Consistency in Dynamic Reconfiguration

Peter Feiler, Jun Li
Software Engineering Institute/Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh, PA 15213

Abstract

This paper examines issues relating to the impact of change in real-time control applications. In particular, Simplex-based systems are being considered, a technology that supports dependable upgrade of systems in a fault tolerant manner through the concept of analytic redundancy [4]. Such systems provide flexibility to real-time systems for dynamic reconfiguration and dependable incremental and online upgrade. The paper focuses on offline analysis to determine inconsistencies in configurations and identify reconfiguration paths to recover to consistent configurations. The results are used by runtime configuration management to avoid such configurations. Identification of inconsistent configurations is improved through modeling of application semantics in the control domain and utilizing them in the analysis. The same analysis supports design time analysis of potential impact of changes.

1. Introduction

Computer-based control systems continue to be a growing trend. The flexibility of software provides an opportunity to allow these systems to be evolved, adapted, and tuned in a rapidly changing operating environment. However, this flexibility also brings liabilities in that such changes may have unintended side effects that can cause the system to fail. In particular, real-time systems are quite sensitive to changes in the timing behavior and the impact of change on the semantics of the application, i.e., control system semantics. Consequently, in managing the execution of a complex real-time system many dependencies among different components must be considered. While these dependencies can be easily understood for a small system, in larger systems they form complex networks. Manual methods used in practice are inadequate and error-prone. Thus, technology-based solutions have to be considered in supporting dependable upgrade of mission critical systems, i.e., component upgrade without degradation of the system operation despite residual errors.

One such solution is the Simplex technology [17]. Simplex supports runtime reconfiguration and component replacement in a fault tolerant manner. Components may exist in variants. Each variant executes as a separate process to provide protection from addressing faults. Scheduling analysis such as Generalized Rate Monotonic Analysis [8] and runtime monitoring of execution time limits protect from timing faults of variants. One variant is identified as the leader, i.e., its output is passed on as the output of the component. Leadership can be changed dynamically, i.e., a component can be reconfigured at runtime from one variant to another. In addition, variants can be replaced and new variants can be added online, i.e., while the system is operating.

The concept of an analytically redundant component (ARC) is used to provide protection from application semantic faults. Variants of a component are considered analytically redundant if they produce differing, but semantically acceptable results. This means that the output may be different, but the effect of the output on the controlled plant in a control system is within an intended operational region. One of the variants is a highly reliable controller variant with a well-known operational region, known as safety region. It can maintain stability of the controlled plant within this state space. Other variants may be given leadership, but their performance is monitored by checking the output and by checking of the resulting plant state against the safety region. If a fault is detected, leadership is changed to another variant, if necessary to the safety controller. Dependable system upgrades are supported by allowing new variants of components to be inserted and leadership given to them.

Report Documentation Page

Form Approved
OMB No. 0704-0188

Public reporting burden for the collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to a penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number.

1. REPORT DATE **OCT 1998**	2. REPORT TYPE	3. DATES COVERED **00-00-1998 to 00-00-1998**

4. TITLE AND SUBTITLE
Consistency in Dynamic Reconfiguration

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)
Carnegie Mellon University,Software Engineering Institute,Pittsburgh,PA,15213

8. PERFORMING ORGANIZATION REPORT NUMBER

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSOR/MONITOR'S ACRONYM(S)

11. SPONSOR/MONITOR'S REPORT NUMBER(S)

12. DISTRIBUTION/AVAILABILITY STATEMENT
Approved for public release; distribution unlimited

13. SUPPLEMENTARY NOTES

14. ABSTRACT
This paper examines issues relating to the impact of change in real-time control applications. In particular Simplex-based systems are being considered, a technology that supports dependable upgrade of systems in a fault tolerant manner through the concept of analytic redundancy [4]. Such systems provide flexibility to realtime systems for dynamic reconfiguration and dependable incremental and online upgrade. The paper focuses on offline analysis to determine inconsistencies in configurations and identify reconfiguration paths to recover to consistent configurations. The results are used by runtime configuration management to avoid such configurations. Identification of inconsistent configurations is improved through modeling of application semantics in the control domain and utilizing them in the analysis. The same analysis supports design time analysis of potential impact of changes.

