation for a property that is declared for each of the components along a path, i.e., a cumulative property

(see Figure 8). For example, components may have a latency property. The sum of the latencies in the implementation must be less than the latency value declared in the component specification. Notice, that we are capturing end-to-end latency constraints about the application.

```
//constraint 1, inside component Coordinator
constraint setpoint->Speed_Capacity >=speed.slow;

//constraint 2, inside CoordinatedPendulums implementation
constraint  Coordinator1.Speed_Capacity ==
            Coordinator2..Speed_Capacity;

// cumulative property constraint
constraint sum(latency) < latency en route path1;
```

Figure 8 Property Constraint Examples

Properties can be used to describe key characteristics of control components without requiring a full set of control equations. Property values can be derived and reevaluated by control engineers.

Property Dependencies. In some cases, component properties depend on each other semantically, e.g., reflected in control equations, or are time-sensitive, i.e., depend on (are affected by) timing properties. We support modeling of such dependencies through property dependencies. For example, a Pendulum Controller has the property Speed_Capacity, which is affected by the rate of port set-point, as well as property Max_Step_Size. These dependency relationships are described in Figure 9. They reflect dependencies hidden in control equations.

```
property Stability_Range = 8 cm;

//property Speed_Capacity is defined in section 3.2.3;

dependency Stability_Range ↔Max_Step_Size;
dependency Speed_Capacity→ Max_Step_Size;
dependency Speed_Capacity→ setpoint.rate;
```

Figure 9 Property Dependencies in Pendulum Controller

The dependency is denoted as "→", which can be interpreted as "depends on" or "is a function of". That is, changing the right-hand side property potentially affects the left-hand side property. One property can be dependent on other properties. Properties may affect each other, expressed as"↔". In our example, if the property *Max_Step_Size* gets changed, the properties *Stability_Range* and *Speed_Capacity* are potentially affected. Control engineers can re-simulate the system model built in the simulation environment, and determine whether the *Stability_Range* and *Speed_Capacity* need to be changed.

Property dependencies can be located in component specifications to address the dependencies between multiple properties within the identical components, or inside component implementations to address the dependencies between properties associated with different components (including the composite components themselves). In contrast to property constraints, the property dependency evaluation does not contribute to the inconsistency exposure.

Timing Properties and Time-Sensitivity. In real-time systems, components represent schedulable units. They have a set of riming properties such as periods, worst case execution times, deadlines, and priorities. These properties are used in schedulability analysis, such as RMA [12]. Schedulability of a system is a constraint that is implicit in our modeling notation.

We have the following timing property relationships:

(1) between components sharing a resource, expressed implicitly;
(2) between port periods and the associated component's period, expressed implicitly. Any port period should be the multiples of the associated component period;
(3) time sensitivity of user-defined component properties, expressed by an explicit property dependency. In Figure 9, the *setpoint* port rate of the Pendulum Controller directly influences the property *Speed_Capacity*. The time-sensitivity records establish the linking between application semantics and timing/scheduling analysis.

4. Scope of Change

Impact analysis delineates the scope of change based on the dependency information associated with a change, i.e., the set of impacted components due to the change. Impact analysis makes use of consistency analysis discussed in Section 3 to reduce the impact scope estimation. The consistency maintaining benefits from impact analysis by evaluating the change scopes of different alternatives to fix the inconsistencies, and then determining the change candidate which leads to the minimum scope.

4.1. Types of Dependency Relationship

In this section, different types of relationship that we derive from model constructs reflect the dependencies to be considered in impact analysis. The relationships include:

(1) specification–implementation, abbreviated as **SIM**;
(2) specification–instance, abbreviated as **SIN**;
(3) component–hardware resource representing resource binding, abbreviated as **RB**;
(4) property definition–use, abbreviated as **PDU**. Properties are used in range/delta port specifications, property declarations and property constraints;
(5) component instance interaction relationship reflecting

port connections, abbreviated as **CI**;

(6) port binding, abbreviated as **PB**. A port in an implementation representing a port in the corresponding specification;

(7) explicit property relationship declared by the user, abbreviated as **EPR**;

(8) property dependencies reflected in property constraints, abbreviated as **DPC**. Two arbitrary properties involved in the identical constraint affect each other.

(9) property relationship built into the model, abbreviated as **PRM**. For example, varying the component period potentially affects the port periods, or vice versa, based on the pre-defined component period and the associated port period constraint relationship.

These types of dependencies form the basis for impact analysis.

The change impact can be determined in two ways: the *potential impact* and the *actual impact*. The potential impact identifies all components potentially affected if a change occurs in the model. This includes all components directly affected as well as those indirectly affected. In other words, we follow the transitive closure of all dependency relationships to be considered. The actual impact is to identify potential candidates in the model, reevaluate the relevant constraints, i.e., input/output constraints, predicates, and property constraints, in which all changed values are known, and terminate the change propagation if the constraints are not violated. Therefore, the constraints can become the propagation barriers. We will illustrate these two different types of change impact in the increasingly complex examples in Section 4.3.

