5th International Workshop on Model-based Methodologies for Pervasive and Embedded Software (MOMPES 2008)

Architectural Concurrency Equivalence with Chaotic Models

Dionisio de Niz
dionisio@sei.cmu.edu
Software Engineering Institute
Carnegie Mellon University

Abstract

During its lifetime, embedded systems go through multiple changes to their runtime architecture. That is, threads, processes, and processor are added or removed to/from the software and hardware. These additions can have multiple motivations such as adding tolerance to failures or changes in the hardware architecture for new releases of the system. During these modifications, one of the big challenges is ensuring that no new error is introduced. This verification tends to be difficult given that modifying the concurrency structure of an application has multiple side effects difficult to discover. In this paper we propose a model-based technique to compare the concurrency structures of two architectural models. This exhaustive comparison is based on the semantics of AADL, an architecture description language, and its model in Alloy, a formal verification language. This verification guides the designer to fully define the desired behaviors as well as the side effects that can be tolerated. We demonstrate the use of the modeling with a simple model from the automotive industry.

1. Introduction

Imperative languages, such as C, have popularized a comfortable sequential execution model where instructions are executed in the order they appear in the program. This model allows the programmer to focus on the instruction he is currently writing, which represents the current point of execution, and its relationship with its previous and next instructions. This model, along with the bundling of instructions into functions to build a functional hierarchy, has proven to be of paramount importance to conquer today's software complexity. However, this sequential execution model is broken when concurrency needs to be introduced and multiple execution points can exist at any given time. Concurrency can be introduced for different reasons. On the one hand, it can be introduce to improve some runtime quality attribute such as throughput or reliability. On the other hand, it can be a change of system architecture, e.g., moving from a centralized window-lifting computer in a car to a net of computers to lift each of the windows individually (to balance workload and reduce wiring). The challenge, when moving from a sequential execution model to a concurrent one, is to ensure that the implicit assumptions made with the sequential model still hold in the new concurrency model.

In this paper we present our approach to verify the equivalence of the architectural execution structure of two systems. In this approach we use one system as the base and verify that the other system, identified as the variant, provides an equivalent model of execution. If the verification fails, our approach provides a counter-example execution that the designer can use to correct his architecture or annotate the model to make explicit the assumption that invalidates the counter example. In our approach we start assuming the worst-case behavior of the hardware (chaotic) evaluating its effect on the software. Order is then introduced into this chaotic behavior as the designer introduces more information to the model.

1.1 Related Work

The original idea of analyzing a system as a chaotic model was provided in [10]. In this paper, the authors apply a combination of model checking and optimization techniques to safety analysis. Their purpose was to find an optimal solution combining the probability of failures and the potential damage caused by each failure.

Other research projects have also approached the analysis of problems introduced by concurrency. In [4] Erdogmus presents a scheme to verify concurrent systems driven by their architecture. In this work the author focuses on the construction of proof obligations and the consistent decomposition of the modules in order to preserve the proof obligations. However, his model is designed to be used through human manipulation and it does not provide indication of how to use these obligations to express commonly used system properties.

From the model checking arena, we can see at least two

0-7695-3104-0/08 $25.00 © 2008 IEEE
DOI 10.1109/MOMPES.2008.10

IEEE computer society

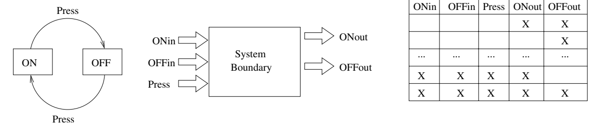
Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

tracks. On one hand there are plenty of languages that allows to model concurrency such as SMV [9] and SPIN [6]. However, model checking still has the problem of state space explosion. To deal with more practical verifications, other efforts have been made to approach industrial size projects. In [3], Chaki et al. present a scheme to verify concurrent programs written in C. In this work the authors combine two abstracting techniques to incrementally increase the granularity of the specification until they prove or refute the specification. In contrast, in our scheme we focus on the architectural module and the non-functional concurrency properties, avoiding the modeling of any functional computation. This way allows us to focus on the introduction of faults when comparing an assumed correct system with a variant with modified concurrency structure. Perhaps closer to our work is [5] but their focus is on simulation and not comparison of models.

Other works, such as [2], [11], [8], have focused on the generation of skeletons or the synchronization mechanisms from formal models in temporal logic. Even though we share with these authors the synchronization concerns, their focus is based on a full definition of the system. As a result, they required a model of the full combined behaviors of all the parallel processes. In our case, we limit ourselves to a higher level abstraction and do not cover the functional interactions.