15. SUBJECT TERMS

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19a. NAME OF RESPONSIBLE PERSON
a. REPORT **unclassified**	b. ABSTRACT **unclassified**	c. THIS PAGE **unclassified**	**Same as Report (SAR)**	**8**	

The basic Simplex fault tolerance approach is reactive in nature. Faults in individual components and inconsistencies between components are observed on a component basis and the system is reconfigured incrementally. A reconfiguration step may result in an inconsistent configuration, the effects of which may be observed by one of the components resulting in further reconfiguration. The system configuration consisting of all safety variants is assumed to be a consistent configuration, generating no further side effects.

In this paper we discuss how the Simplex fault tolerance capability can be augmented with proactive fault avoidance through detection of and recovery from inconsistent configurations. Our approach is to perform offline analysis to determine

- whether a given configuration is inconsistent, thus, should be avoided as a target configuration;
- what a desirable reconfiguration is in case of an observed fault or inconsistency;
- what the impact of a change is and how it can be reduced.

The results of the offline analysis are then used by runtime reconfiguration management support to avoid switching to configurations that are deemed inconsistent based on the model description.

The paper proceeds as follows. Section 2 describes a basic system modeling capability and introduces the concept of configuration consistency in terms of syntax, type, resource utilization, and semantics. Section 3 discusses inconsistency of configurations in the context of analytically redundant, multi-variant components and approaches to managing such inconsistencies. Section 4 focuses on capturing application semantics, resulting in recognition of additional inconsistencies. Section 5 discusses the impact of change and ways to reduce the scope of impact.

2. Consistency of System Configuration Models

A number of modeling approaches have been pursued in describing the configuration of systems. In the 70s and 80s emphasis has been placed on modeling source code configurations and supporting the system build process based in this system model [6, 21]. The models captured provision/use dependencies between components. The concept of well-formed models has been introduced by Habermann and Perry [7]. In the 80s configuration languages were developed as a means of modeling of runtime configurations, and attention was given to mechanisms for consistently performing runtime reconfiguration [9]. More recently, architectural description languages have received research attention [18]. Their focus is on modeling and analyzing systems as logical components, their interconnections, and constraints on components and connectors in terms of type and behavior. Their initial focus has been on modeling a single system configuration, though, recently the need for modeling more dynamic architectural characteristics has been recognized [2]. Some architectural languages, such as MetaH [22], specialize in modeling real-time applications - their toolsets analyzing schedulability and generating appropriate execution environments.

For modeling Simplex-based applications we build on the capabilities of such modeling languages. In such models systems are expressed as compositions of components and connectors. Components have ports to which connectors are attached. Ports can be directional (in, out). Ports can be the source or destination of more than one connector. The resulting system topology represents a dependency graph. The destination component of a connection is referred to as a direct successor of a (source) component. The source component of a connection is referred to as a direct predecessor of a (destination) component. A component can have multiple direct dependents, and it may be directly dependent on several other components. A change to a component may impact its direct dependents as well as their dependents, i.e., the transitive closure of this dependency relationship. The paths of these transitive closures are referred to as dependency chains in [20].

A given composition (configuration) is considered syntactically well-formed [14], if it is complete and consistent. A configuration is complete if all component ports are connected, and all connectors have a source and destination. A configuration is *syntactically consistent* if connectors are attached to component ports and the direction of the connection matches the direction of the ports. Components, connectors, and ports may be typed. A configuration is considered *type consistent* if it satisfies type restrictions, e.g., a connector of a certain type may only accept components of certain types as its source and destination and port types of the source and destination ports must match. Configurations that are syntactically consistent may be type inconsistent.

This concept of configuration consistency can be extended to include resource utilization and schedulability - critical for real-time applications. In particular, schedulability analysis based on particular scheduling approaches, e.g., Generalized Rate Monotonic Analysis (GRMA) [8] can determine whether a particular task configuration is schedulable, i.e., the configuration is consistent with respect to timing. MetaH, an architectural language for real-time applications [23], has the capability of modeling and analyzing multiple task configurations. Such analyses can be adapted to incorporate resource knowledge of ARCs.

Finally, configurations may be considered semantically inconsistent. Interface specifications often do not capture the application semantics, while components make use of such knowledge. An example is a higher-level control system component making assumptions about the responsiveness of the low-level controller. Such hidden dependencies can cause problems when systems change. In short, configurations are analyzed to identify inconsistencies - the richer the model the more inconsistencies can be detected.