4.3. Impact Analysis Scenarios

In this section, we examine several scenarios of anticipated and actual change to the application. We illustrate how the different types of dependency relationships described in the previous section are used to identify the potential and actual impact of a change on other parts of the application.

Suppose a change occurs to the measurement unit of port *actuator*, from voltage to ampere, inside the Pendulum Controller's specification. The change propagation is illustrated as follows.

(a) Following the specification-instance relation (SIM), *PendulumController1* and *PendulumController2* are obtained, whose corresponding ports have the respective measurement unit change;

(b) Following the component instance interaction relation (CI), all components directly connected to these changed ports, in particular, the ports which are part of connections are potentially affected. By conducting the reevaluation of input/output specification related checking, the actual affected ports can be determined by inspecting the resulted inconsistencies;

(c) Following the specification-implementation realation (SIM), the implementation is obtained. By further following the port binding relation (PB), the impact includes ports bound to the modified port. Similar to (b), the reevaluation leads to the actual impact.

The change propagation pattern is illustrated in Figure 10. The same pattern applies when coordinate system or the type of the data is changed. For a change to predicates, the change propagation differs slightly in that when evaluating the actual impact, the indirectly connected components are affected. A postcondition change can affect other components whose preconditions/obligations are satisfied by (depend on) this postcondition. A change to preconditions/obligations requires reevaluation of predicate satisfaction.

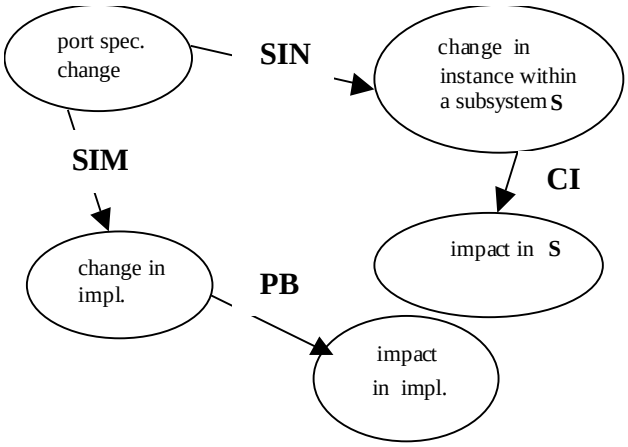

Figure 10 Port Change Propagation Pattern

We illustrate the change impact to properties next. Suppose a change occurs to user-defined property *Max_Step_Size* in the Pendulum Controller. We have the change propagation as follows.

(a) Following the property definition-use relation (PDU), we identify that the *Max_Step_Size* is used inside the *setpoint* port range/delta specification (see Figure 4), as well as the property dependencies (in Figure 9). The impact on the range/delta specification further propagates as discussed above. From the dependency declarations, it is found that the *Stability_Range* and *Speed_Capacity* are potentially affected. The property change may also impact property constraints. The potential impact associated with the constraints can be determined by the property dependency relation (DPC). From the actual change perspective, the property constraints can be re-evaluated, and if no violation occurs, no further propagation is necessary.

(b) Following the specification-instance relation (SIN), *PendulumController1* and *PendulumController2* are found in the Coordinated Pendulums system, whose *Max_Step_Size'*s get changed. Given these potentially changed properties, following the property definition-use relation (PDU), the set of relevant property dependencies and property constraints can be found. By evaluating the property dependencies and constraints as described in (a), the set of instances whose properties get affected are determined as the actual impact. Then following reversed specification-instance relations (SIN), the component specifications associated with the component instances, whose properties are impacted, are found. The further propagation follows (a).

(c) Following the specification-implementation relation (SIM), the implementation is obtained. Following the property definition-use relation (PDU), the associated property dependencies and constraints are found, and further analysis is identical to (b). In our example, assume constraint: *Max_Step_Size == 2.0 * Gain* is specified in the implementation in which *Gain* is a property. If the changed value of *Max_Step_Size* is known, the constraint is reevaluated, and the violation determines the impact on the *Gain.*