2. Problem

To study the problems introduced by concurrency we focus on two dimensions: execution sequencing and reliability. The former refers to the variations to the execution sequence that was not considered in the original design of the software. The latter, refers to the fact that introducing additional computers and networks can introduce new failure modes. That is, in a single computer system, if the computer fails, then the whole system fails, hence when no fault-tolerance is required nothing needs to be done and the system can be considered to be stopped. On the other hand, if the computation is distributed among multiple computers and networks, if one of them fails it may be the case that an incorrect action is taken instead of stopping the system. For instance, if an ABS system is distributed into multiple computers (one per wheel) one of the messages to release the brake to one of the computers could be lost in the network causing an incorrect braking pattern, e.g. releasing the brake in three wheels with a fourth wheel still braking. Note that these partial failures can have far more extreme consequences than a total failure that leads to a stop of the functionality (all the wheels braking).



ONin	OFFin	Press	ONout	OFFout
			X	X
				X
...	...	...	...	...
X	X	X	X	
X	X	X	X	X

Figure 1. Chaotic Systems

3. General Approach

Our approach to explore incompatible behaviors is based on modeling a chaotic system as defined in [10]. In this paper, the authors define a chaotic system as a system that receives as input the user-provided inputs and the state of the system and as output the new state and the outputs to the user. A simple chaotic model of a switch can be seen in Figure 1.

Figure 1 depicts the chaotic model of a toggle switch. You can see in the middle of the figure the system defined with three inputs: the current state of the system as two Boolean variables ON_{in} and OFF_{in}, and the input from the user as the Boolean variable $Press$. As output we have the new state of the system with two Boolean variables ON_{out} and OFF_{out}. In the right side of the figure we have a table that lists all the possible combination of mappings from inputs to outputs. This table lists all the possible behaviors and represents the behavior of a chaotic system because there is no order in them. To this chaos, order is added by discarding mapping (rows) that are impossible to have. For instance it may not be possible to have both ON_{out} and OFF_{out} on together. After discarding all impossible behaviors we are left with two classes of behaviors: the ones that describe the behavior we want, and the ones that constitute faulty behaviors that need to be eliminated.

3.1 Non-Functional Chaotic Systems

Since the focus of our work is identifying discrepancies in the executions due to change in the concurrency structure of a system we need to redefine the chaotic model to capture non-functional behaviors. In this case we focus on the stream of instructions that need to be executed and two main disturbances to such stream: failure and reorder.

The instruction stream of our chaotic model is abstracted to an architectural level as a stream of jobs. A job represents the execution of a function. The sequence in which the calls are made constitutes the required sequence of the jobs. Jobs that belong to the same function are grouped into tasks. A task then constitutes the sequence of all the jobs that correspond to the calls to a function over time[1].

[1]We assume the rate-monotonic periodic task model.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

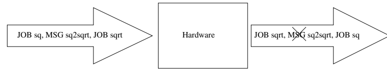


Figure 2. Non-Functional Chaotic Systems

INPUT			OUTPUT		
job_{sq}	$msg_{sq2sqrt}$	job_{sqrt}	job_{sqrt}	~~$msg_{sq2sqrt}$~~	job_{sq}
job_{sq}	$msg_{sq2sqrt}$	job_{sqrt}	$msg_{sq2sqrt}$	job_{sqrt}	job_{sq}
job_{sq}	$msg_{sq2sqrt}$	job_{sqrt}	job_{sq}	$msg_{sq2sqrt}$	job_{sqrt}
...	...	...	...	...	...

Table 1. Table of Behaviors

Data communicated between functions is modeled as messages between them. They represent the same "execution of the communication" as a job represents the execution of a function. All the messages that represent the different execution of the same communication are grouped into message streams. As a result, sequences of function calls that communicate among each other are modeled as chains of jobs and messages. For instance, consider a program that calculates the absolute value of a number. This program first calls the function `square()` over some input from the user and then passes the result to a second function called `square_root()`. In our model they are represented as two tasks, say `task_sq` and `task_sqrt` and one message stream, say $msgstream_{sq2sqrt}$. In the end, the collection of interconnected tasks and message streams forms a software graph that represents its execution structure.

Both the tasks and the message streams can be considered as arrays indexed by time instances. Each time instance represents a time period in which the integrity of the execution needs to be verified. For instance, in our absolute calculation program we need to verify that the jobs from $task_{sq}$ and $task_{sqrt}$ and the message from $msgstream_{sq2sqrt}$ execute in the correct order and do not fail. If we consider this time instances just as each of the times we execute the program (i.e., 1 for the first time we execute it, 2 for the second, and so on) then we need to check that $task_{sq}[1]$ executes first, then $msgstream_{sq2sqrt}[1]$ and finally $task_{sqrt}[1]$ (that can be represented as a sequence like $\{task_{sq}[1], msgstream_{sq2sqrt}[1], task_{sqrt}[1]\}$). Obviously we not only need to do this verification for the first time instance but for all possible time instances.