In the next sections we will first address the notion of configuration inconsistency in the context of Simplex-based systems, and then examine how representing application semantics can improve our ability to detect inconsistent configurations.

3. Variant Configurations and Reconfiguration

In systems with ARCs, components have multiple variants with one variant (the leader) providing the output. In such a system we have *variant configurations*, i.e., the set of leader variants, and *execution configurations*, i.e., the set of variants executing at any one time. Consistency of each possible configurations can be analyzed statically. Static analysis determines which configurations are inconsistent. It also determines, which component variants must be changed to transition from one consistent configuration to another. This set is also known as changeset in traditional configuration management [5]. The system may not be able to perform this transition atomically, i.e., intermediate reconfiguration steps may result in temporary inconsistent configurations. In this section we first discuss the identification of inconsistent configurations in the context of variants, and then examine the concept of reconfiguration transaction.

An actual Simplex-based prototype system is used to illustrate some of the points in the paper. This system consists of two independent controlled devices that need to operate in a coordinated manner. In the lab prototype these devices are highly unstable inverted pendulums that transport a rod on their tip. The software system is illustrated in Figure 1. It consists of a device interface component and a controller component for each pendulum (PC1, PC2). The pendulum controller is responsible for balancing the pendulum. In addition, a coordinator component is responsible for keeping the pendulums aligned. The software runs on several networked processors. The pendulum controllers and the coordinator are ARCs, i.e., consist of a safety, a baseline, and a upgrade variant. The safety variant of the pendulum controller together with the monitored safety region ensures that the pendulum does not fall. The other two variants move the pendulum to desired positions (setpoints). The baseline and upgrade variants of the coordinator move the pendulums at speeds supported by the respective pendulum controller variant, while the safety variant handles misalignment of pendulums, recovering pendulum controllers, and system faults, such as network communication faults.

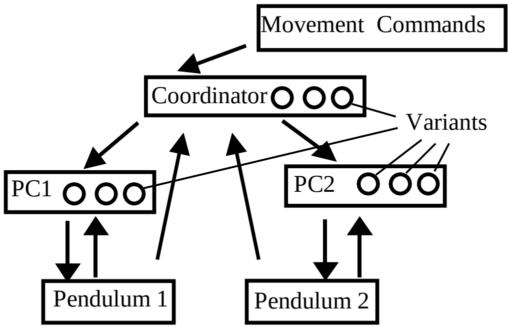

Figure 1 Coordinated Pendulum System

3.1. Inconsistent Variant Configurations

Static analysis can determine the set of configurations that are considered inconsistent. A configuration is inconsistent if the connection between at least one pair of components is considered inconsistent. Beladi [3] used this idea to determine inconsistencies in configurations of operating systems loaded with various patches based on a specified set of constraints between combinations of patch sets. Such constraints may be supplied by the developer based on empirical evidence, even though they may not know or be able to express the root cause of this inconsistency. Our consistency analysis can identify such configuration constraints by checking for syntactic, type, resource, and semantic inconsistency between connected component variants. In our example, the baseline and upgrade variants of the coordinator are inconsistent with the safety variant of the pendulum controller as they supply setpoints, which are ignored by the safety variant.

Variants may be considered compatible, i.e., interchangeable with respect to their interaction with other components. The concept of compatibility has been investigated and formalized previously [21, 15, 12, 14]. In our context two types of compatibility are of interest: strict compatibility, and upward compatibility. Two variants are considered strictly compatible if other components are only dependent on an interface specification that is satisfied by both variants. This means that a reconfiguration between those variants in either direction has no detectable side effects.

A variant is considered upward compatible with a second variant, if the first variant can replace the second and all specified dependencies by connected components continue to be satisfied, i.e., a reconfiguration from the first to the second variant has no side effects. This does not hold for the inverse reconfiguration operation. As with connection constraint sets, compatibility constraints may be supplied by the developer based on empirical evidence, or they may be derived from other specified information. Compatibility can be determined for each component with variants, and utilized in determining the set of inconsistent configurations.

Additional constraints may apply to transitions between the variants of a component. For example, a safety region violation may trigger the need for reconfiguration, and the safety variant is the only acceptable destination variant to recover from this violation. In case of a failure by a variant other than safety region violation it is acceptable to transition to a variant other than the safety variant. These reconfiguration constraints are monitored at runtime, but the possible transitions can be validated offline against the compatibility constraints.