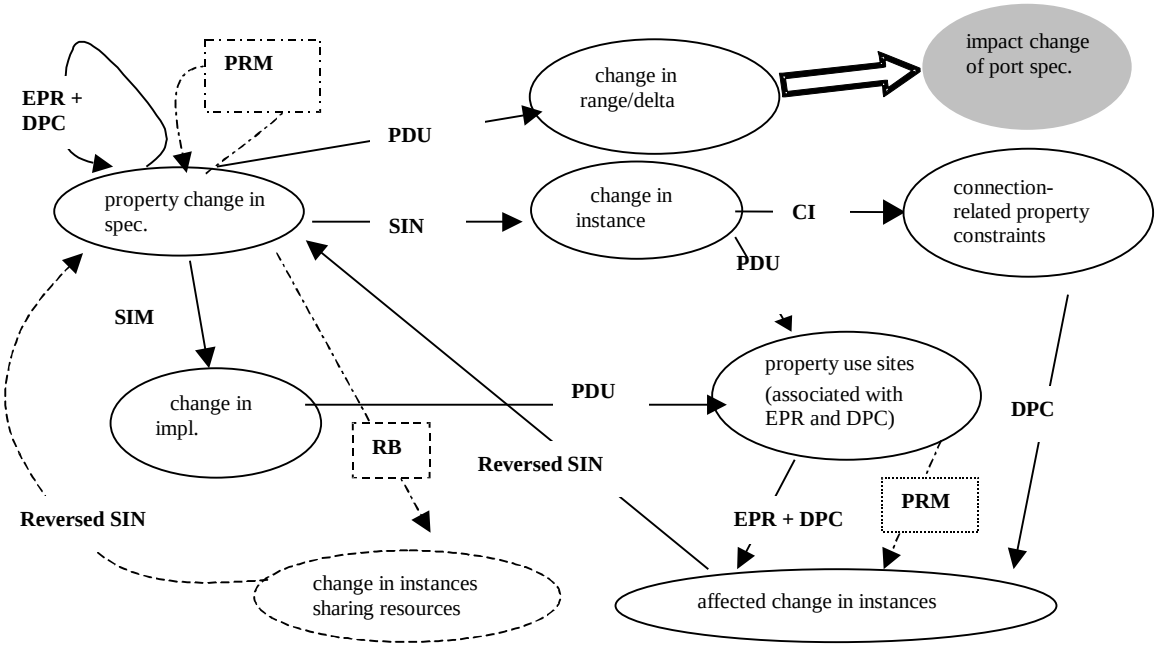

Figure 11 Property Change Propagation Pattern

The propagation pattern associated with properties is given in Figure 11. When the modified property is a timing property, the resource binding relation (RB) identifies the resource, whose resource allocation and schedule needs to be reevaluated. If the property is a pre-defined property (including timing properties), the model built-in relation (PRM) is followed as well. The PRM and RB related dependencies and the respective propagation are illustrated by the dashed lines and boxes in Figure 11.

5. Use of Constraint-Based Modeling

In the previous two sections we have presented an architectural modeling notation that allows application developers and maintainers to evolve a model of their system. They can incrementally enrich the model with information about application semantics, assumptions components make of each other, and time sensitive properties to address the interplay between timing properties of real-time process architectures and application semantics.

The notation and analysis capability is supported through an interactive tool. This tool offers a graphical representation of the architectural structure (as illustrated in Figure 2) that is natural to use without having to learn some of the textual syntax. Constraints, predicates, and properties are associated with components and ports as attributes. Again, the interactive graphical tool can offer dialog boxes to allow the user to enter and modify such information in a user-friendly way. Alternatively, developers can evolve these descriptions in textual form and process them through the analysis tool.

The consistency and impact analysis are implemented in the framework of Armani constraint system, which is available as part of the Acme architectural design environment [13]. The consistency checking rules and

constraints are described as a mix of Armani invariants and Java-based Acme evaluation functions. As such they act as a truth maintenance system [14] or a model checking capability [15]. As the user makes modifications to the model and the set of constraints, the analysis tool automatically reevaluates the constraints, identifies and reports inconsistencies and impacted components to the user. This allows the user to bring the application into a consistent state before releasing it into the operational environment.

Developers and maintainers can use such a modeling and analysis capability in several ways. Maintainers may have difficulty routinely updating a system due to hidden side effects. They may start by quickly creating a high level model of the system in terms of components and their interactions. This basic model may be augmented with semantic properties and assumptions about various system components as they are identified as root causes of hidden side effects. Port specifications can be refined with data stream characteristics and range constraints, and augmented with predicates as a simple way of recording assumptions and assertions. Maintainers may consult design documents and domain experts to gain insight into application semantics that are not encoded in the source code. In short, as maintainers are trying to make changes to a poorly documented system they record insight they gain in a notation that can be analyzed automatically. The more insights they record the more effective the analysis tool will be in identifying problems in the future. Impact analysis is an effective tool to minimize the number of system components that have to be revisited in order to complete a system change. In particular, given the different types of dependencies identified in Section 4, it is understandable why mistakes can easily be made when changes are propagated through manual methods.