Disturbances to our non-functional execution model include reordering of the execution, e.g. $\{task_{sqrt}[1], task_{sq}[1], msgstream_{sq2sqrt}[1]\}$ and failure of jobs, e.g. $\{task_{sq}[1], task_{sqrt}[1]\}$ (where $msgstream_{sq2sqrt}[1]$ is not present because it failed). A graphical depiction of this concept is presented in Figure 2.

In Figure 2 we can see as input to the hardware a sequence of a job to the `square()` function (job_{sq}), a message for the message between the `square()` and the `square_root()` ($msg_{sq2sqrt}$) and the job for the `square_root()` (job_{sqrt}) all of them from the same time instance. As output from the hardware we can see examples of both types of disturbances, a reordering (job_{sqrt} before job_{sq}) and a failure of the message (represented with a cross over $msg_{sq2sqrt}$).

The model of Figure 2 can be translated into a tabular form where all the behaviors are described. A partial table for the system in Figure 2 can be seen in Table 1.

Table 1 shows a partial enumeration of the different execution possibilities from a chaotic model. Each of these rows is a mapping from the input software graph consisting of a flow of jobs to be executed to an output of the flow of jobs executed in the order in which they were executed and the jobs that were lost due to the chaotic behavior of the hardware. Each of these mappings is considered a potential behavior.

3.2 Discarding Behaviors

As with functional chaotic models, key to performing behavioral analysis on our chaotic model is the discarding of behaviors. We have three types of discarding strategies: impossible behaviors, executing platform properties, and software tolerance.

3.2.1 Discarding Impossible Behaviors

Impossible behaviors are discovered by analyzing the causality of the software graph. For instance in Table 1, the second row can be discarded based on our knowledge that $msg_{sq2sqrt}$ is produced by job_{sq} and, hence, cannot appear before it.

3.2.2 Discarding Behaviors Based on Executing Architecture Properties

Given that we are exploring the discrepancy of behaviors due to changes in the architecture, this architecture will constrain, i.e. reduce, these behaviors. There are two origins for these constraints. The first origin is the execution semantics inherent to the threads and processors. For instance, we know that in a single processor only one thread can be executing at any given time. The second origin of constraints is the properties we add to our system. For instance, if we use fixed priority scheduling in a processor, then we know that in the absence of locks the highest priority thread that is ready to run will be always running.

3.2.3 Discarding Behaviors Based on Software Tolerance

Some of the behavior discrepancies may be irrelevant. For instance in a stability control system we may want to inform

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

the driver that the system is operating (trying to stabilize the vehicle). This is usually done by turning on an indicator on the dashboard. However, whether such an indicator turns on one millisecond after or before the system starts to work is irrelevant. In this case it is useful to indicate that the case is not relevant and the software can tolerate the reordering of this particular execution.

4 Modeling for Automatic Analysis

Analyzing the discrepancies between models manually is impractical. Hence we have combined two languages and integrated their tools together to build an automatic analysis tool. To model the architectural constraints and the hardware we use AADL [1]. We also use AADL as the front end with which the user interacts. To evaluate the potential behaviors and behavior discrepancies we use Alloy [7]. We first discuss the Alloy model and abstractions and then we discuss the matching of this semantics with AADL. Next we discuss the special annotations for AADL models to allow the designer to restrict the behaviors.

4.1 An Alloy Non-Functional Chaotic Model

To be able to perform an automatic analysis we choose Alloy to encode the semantics of the execution structure of AADL models. Alloy is a language and tool for formal analysis that is classified as a “model finder.” Alloy uses relational logic to describe the system in terms of sets and relationships between sets. Facts and assertions over these relationships (set are considered singleton relationships) are expressed in the language and the model finder algorithm adds elements to the sets to try to find a counter example that invalidate the assertions. For instance, if we define two sets, $Book$ and $Reader$, and a relationship called $Reading$ that relates a reader to a book we could express the assertion that a book cannot be read by two readers. The model finder algorithm will then create an element of the set $Book$, say $book_1$, and two elements of the set $Reader$, say $reader_1$ and $reader_2$, and create two relationships $Reading$ from each of the reader elements to the single book element, i.e., $reading_1(reader_1, book_1)$ and $reading_2(reader_2, book_1)$ proving that the assertion is false. Facts can be added to constraint the system, e.g., to ensure that two reading relationships cannot exist with the same book in it.