3.2. Consistent Reconfiguration

Given the above analysis information we can determine whether a configuration is inconsistent, either a configuration requested by the operator of the system before it is carried out, or a configuration as a result of an ARC changing leadership due to an observed fault. Such a fault is considered hidden because it is observed by monitoring the safety region rather than detected through analysis of the model. Instead of waiting for other ARCs to observe the effects of such a inconsistent configuration and reconfiguring themselves, we can identify a desirable reconfiguration transaction, i.e., a set of reconfiguration steps by different ARCs that lead to a consistent configuration. To illustrate assume that all ARCs operate with the baseline variant leading and one pendulum controller ARC switches to the safety variant. Without proactive reconfiguration, misalignment is observed and the coordinator switches to its safety variant to attempt realignment. This misalignment fault can be avoided by recognizing that the configuration is inconsistent, i.e., the coordinator provides setpoints to both pendulum controllers, one of which ignores them as input. A desirable consistent configuration is for the coordinator to have the other pendulum follow the recovering pendulum.

The set of desirable reconfiguration transactions can be determined through static analysis and constrained by the statically known transition constraints between variants. Appropriate subsets of desirable reconfiguration transactions can be associated with each of the runtime determined (fault triggered) transitions. Each candidate transaction containing a reconfiguration step with runtime constraints will have to be checked at runtime whether it is eligible. Typically, transactions representing recovery from faults do not contain runtime constraints, while transactions intended to include new variants are constrained at runtime. If more than one desirable reconfiguration transaction remains, one is selected based on criteria such as smallest number of reconfiguration steps, or smallest loss in capability or performance by choosing the configuration with the largest number of high performance variants.

It may not be possible to perform a reconfiguration transaction as an atomic operation, i.e., at a single point in time. For example, different components may execute at different periods. A reconfiguration from the task configuration perspective requires that the tasks must be aligned with the common hyper-period before switching task sets - taken into consideration by MetaH in their realization of mode switching [23]. Fortunately, in the domain of control systems there is lag [19] in the system to tolerate temporarily inconsistent configurations, i.e., permits the incremental execution of reconfiguration steps within time bounds. Further fault triggered reconfigurations of analytically redundant components may occur during a reconfiguration transaction. Either that particular variant transition is already part of the reconfiguration transaction being executed, i.e., has no further impact, or a reconfiguration transaction that includes this transition as reconfiguration step has to be identified.

4. Semantics

Our goal is to make use of semantic information in the configuration model in order to identify semantically inconsistent configurations that are syntactically considered consistent. The more semantic information is provided in the model the better the analysis capability in identifying inconsistency. Our approach does not require complete semantic specification because the fault tolerance mechanisms of Simplex will catch faults not discovered through static analysis.

A number of ADLs have focused on capturing behavior properties (Rapide through POSETs [10], Wright with event pattern constraints on connectors [1]). Other notations express properties as predicate constraints. We are building on Perry's approach of using predicates in a light-weight form to determine semantic consistency of configurations (he refers to them as compositions) [11, 12, 13]. Assumptions about acceptable input to a component are specified as preconditions, constraints that the output of a component satisfies are modeled as postconditions, and assumptions the component makes about further output processing is referred to as obligations. The approach is considered light-weight because the focus is not on a verification of a component implementation against its specification, but on whether the predicates are satisfied in the configuration. For preconditions we are looking for matching postconditions of components earlier in the dependency graph, while for obligations we are looking for matching postconditions of successors in the dependency graph.

We will proceed by first illustrating how such predicates are used to model application semantics, and then discussing how these predicate-based models are analyzed to identify potential inconsistency in configurations.

4.1. Capturing Control Semantics

The application domain of control systems has two major categories of components: inner loop control, i.e., components processing continuous signal streams, and supervisory control, i.e., managing of various operational modes as result of discrete events in the system. In the inner loop control, the emphasis is on continuity, accuracy, and timeliness of data streams and on the data stream characteristics as a result of various processing steps. In supervisory control, the emphasis is on the coverage of the event space and reachability of different modes. The latter are typically modeled as state machines or petrinets and will not be discussed in this paper.