During system development and evolution, design reviewers may use the modeling capability to create an analyzable model based on the information they may gleam from design documents. Again, the process of creating such a model of the application will provide insight into the application and may create awareness of potential problem spots, even before consistency analysis is deployed. Once assumptions and claims are added to the model as constraints, tool-based analysis can identify inconsistencies. Equipped with the model and analysis results, a reviewer can do an effective and thorough job in providing feedback.

Evolution and refinement of a system model based on architectural notations with constraint expressions like ours can also be made part of the development process. Such a model can be a useful tool to the developers themselves as they are integrating and debugging a system. The structural information of the architectural model can be the basis of automatic generation of integrating glue code - as demonstrated by MetaH [10]. It can also become a useful tool in assessing change requests. Proposed changes to the system can be examined from the perspective of the modeled system. The impact analysis capability can provide insight regarding the scope of change.

6. Summary

Actual systems in real-time control applications have semantic and time-sensitive dependencies not typically recorded in the source code implementation. Therefore, these dependencies have hidden side effects. As a result, latent errors are not discovered until integration testing and operation. In this paper, we have described the notation to capture these hidden dependencies and provided the capability to perform consistency analysis and impact analysis. These analyses identify side effects of changes during maintenance and evolution of an application that were previously hidden and not discovered until system operation.

In order to address semantic and time sensitive dependencies, the modeling and analysis capability discussed in this paper takes into consideration not only logical component interconnections, but also relevant semantic information, captured as input/output specifications, predicates, constraints and dependencies of component properties. Our system modeling and analysis approach is incremental. As errors are detected in the actual system, previously hidden dependencies that caused the error can be added to the model. These additional constraints will identify side effects and therefore reduce future errors due to these dependencies.

We have implemented a prototype of an interactive modeling and analysis tool supporting our notation. This tool offers a graphical representation of architectural structures that can be annotated with properties and constraints. As the user makes modifications to the model and the set of constraints, the analysis tool automatically reevaluates the constraints, identifies and reports inconsistencies and impacted components to the user. This allows the user to bring the application into a consistent state before releasing it into the operational environment.

7. Acknowledgement

Research supported by the US Defense Advanced Research Projects Agency, under contract F33615-97-C-1012.

References

[1] S. A. Bohner, R. S. Arnold, "An introduction to software change impact analysis," *Software Change Impact Analysis*, pp. 1-26, edited by S. A. Bohner and R. S. Arnold.

[2] M. Weiser, "Program slicing," IEEE Trans. on Software Engineering, vol. SE-10, no. 4, 1984, pp. 352-357, 1984.
[3] S. Horwitz, T. Reps, and D. Binkley, "Interprocedural slicing using dependence graphs," ACM Trans. Programming Languages and Systems, vol. 12, no. 1, pp. 26-60, 1990.
[4] G. Kaiser, P. Feiler, and S. Popovich, "Intelligent assistance for software development and maintenance," IEEE Software, pp. 40-49, May 1988.
[5] D. E. Perry, "the Inscape Environment," Proceedings of the 11th International Conference on Software Engineering, pp. 2-12, 1989.
[6] M. Shaw and D. Garlan, *Software Architecture: Perspectives on an Emerging Discipline*. Prentice Hall, 1996.
[7] R. Allen, D. Garlan, "Formalizing architectural connection," Sixteenth International Conference on Software Engineering, pp. 71-80, 1994.
[8] D. C. Luckham, L. M. Augustin, J. J. Kenny, J. Vera, D. Bryan, W. Mann, "Specification and analysis of system architecture using Rapide," IEEE Trans. 1995, SE-21, pp. 336-355.
[9] J. A. Stafoord, D. J. Richardson, and A. L. Wolf, "Chaining: a software architecture dependency analysis technique," Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado.
[10] P. Binns, S. Vestal, "Scheduling and communication in MetaH," Real-Time System Symposium, Raleigh-Durham, NC, Dec. 1993.
[11] D. E. Perry, "Software interconnection models," Proceedings of the 9th International Conference on Software Engineering, pp. 61-69, 1987.
[12] M. Klein, et. al., "A practitioner's handbook for real-time analysis: guide to rate monotonic analysis for real-time systems," Boston: Kluwer, 1993.
[13] R. T. Monroe, "Capturing software architecture design expertise with Armani," Technical Report CMU-CS-98-163, School of Computer Science, Carnegie Mellon University. www.cs.cmu.edu/~able/paper_abstracts/armani-lrm.html.
[14] J. Doyle, "A truth maintenance system," Artificial Intelligence, Vol. 12, No. 3, 1979, pp. 231-272.
[15] E. M. Clarke, O. Grumberg, and D. E. Long. "Model checking and abstraction." In Proceedings of the Nineteenth Annual ACM Symposium on Principles of Programming Languages, January 1992.