4.1.1 Modeling Execution in Alloy

Alloy offers ordering relationships to be able to model sequences in Alloy. Sequences are modeled with the first, previous, next, and last relationships between elements of the same set. This construct allows us to do two things. On the one hand, it allows us to get the first and last elements of the sequence. And on the other hand, given an element of the sequence we can get the next or previous element in the sequence. One way to model execution in Alloy is to define the different states the system can be in over time. This time can be recorded as ticks in a sequence. For instance, if we add time to our book and reader system we could add time to the reading relationship. To do this we need to define a set ordered in a sequence say $Tick$. Now it is possible to define who is reading a book at each time instance. For example, the relationships $reading_1(tick_1, reader_1, book_1)$ and $reading_2(tick_2, reader_2, book_1)$ express that at time $tick_1$ $reader_1$ is reading $book_1$ and at time $tick_2$ the $reader_2$ is reading the same $book_1$. Restrictions in the evolution of the system can then be modeled as restrictions between one tick and the next.

Listing 1 Definitions of Sets in Alloy

```
sig Book{}
sig Reader{}
```

Listing 2 Relationships as Fields

```
sig Book{}
sig Reader{ reading: Book }
```

4.1.2 Basic Alloy Syntax

Alloy provides an object-oriented style for its syntax. Its sets are defined as signatures with the keyword sig. You can see the definition of the sets $Book$ and $Reader$ in Listing 1.

Relationships, on the other hand, can be expressed as fields inside a signature. For instance, to define the relationship $Reading$ that relates a $Reader$ to a $Book$, we can define a field inside $Reader$ of the type $Book$. Listing 2 extends Listing 1 to depict this relationship.

In addition, relationships between more than two elements can be used in the form of indexing, where one element acts as an index in the relationship of the other two. For example, if we add time to the reading relationship through a tick we would get a model like the one shown in Listing 3.

The reading relationship in Listing 3 expresses that a $Reader$ has a relationship with $Book$ that is indexed by $Tick$. In other words, $Reader$ may have different relation-

Listing 3 Introducing Time to Index Relationships

```
sig Tick{}
sig Book{}
sig Reader{ reading: Tick -> one Book }
```


Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

Listing 4 Fact for Temporal Restrictions

```
fact {
 all r:Reader{
  no disj tick1,tick2:Tick{
   r.reading[tick1] = r.reading[tick2]
  }
 }
}
```

Listing 5 Task Definition

```
sig Executable{}
sig Job extends Executable{}
sig Message extends Executable{}
sig Task{
        job: Tick-> one Executable
}
```

ships with $Book$ depending on the $Tick$. Note that we also have the keyword *one* before $Book$ that indicates that each $Tick$ is mapped only to one $Book$. The absence of such a keyword would allow having multiple books per $Tick$. In this way we can express, in a fact for instance, that if $Tick$ represents time, we will not read a book twice by saying that for two different ticks we cannot have the same book related. This is depicted in Listing 4.

Just as with object-oriented programming it is possible to declare a set as extending another set to keep the same relationships.

4.1.3 The Chaotic Model

For our chaotic model one of our key concerns is to be able to deal with scalability. As a result, we focus on the execution at the granularity of tasks. Hence, two core abstractions are used for this purpose: a task-based software model and a task-based executor that represents the hardware and operating system. These two abstractions will be related with a time sequence to model the execution of the system.

4.1.4 Task-Based Software Model

The software is modeled as a graph of tasks as defined in Section 3.1. This task graph is constructed in Alloy using jobs and tasks and a relationship that associates a task to a set of jobs through ticks. These ticks represent the sequence of the jobs, i.e., the sequence of executions of the same function. This is depicted in Listing 5.

Note that in Listing 5 we define both Jobs and $Message$s as extensions of an $Executable$ and a task has a mapping from ticks to executables. This allows us to have a generic task that can be either a task of jobs to be executed

Listing 6 Dependencies Among Tasks

```
sig Task{
 job: Tick-> one Executable,
 dependsOn: set Task
}
```

Listing 7 Executable Graph Signature

```
sig ExecutableGraph{
 tasks: set Task,
}{
 // no job is scheduled twice
 all task:tasks{
  no disj t1,t2:Tick{
   task.job[t1] = task.job[t2]
  }
 }
 // no self dependencies
 all t: tasks{
  t not in t.dependsOn
 }
}
```

by a processor or a task of messages to be executed by a network. To be able to encode the sequencing between tasks we add to the task signature a $dependsOn$ relationship. This is depicted in Listing 6. Note that the $dependsOn$ relationship maps the task to a set of tasks, meaning that it can depend on multiple tasks.

Once we have the tasks and their dependencies then we need to define the constraints of their structure as a graph. This is done by defining an $ExecutableGraph$ signature that contains a set of tasks and state the facts that define its structure (grouped between the braces after the definition of the signature). This is shown in Listing 7.