Some of the application semantics can be captured through the type mechanism of the interface specification. The representation type (int, real, etc), unit of measurement (meter, inch), and coordinate system can be mapped into user defined types. Many applications in the control system domain deal with multiple coordinate systems, e.g., avionics systems may use a earth-fixed coordinate as well as an aircraft body-axis coordinate system. In our example, the position of the pendulum is represented as integer in units of centimeters with the middle of the track being the reference point of a one-dimensional coordinate system.

Additional semantic constraints can be expressed as range constraints on the input and output values. A controller may require the setpoint, i.e., the desired state of the controlled device, it receives to be within certain limits for it to operate in a stable manner. The component may supply setpoint within a specified range. The supplier range has to be contained in the recipient range. In our example, range constraints include limits on device state such as cart position and pendulum angle as well as voltage values supplied to drive the motor of the device - reflecting physical constraints of the device. In case of the motor voltage, the motor device enforces the range constraint by clipping higher voltages to the acceptable maximum, in our example 4.95V.

In the control domain we are dealing with data streams and system events. These have properties such as an arrival rate, which reflects the sampling rate of a device, or the maximum allowable difference between two elements of the data stream. In our example, the pendulum device is sampled at a rate of 50Hz, while setpoints are supplied to the low-level controller at a rate of 5Hz. A setpoint is constrained relative to the previous setpoint value in the data stream, i.e., limit the delta between desired positions is to be within the stability region of the controller. Connections of ports with different sampling rates results in under- or over-sampling. Intentional over- or under-sampling may be specified with an input port, while mismatches in rates are considered inconsistent.

```
Variant baseline_controller is
   Input sensor: { pend_state: Device_State } every 20 ms;
   Input  setpoint: {target_pos: desired.pos } every 200 ms;
   Output  actuator: {m_volt: motor.voltage } every 20 ms;
   Pre target_pos in [±(Max_Position - Stability_Range)] delta
       ±Max_Step_Size;
   Post m_volt in [±4.95 ];
   Property Stability_Range = 10 cm;
   Property Max_Step_Size = 5 cm;
   Property Speed: slow;
End Variant;
```

Figure 2 A Pendulum Controller Variant

Predicates can also be used to model the abstracted control algorithm as state associated with a data stream, e.g., whether a signal stream has been filtered or amplified. Constraints on flow order, i.e., the order in which data is to be processed can be modeled as preconditions, postconditions, and obligations associated with each component (primary use of obligations in Perry's examples [11]).

In some cases it is more natural to associate a property with a component than with the data stream being processed by a component. For example, different controllers have different performance characteristics, e.g., responsiveness or smoothness of movement. The assumptions made by some components about properties of other components can be expressed as constraints on those properties.

Some of the components are time sensitive, i.e., they make assumptions about the execution behavior of the application. For example, many controllers have an estimator element that compensates for the time delay between the reading of a signal (state of the device) and the time a response is supplied (to the device in our example, or as a displayed symbol to an operator, e.g., pilot). We can model this as a dependency between the implementation of a component and properties of the input or properties of other components.

4.2. Semantic Analysis

Configurations are considered semantically inconsistent if preconditions and obligations cannot be satisfied by (matched up with) postconditions by following the dependency graph. In this section we first focus on how the value of a predicate is determined through propagation through the dependency graph, and discuss what are semantically acceptable matches of interface specifications.

We apply Perry's concept of propagation logic in form of propositional calculus [13] to concurrent applications. Postconditions are assumed to be true, i.e., the developer asserts that the implementation satisfied the postconditions. The possible values for the precondition and obligation predicates are summarized in table 1.

Value	**Interpretation**
Unknown (U)	Nothing is known about the predicate.
True (T)	The predicate holds true.
False (F)	The predicate holds false.
Possible (P)	Known to be true or false along at least one path and unknown along at least one other path.

Table 1 Propositional Values

Initially it is unknown whether preconditions and obligations of a component are satisfied. For the set of preconditions the graph is traversed in reverse direction, while for the set of obligations it is traversed in forward direction. For each encountered component it is determined whether its postcondition list contains the same predicate or its negation. A predicate match indicates that the precondition is satisfied. The negation indicates that it cannot be satisfied and the search can be terminated. Perry refers to the latter as precondition ceiling and obligation floor [13]. The dependency graph may be a cyclic directed graph. When encountering a cycle in the traversal of the graph in order to find predicate matches, the cycle has to be traversed only once.