The constraints in Listing 7 define the minimum structural constraints. In this case it states that one job cannot be schedule twice and that there are no self dependencies between tasks.

4.1.5 Task-Based Executor Model

The graph executor is modeled first by defining how a software graph is executed in a $GraphExecution$ signature and then how this execution is modified with respect to threads and different faulty hardware.

The $GraphExecution$ signature contains one $Executable$ graph along with two mappings from ticks to jobs: $dispatchSchedule$ and $completion$. The jobs are represented by the signature $CycleConsumer$ (parent of $Executable$). Note that these mappings go from one tick to a set of cycle consumers, meaning that there can be more than one cycle consumer executing at any given

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

point. This allows the mapping of concurrency at a coarse granularity level reasoning about interference based on active intervals (between dispatch and completion). This is depicted in Listing 8.

Note that in Listing 8 we have five constraining facts. First, we constrain the jobs being dispatched as belonging to the software. Secondly, we constrain the completed jobs to those that were dispatched before. Thirdly, we force the dispatching of jobs to follow their sequence in the task. Fourthly, we force them not to be dispatched twice since each job represents a single execution. Finally, we force the jobs to complete only once.

We define the ordering of task execution with an extended graph execution that includes sequencing threads. In this execution new restrictions are applied based on whether two tasks belong to the same thread. This can be seen in Listing 9.

In Listing 9, we define a mapping between tasks and threads and add a fact that restricts the execution of jobs from the threads based on the task-to-thread mapping. In particular, if two tasks run on the same thread, then their jobs from the same sequence will run one after another, ensuring that one job cannot be dispatched until the other has completed.

To represent the potential loss of jobs, we created another extension to the graph execution that has a mapping from tasks to processors. We also defined a new extension of a processor that is the $LosslessProcessor$. With these abstractions we model lossless processors by forcing the jobs of tasks deployed over lossless processors to have a completion if they have a dispatch. Note that if a task is not on a lossless processor then jobs can be lost at any point. This is depicted in Listing 10.

It is worth noting that these two extensions are independent, i.e., two tasks can run on two different processors while both of them can run in the same sequential thread. A sequential thread across processors is possible because it represents a synchronized execution perhaps through synchronization protocols. On the other hand, these two tasks could be running on the same processor but on different sequential threads.

The abstractions presented in this section form the basis of the non-functional interpretation of AADL models to evaluate discrepancies. The use of these models is the discussion of our next section.

4.2 The AADL Language

To initiate the discussion of the use of AADL models in our project, a brief introduction is in order. AADL is an architecture description language for the description of runtime architectures of embedded systems. An AADL model describes the hardware and the software as well as the map-

Listing 8 Graph Execution

```
sig GraphExecution{
 software: one ExecutableGraph,
 dispatchSchedule: Tick->CycleConsumer,
 completion: Tick->CycleConsumer
}{
// all dispatch jobs are from software
 all j:Executable,t:Tick{
  j in dispatchSchedule[t] =>
  some t2:Tick,task:software.tasks{
   j in task.job[t2]
  }
 }
 // all jobs complete after
 // they are dispatched
 all t:Tick,j:CycleConsumer{
  j in completion[t] =>
  some t2:Tick{
   j in dispatchSchedule[t2]
   and
   t in time/nexts[t2]
  }
 }
 // all jobs are dispatch in the
 // order they appear in the task
 all disj j1,j2:Executable{
  all t1,t2,t3,t4:Tick,task:software.tasks{
   (j1 in task.job[t1]
   and
   j2 in task.job[t2]
   and
   j1 in dispatchSchedule[t3]
   and
   j2 in dispatchSchedule[t4]
   and
   t2 in time/nexts[t1] ) =>
   t4 in time/nexts[t3]
  }
 }
 // Not dispatch twice
 all j:CycleConsumer{
  no disj t1,t2:Tick{
   j in dispatchSchedule[t1]
   and
   j in dispatchSchedule[t2]
  }
 }
 // jobs cannot complete twice
 all j:CycleConsumer{
  no disj t1,t2:Tick{
   j in completion[t1]
   and
   j in completion[t2]
  }
 }
}
```


Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

Listing 9 Threaded Graph Execution

```
sig ThreadedGraphExecution
 extends GraphExecution{
 executedBy: Task -> one Thread
}{
 // all tasks executed by the same thread are
 // sequential (not parallel) to each other
 all disj task1, task2 :software.tasks{
  executedBy[task1] = executedBy[task2] =>
  all s1: Tick, j1,j2:Executable{
   j1 in task1.job[s1]
   and
   j2 in task2.job[s1] =>
   all disj t1, t2: Tick{
    j1 in completion[t1]
    and
    j2 in dispatchSchedule[t2] =>
    t2 in time/nexts[t1]
   }
  }
 }
}
```