Pre_c / Obl_c + $Post_e$		Pre_c or Obl_c		
		U	T	F
$Post_e$	U	U	T	F
	T	T	T	F
	F	F	T	F

Table 2 Predicate Match Operator

The set of preconditions and obligations can be viewed as a vector Pre_c or Obl_c. The postconditions of the encountered components can be viewed as vectors ($Post_e$)of the same size with values true (T) - postcondition holds, false (F) - negation holds, unknown (U) - predicate not in postcondition set. We can introduce two operators on those vectors. The operator "+" represents matching of Preconditions/Obligations with Postconditions of encountered components. Table 2 defines how the operator applies to each element of the vectors. The operator is not commutative.

Components may have multiple direct predecessors or successors. In this case all paths have to be considered and the results merged. The merge operator "Π" on Pre_c/Obl_c is introduced for this reason. Table 3 defines its value mappings. The propositional value *possible* is indicated as P_T(at least one True), P_F(at least one False), $P_{T/F}$(at least one True and False) This operator is commutative. This merge operator can be interpreted in two ways: all paths must satisfy the predicate, or at least one path must satisfy it. In the former case all values U, P_T, P_F, and $P_{T/F}$ are interpreted as F. In the latter case P_T and $P_{T/F}$ are interpreted as T, while U and P_F are interpreted as F. Typically the first interpretation of merge is used, thus, is the default interpretation.

$V_1 \Pi V_2$		V_2					
		U	T	F	P_T	P_F	$P_{T/F}$
V_1	U	U	P_T	P_F	P_T	P_F	$P_{T/F}$
	T	P_T	T	$P_{T/F}$	P_T	$P_{T/F}$	$P_{T/F}$
	F	P_F	$P_{T/F}$	F	$P_{T/F}$	P_F	$P_{T/F}$
	P_T	P_T	P_T	$P_{T/F}$	P_T	$P_{T/F}$	$P_{T/F}$
	P_F	P_F	$P_{T/F}$	P_F	$P_{T/F}$	P_F	$P_{T/F}$
	$P_{T/F}$	$P_{T/F}$	$P_{T/F}$	$P_{T/F}$	$P_{T/F}$	$P_{T/F}$	$P_{T/F}$

Table 3 Merge Operator

In its simplest form the transitive closure of dependencies in a dependency graph takes into consideration all outgoing connections (connections through input ports in reverse traversal and output port connections in forward traversal). This graph can be reduced by taking into account the data flow within a component between input ports and output ports. A data object in the input port may also be specified in an output port, or a data object in an output port is affected by it. In the latter case, the developer may have to explicitly provide such a dependency, or it can be derived by utilizing program slicing techniques.

Predicates are matched as follows to determine whether a postcondition satisfies a precondition or obligation. It is assumed that the types of the output and input ports match. The input port may use only a subset of the data elements provided in a message, but this can indicate a potential problem if the supplier makes assumptions about certain data elements being processed, which can be captured with obligations. For each data element with a range specification it is assumed that the set of values supplied are within the range acceptable to the recipient, i.e., range(out) ⊆ range(in). For predicates on properties matching of property value is sufficient.

Compatibility can be refined as follows. Elem(port) indicates the set of data elements required or provided through ports. Variant B is upward compatible with variant A, if

$elem(A.in) \supseteq elem(B.in) \wedge range(A.in) \subseteq range(B.in)$

and

$elem(A.out) \subseteq elem(B.out) \wedge range(A.out) \supseteq range(B.out)$

Variants are strictly compatible, if they satisfy a common interface C as follows:

$elem(A/B.in) \subseteq elem(C.in) \wedge range(A/B.in) \supseteq range(C.in)$

and

$elem(A/B.out) \supseteq elem(C.out) \wedge range(A/B.out) \subseteq range(C.out)$

A configuration is considered consistent if all predicates are satisfied, i.e., for all preconditions and obligations of all components they evaluate to true. A system is considered well-formed if it is complete and consistent. Expected inputs must always be supplied, but an output from one component may not be consumed by any component. Such an incomplete configuration may be acceptable, but can also represent a potential problem. In our example, the pendulum controller may be supplied with setpoints for all variants while the safety variant ignores them. If the supplier of setpoints assumes that the setpoint will have been reached when the next setpoint is supplied, i.e., operates without feedback, an unstable situation may occur and have to be recovered at runtime.

For systems with ARCs we can not only check the consistency of the logical configuration, i.e., variant configuration, but also the execution configuration. For an execution configuration to be consistent, its respective leader variant configuration must be consistent, and at a minimum any additional input necessary for fault monitoring must be available. Other variants may only execute if their inputs are satisfied as well and they make no assumptions about their output actually affecting the controlled device, i.e., they do not keep state specific to their generated output. Thus, static analysis can determine which variants can execute concurrently at any one time.

Components may be composed into higher-level components with their own interfaces. We follow Perry's example [11] to analyze such hierarchies. Propagation of preconditions and obligations proceeds to the enclosing component, determines whether any unmatched predicates are listed in the interface specification of that component, and terminates. The preconditions and obligations of the enclosing component are separately propagated to be matched.

5. Impact Analysis and Reduction

The algorithm is incremental in that it can be performed on one component at a time. Therefore, it is amenable to be used in an interactive tool, where the developer can make incremental changes to the system model and the tool provides immediate feedback about the consequences. A change to a precondition or obligation is propagated as discussed earlier. A change to a postcondition needs to be propagated to reevaluate all components in the transitive closure, whose satisfaction of a precondition or obligation depended on this predicate.

The set of components to be considered as impacted based on this dependency graph can be reduced in a number of ways. First, if the change can be identified as affecting selected ports, only the dependency relations emanating from those ports need to be considered. Second, identification of compatibility between component variants with respect to a dependency relationship eliminates further propagation of impact. Third, identification of a dependency mapping between input ports and output ports of components reduces the number of affected indirect dependents. Such a mapping can be provided by the developer or can be derived from the implementation through program slicing technology (see section 4.2). Fourth, component properties can be subjected to sensitivity analysis. For example, a set of tasks may be schedulable within a range of execution times [24], or a controller may maintain stability within a range of sampling rates [16]. Finally, for certain properties, such as those related to resource utilization, a certain value may be specified as allocated to a component. Only changes in implementations that exceed these values need to be propagated.

Changes in resource utilization and timing properties of one component may impact the schedulability of otherwise unrelated components that share the same resource. Changes in timing properties may also impact the application semantics of a component, i.e., those semantic properties of application components that are time sensitive, such as the sampling rate of a controller. This creates property dependencies that must also be considered in change impact analysis. Some timing properties specify worst case allocations, e.g., execution time. Only if changes in actual timing properties exceed the specified ones or are drastically less, is a change necessary or desirable and its impact needs to be considered. Similarly, a change may affect utilization demands for a shared resource, e.g., change in code size or change in the amount of data to be communicated. Again, specified allocations reduce the impact of change if the actual implementation values stay within the specified bounds.

6. Summary

In this paper we have examined issues relating to the impact of change in real-time control application that are software-intensive systems. We have done so in the context of systems that are based on Simplex, a technology solution that supports dependable upgrade of systems in a fault tolerant manner through the concept of analytic redundancy. In doing so it provides flexibility to real-time systems for dynamic reconfiguration of systems to adapt to changes in the environment or unexpected behavior of system components and to support incremental and online upgrade. Several prototypes of Simplex-based applications are in operation.

We have focused on improving the capabilities of Simplex-based systems through proactive fault avoidance. Through offline analysis of system models, inconsistent configurations and transactions to consistent configurations are identified. Based on these results, a runtime configuration manager can avoid configurations considered inconsistent according to the model. Similarly, potential impact of changes to component can be identified at design time and developers supported in resolving them.

The modeling capability discussed here combines modeling and analysis ideas from several areas: system build models, configuration management, and architectural modeling. Emphasis has been placed on capturing application semantics through the use of predicates. Important to real-time applications, scheduling and resource utilization considerations have been taken into account.

7. Acknowledgement

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

8. References

[1] R. Allen, D. Garlan, “Formalizing architectural connection,” Proceedings of the Sixteenth International Conference on Software Engineering, 1994, pp. 71-80.

[2] R. Allen, R. Douence, D. Garlan, “Specifying dynamism in software architecture,” submitted for publication, Sept. 1997.