Listing 10 Deployed Lossless Graph Execution

```
sig LosslessProcessor
 extends Processor{}
sig DeployedGraphExecution
 extends GraphExecution{
 deployedTo: Task -> one Processor
}{
 all task:software.tasks {
  deployedTo[task] in LosslessProcessor =>
  all seq1: Tick, j: Executable {
   j in task.job[seq1] and
   some t1:Tick{
    j in dispatchSchedule[t1]
   } =>
   some t2:Tick{
    j in completion[t2]
   }
  }
 }
}
```

ping between them. The software is described with systems, processes, threads, data, and subprograms that communicate throught ports (that are part of their features) with port connections. Hardware is described as interconnected processors, devices, and busses. Properties are used in these models to add annotations to the model that complements their structural description. AADL execution semantics is fully defined specifying the conditions under which computations start and stop due to communication interactions and/or time passing. While more discussion is needed to understand the full semantics of AADL, we believe the language is intuitive enough to allow an *in-situ* discussion of the AADL listings.

4.3 AADL Models for Non-Functional Discrepancies Discovery

The focus of the analysis of AADL models is on threads, processors, and busses. With threads and their connections we construct the sequential semantics of the model. With processors and busses we define the possibility of losing jobs. To be able to compare two models we use a common set of subprogram calls between the models. With this set we define the base of the comparison. In other words, we require both models to have the same set of calls to the subprograms even though the exact location of the calls may vary. These variations include distributing the calls in different threads and processors. The difference between the models is expected to produce different executions in terms of sequencing and reliability.

4.4 Restricting Behaviors with AADL Properties

Part of the purpose of our analysis is to enable the designer to make choices on the modification of the architecture of a system already working to enhance some quality attributes such as fault-tolerance or throughput. As a result, it is not only important to discover the discrepancies but also to annotate the model when additional properties are added to the architecture. For instance, one way to avoid message loss in a bus is to add a reliable protocol to it. Furthermore, we also want to define when a discrepancy is acceptable, e.g. that lossing messages is OK. We believe that annotating the acceptance of a discrepancy is of paramount importance to keep the designers intent properly documented in a machine checkable format. This is the reason why our default interpretation of the model is chaotic, i.e., to ensure that the designer annotates completely its intent.

There are two types of annotations in our chaotic models: increments in the hardware / operating platform order behavior and increments in the software tolerance to discrepancies. Both annotations are represented as AADL proper-

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

ties. All the properties for chaotic analysis are bundled into the chaotic property set.

4.4.1 Annotating Order

Order can only be obtained if the parties involved are synchronized either by being contained in a common AADL thread or if the threads are connected by a synchronized connection. AADL has an explicit construct for synchronized connections: immediate connections. These connections ensure that if two threads are connected only with immediate connections and both are periodic with the same period then they should be executed in sender-receiver order, not allowing the receiver to run until the sender is done and the data in their connection is communicated. To be able to take into account other forms of synchronization we have added a property that can be applied to both connections and flows. This property is called $InOrder$. When an $InOrder$ property is set to true in a flow or connection, the connected components are considered to be executed in sender-receiver order.

Faulty behavior is simpler to encode. By default, all hardware is considered faulty. When we want to capture that hardware does not fail (e.g. when retransmitting protocol is added) then we set its $Lossless$ property to true. This property is applicable to Processors and Busses.

4.4.2 Annotating Software Tolerance

Three properties are used to encode software tolerance: $LossTolerant$, $ReorderTolerant$, and $OrderIrrelevant$. $LossTolerant$ and $ReorderTolerant$ are applied to connections, flows, and subprogram calls. It means that the execution of such a piece is not that relevant and we can tolerate occasional losses or reordering of messages. The $OrderIrrelevant$ property on the other hand allows us to define sets of threads that can be executed in any order respect to another. For instance putting the property as `OrderIrrelevant (reference thread1, reference thread2) applies to thread3` implies that we are not concern with the ordering of jobs from threads $thread1$, $thread2$, and $thread3$.