[3] L. A. Beladi, P. M. Merlin, "Evolving Parts and Relations - A Model of System Families," Program Evolution, Academic Press, 1985, pp. 221-236.

[4] M. Bodson, J. P. Lehoczky, R. Rajkumar, Lui Sha, M. Smith, J. Stephan, “Software fault-tolerance for control of responsive systems,” Proceedings of the Third International Workshop on Responsive Computer Systems, pp. 133-141, 1993.

[5] P. H. Feiler, "Configuration Management Models in Commercial Environments," Software Engineering Institute SEI-91-TR-13, July 1991.

[6] S. I. Feldman, "Make - A Program for Maintaining Computer Programs," Software - Practice and Experience, 9(3), Mar. 1979, 255-265.

[7] A. N. Habermann, D. E. Perry, "Well Formed System Composition," Carnegie Mellon University, Technical Report CMU-CS-80-117. March 1980.

[8] M. Klein, et.al., "A Practitioner's Handbook for Real-time Analysis: Guide to Rate Monotonic Analysis for Real-Time Systems". Boston: Kluwer, 1993.

[9] J. Kramer, J. Magee, “Dynamic configuration for distributed systems,” IEEE Trans. Software Eng., vol. 11, pp. 424-436, 1985.

[10] D. C. Luckham, L. M. Augustin, J. J. Kenney, J. Vera, D. Bryan, W. Mann, “Specification and analysis of system architecture using Rapide,” IEEE Trans. Soft. Eng., Vol. 21, 1995, pp. 336-355.

[11] D. E. Perry, “Software Interconnection models,” Proceedings of the 9th International Conference on Software Engineering, pp. 61-69, 1987.

[12] D. E. Perry, "Version Control in the Inscape Environment,” Proceedings of the 9th International Conference on Software Engineering, pp. 142-149, 1987.

[13] D. E. Perry, ”The logic of propagation in the Inscape Environment,” ACM SIGSOFT ’89 Third Symposium on Software Testing, Analysis and Verification, pp. 114-21, 1989.

[14] D. E. Perry, “System compositions and shared dependencies,” Software Configuration Management, ICSE'96 SCM-6 Workshop, pp. 139-153, 1996.

[15] E. Ploedereder, A. Fergany, "A Configuration management Assistant," Proceedings of the Second International Workshop on Version and Configuration Control, Oct. 1989, ACM Press.

[16] D. Seto, J. P. Lehoczky, L.Sha, K. G. Shin, “On task schedulability in real-time control systems,” Proceedings 17th IEEE Real-Time Systems Symposium, 1996, pp. 13-21.

[17] L. Sha, R. Rajkumar, M. Gagliardi, "Evolving Dependable Real-Time Systems," Proceedings of the 1996 IEEE Aerospace Applications Conference. Aspen, CO, Feb., 1996. New York, NY: IEEE Computer Society Press, 1996. Also published in: “Component-Based Software Engineering” Selected Papers from the Software Engineering Institute. Alan Brown Ed. IEEE Computer Society Press 1996. ISBN 0-8186-7718-X.

[18] M. Shaw, D. Garlan, Software architecture: perspectives on an emerging discipline, Upper Saddle River, N. J. : Prentice Hall, 1996.

[19] K. G. Shin, H. Kim, “Derivation and application of hard deadlines for real-time control systems,” IEEE Trans. on Systems, Man, and Cybernetics, vol. 22, No.6, 1992, pp. 1403-1412.

[20] J. A. Stafford, D. J. Richardson, and A. L. Wolf, “Chaining: a software architecture dependence analysis technique,” Technical Report CU-CS-845-97, Department of Computer Science, University of Colorado.

[21] W. Tichy, "A Data Model for Programming Support Environments and Its Application," Automated Tools for Information System Design. North-Holland Publishing Company, 1982, 31-48.

[22] S. Vestal, P. Binns, "Scheduling and communication in MetaH," Proceedings on Real-Time Systems Symposium, IEEE, 1993, 194-200.

[23] S. Vestal, "Mode Changes in a Real-Time Architecture Description Language," Second International Workshop on Configurable Distributed Systems, March 1994.

[24] S. Vestal, "Fixed-Priority Sensitivity Analysis for Linear Compute Time Models," IEEE Transactions on Software Engineering, Vol. 20, No. 4, 1994, 308-317.