5 Example

To illustrate the use of the chaotic model consider a stability control system (SCS) interacting with a cruise control system (CCS) in a car. The SCS is in charge of controlling the car dynamics if it detects a maneuver that can put in jeopardy the stability of the car. A typical correction performed by the system would be the braking of individual wheels and the deceleration of the engine. On the other

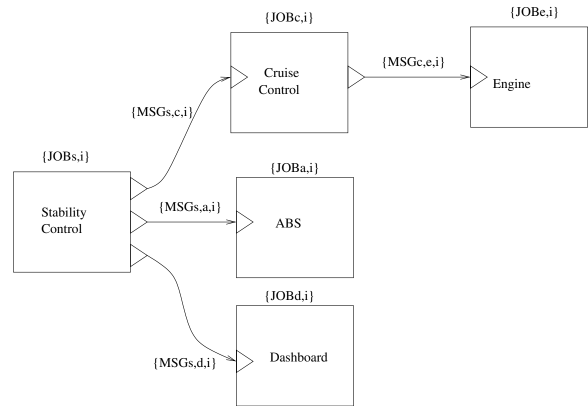


Figure 3. Stability Control Example

hand, the CCS controls the acceleration of the car to maintain a fixed speed. This system is generally used when driving on the highway. However, when the SCS kicks in while the CCS is active, we need to validate that the SCS deactivates the CCS before it attempts to brake to avoid braking while accelerating. This system is depicted in Figure 3.

Now consider the case where the prototype of the system is designed in a single-thread system to test the functionality. Its AADL description is specified in Listing 11.

In this model you can see a single thread and all the calls inside that thread. This sequence of calls involves a sequential control flow that is implicit. It means that, if we spread these calls into different threads and processors, we need to verify that both the order of execution and the failure modes are equivalent to the single-thread model.

Now consider a fully-threaded variation of the SCS-CCS model. This system is partially listed in Listing 12 where each thread calls the subprogram with the similar name (not shown).

In our discussion we focus on a single sequence that can go wrong: braking while accelerating. This phenomenon can happen because the SCS brakes individual wheels while it failed to stop the CCS. This is because the CCS would detect a decrease in the speed it is suppose to maintain and it would accelerate. Given that our focus is in two dimensions, failure and sequencing, running our analysis would show us the discrepancies in these two dimensions. In the process of the constructions we would then need to ensure that the annotated properties are implemented in the implementation. For instance, if we add a $Lossless$ property to a bus, we need to ensure that a lossless protocol is added to the implementation.

In this particular example we added properties to ensure that our execution platform does not loss jobs or reorder them except for the dashboard indication subprogram. For this last program, we added properties to specify that it is

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

Listing 11 Sample Reference Model

```
subprogram StabilityControlProgram
 features
  wheelToBrake: out parameter;
end StabilityControlProgram;

subprogram DisableCruiseControlProgram
end DisableCruiseControlProgram;

subprogram ABSBrakeWheelProgram
 features
  wheel: in parameter;
end ABSBrakeWheelProgram;

subprogram DashboardStabilityIndicatorOnProgram
end DashboardStabilityIndicatorOnProgram;

thread StabilityControlSingleThread
end StabilityControlSingleThread;

thread implementation
        StabilityControlSingleThread.i
 calls
  triggered: {
   stability: subprogram
        StabilityControlProgram;
   disableCC: subprogram
        DisableCruiseControlProgram;
   brakeWheel: subprogram
        ABSBrakeWheelProgram;
   indicateDashb: subprogram
        DashboardStabilityIndicatorOnProgram;
  };
 connections
  c1: parameter stability.wheelToBrake->
        brakeWheel.wheel;
end StabilityControlSingleThread.i;
```

Listing 12 Top Level Full-Threaded Model

```
process implementation
        StabilityControlSingleProcess.i
  subcomponents
    stability: thread StabilityControlThread.i;
    cruise: thread CruiseControlThread.i;
    abs: thread ABSThread.i;
  connections
    c1: event port stability.disableCC ->
        cruise.disable;
    c2: event data port
        stability.brakeWheel ->
        abs.brakeWheelPort;
    c3: event port
        stability.indicateDash -> indicateDash;
end StabilityControlSingleProcess.i;

process implementation DashboardProcess.i
  subcomponents
    dashb: thread DashboardThread.i;
  connections
    c1: event port indicateDash ->
        dashb.turnOnCCIndicator;
end DashboardProcess.i;

system implementation Final.i
  subcomponents
    stabilityProcess: process
        StabilityControlSingleProcess.i;
    dashboardProcess: process
        DashboardProcess.i;
    procOne: processor ProcOne;
    procTwo: processor ProcTwo;
    busOne: bus BusOne;
  connections
    b1: bus access busOne ->
        procOne.busConnection;
    b2: bus access busOne ->
        procTwo.busConnection;
    e1: event port
        stabilityProcess.indicateDash ->
        dashboardProcess.indicateDash;
  properties
    Actual_Processor_Binding =>
        reference procone applies to
        stabilityProcess;
    Actual_Processor_Binding =>
        reference proctwo applies to
        dashboardProcess;
    Actual_Connection_Binding =>
        (reference busone) applies to e1;
end Final.i;
```


Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

not important and it can be either lost or reordered with respect to the rest. The annotated model is listed in Listing 13.

Note that to ensure consistency we need to put annotations about both failure and reorder. In addition note that in the case of the communication to the dashboard process, the connection is characterized as loss and reorder tolerant.

6 Final Remarks

Discovering faults introduced by multi-processor deployment and concurrency is difficult. In this paper we presented a model to automatically discover potential faults in the architecture of a system in the form of discrepancies between a model and a variant with modified concurrency. In our approach we used Alloy to specify the execution semantics of AADL concurrency and automatically analyze discrepancies between two models. We then defined a property set in AADL to annotate AADL models with properties about sequencing, reliability and tolerance to faults. We also presented an example were the utility of this model is illustrated. We believe this approach is both practical and thorough due to the high level of abstraction at which it works. In the future we expect to add a larger variety of faults such a value faults and quantitative sequencing faults where, assuming a periodic execution, a value received can be from several periods before. Given the generality of Alloy, the speed of the verification is limited. To overcome this limitation a static analysis algorithm was developed providing a significant speed improvement.

References

[1] S. Aerospace. Architecture Analysis and Desgin Language (AADL), 2004.

[2] P. Attie, A. Arora, and E. A. Emerson. Synthesis of fault-tolerant concurrent programs. ACM Transactions on Programming Languages and Systems, 26(1):125–185, January 2004.

[3] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K.Yorav. Efficient verification of sequential and concurrent c programs. Formal Methods in Systems Design, 25(2):129–166, 2004.

[4] H. Erdogmus. Architecture-driven verification of concurrent systems. Nordic Journal of Computing, September 1997.

[5] N. Halbwachs, E. Jahier, P. Raymond, X. Nicollin, and D. Lessens. Virtual Execution of AADL models Via a Translation into Synchronous Programs. In Proceedings of the Seventh International Conference on Embedded Software (EMSOFT 2007), September 2007.

[6] G. Holzmann. The model checker spin. Software Engineering, 23(5):279–296, 1997.

[7] D. Jackson. Alloy: A lightweight object modelling notation. Technical Report 797, 2000.

Listing 13 SCS / CCS System with Chaotic Annotations

```
process implementation
      StabilityControlSingleProcess.i
  subcomponents
    stability: thread
      StabilityControlThread.i;
    cruise: thread CruiseControlThread.i;
    abs: thread ABSThread.i;
  connections
    c1: event port stability.disableCC ->
      cruise.disable
      {chaotic::InOrder => true;};
    c2: event data port stability.brakeWheel ->
      abs.brakeWheelPort
      {chaotic::ReorderTolerant => true;};
    c3: event port stability.indicateDash ->
      indicateDash;
end StabilityControlSingleProcess.i;
process implementation DashboardProcess.i
  subcomponents
    dashb: thread DashboardThread.i;
  connections
    c1: event port indicateDash ->
      dashb.turnOnCCIndicator;
end DashboardProcess.i;
system implementation Final.i
  subcomponents
    stabilityProcess: process
      StabilityControlSingleProcess.i;
    dashboardProcess: process
      DashboardProcess.i;
    procOne: processor ProcOne
      {chaotic::Lossless => true;};
    procTwo: processor ProcTwo
      {chaotic::Lossless => true;};
    busOne: bus BusOne
      {chaotic::Lossless => true;};
  connections
    b1: bus access busOne ->
      procOne.busConnection;
    b2: bus access busOne ->
      procTwo.busConnection;
    e1: event port
      stabilityProcess.indicateDash ->
      dashboardProcess.indicateDash
      {chaotic::ReorderTolerant => true;
       chaotic::LossTolerant => true;};
  properties
    Actual_Processor_Binding =>
      reference procone applies to
      stabilityProcess;
    Actual_Processor_Binding =>
      reference proctwo applies to
      dashboardProcess;
    Actual_Connection_Binding =>
      (reference busone) applies to e1;
end Final.i;
```


Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.

[8] Z. Manna and P. Wolper. Synthesis of communicating processes from temporal logic specifications. ACM Transactions on Programming Languages and Systems, 1984.

[9] K. McMillan. Symbolic Model Checking - An Approach to the State Explosion Problem. PhD thesis, Department of Computer Science - Carnegie Mellon University, 1992.

[10] F. Ortmeier, A. Thums, G. Schellhorn, and W. Raif. Combining formal methods and safety analysis: The formosa appraoch. Lecture Notes in Computer Science, 31478/2004:474–493.

[11] P. Wolper. Specification and synthesis of communicating processes using an extended temporal logic. In Proceedings of the Symposium on Principle of Programming Languages, 1982.

Authorized licensed use limited to: IEEE Xplore. Downloaded on October 28, 2008 at 23:29 from IEEE Xplore. Restrictions apply.