2017-12-01

A Coverage Metric to Aid in Testing Multi-Agent Systems

Jane Ostergar Linn
Brigham Young University

A Coverage Metric to Aid in Testing Multi-Agent Systems

Jane Ostergar Linn

A thesis submitted to the faculty of
Brigham Young University
in partial fulfillment of the requirements for the degree of

Master of Science

Eric G. Mercer, Chair
Michael A. Goodrich
William Arthur Barrett
Neha Shyam Rungta

Department of Computer Science

Brigham Young University

ABSTRACT

A Coverage Metric to Aid in Testing Multi-Agent Systems

Jane Ostergar Linn
Department of Computer Science, BYU
Master of Science

Models are frequently used to represent complex systems in order to test the systems before they are deployed. Some of the most complicated models are those that represent multi-agent systems (MAS), where there are multiple decision makers. Brahms is an agent-oriented language that models MAS. Three major qualities affect the behavior of these MAS models: workframes that change the state of the system, communication activities that coordinate information between agents, and the schedule of workframes. The primary method to test these models that exists is repeated simulation. Simulation is useful insofar as interesting test cases are used that enable the simulation to explore different behaviors of the model, but simulation alone cannot be fully relied upon to adequately cover the test space, especially in the case of non-deterministic concurrent systems. It takes an exponential number of simulation trials to uncover schedules that reveal unexpected behaviors. This thesis defines a coverage metric to make simulation more meaningful before verification of the model. The coverage metric is divided into three different metrics: workframe coverage, communication coverage, and schedule coverage. Each coverage metric is defined through static analysis of the system, resulting in the coverage requirements of that system. These coverage requirements are compared to the logged output of the simulation run to calculate the coverage of the system. The use of the coverage metric is illustrated in several empirical studies and explored in a detailed case study of the SATS concept (Small Aircraft Transportation System). SATS outlines the procedures aircraft follow around runways that do not have communication towers. The coverage metric quantifies the test effort, and can be used as a basis for future automated test generation and active test.

Keywords: Coverage, multi-agent systems, Brahms, static analysis, simulation, model checking

ACKNOWLEDGMENTS

Thank you to my husband John, who took it upon himself to do laundry, change diapers, make meals, and anything else that his wife could use as "productive" procrastination.

A big thank you to baby William who was a painless pregnancy and who started sleeping through the night at only a month old.

Also thank you to Dr. Burton, who encouraged me to pursue a graduate degree in the first place.

Finally, thank you to Dr. Mercer who helped me focus my ideas and work them into a cohesive final product.

Table of Contents

List of Figures

List of Tables

List of Listings

Chapter 1

Introduction

1.1 Background

Modeling is a tool used to quantify the behavior of a system in order to better understand it. Verification of these models has come to be known as model checking, a technique used to explore the different states of a system to determine if a given property is violated. The first step in model checking is to construct a model that represents the system. If that model is reliable, exploring the model can reveal ambiguities, holes, and inconsistencies that otherwise may not have been found until much later in the development of the system [2].

Models that replicate the behavior of multiple agents in a single environment are Multi-Agent Systems (MAS). These systems are more complicated than other models because they attempt to capture the behavior of multiple decision makers working independently to reach their own goals. Though these agents are independent, they are affected by each other because they use information from the same environment, and their actions can change the environment. This can cause dependencies that can be hard to detect. MAS are complex and concurrent pieces of software that are difficult to test.

The primary tool for MAS model validation is simulation, but because of the complexity of the systems, simulation can be inadequate on its own. Generally, multiple simulations are run and then compared to results from either known expected system behavior or validated behavior from human-in-the-loop simulation on prototype systems. In some instances, especially those involving brand new systems, there is no known expected system behavior so the results must be compared to results from human-in-the-loop simulation. This validation comes very late in the development process, so any change to the model or system at that point is expensive.

Also, simulation may not cover the test space well. It is possible that the provided scenarios explore the same behavior space of the code over and over again. Simulation does not have the ability to choose to go down unexplored paths in the code, or even systematically handle non-determinism at all. Since MAS are likely non-deterministic, it is difficult to capture possible behaviors with simulation since simulation relies on the runtime which may be more deterministic than wanted. On its own, simulation is an inefficient way to test multi-agent systems.

With the rise of MAS models as a viable choice in designing and analyzing complex human-computer systems, it is necessary to develop cheaper ways to verify these models while keeping humans and the software safe. The complexity of these models makes simulation insufficient in determining whether or not these models are correct relative to the actual intended concept. Anyone creating complex models like these with multiple decision makers needs systematic and formal techniques to validate the models, the kind of validations that have been developed and researched much more thoroughly in software engineering.

1.2 Coverage

Software engineering has had extensive research dedicated to validating complex concurrent systems. One of the most common approaches to software validation is to define coverage metrics on the program test, and then generate tests to achieve higher coverage. There are many different types of coverage, the main four for software being method, branch, condition, and statement coverage. These metrics inform the tester which areas of the program are still uncovered, prompting the tester to try different input in an attempt to cover those areas. Concurrent systems have additional metrics like context switches and thread coverage which can provide additional insight when testing the software. Coverage is a way to quantify the test effort in terms of observed events. A list of events to be witnessed are the coverage requirements, and the ratio of witnessed events to the list of events to witness is the coverage of a system. The higher the ratio (the more events that have been witnessed), the higher the coverage. The idea of coverage in software can be applied to MAS models as well.

Brahms is an agent-oriented language that can model MAS, and is the language used for the research in this thesis. NASA uses Brahms to model the interactions between humans, aircraft, and other systems with respect to aviation safety [9]. The coverage metric defined by this thesis is in terms of the Brahms language. In Brahms, each agent or object has workframes that can be activated under the right conditions. These workframes do the "work" for the agents and objects. Communication activities exchange information between two agents or objects. Because of the non-determinism in many MAS, the order in which workframes are executed depends on the scheduler. Different schedules may have different consequences, so workframe scheduling is taken into account when defining the coverage metric.

In this thesis, coverage is defined as events that can be witnessed during simulation. Repeated simulation is helpful as long as interesting test cases are chosen that may behave differently in different simulations, but no matter how good the input is, it is difficult (and sometimes impossible) for simulation to cover different schedules of concurrent interactions. This is why a list of coverage requirements can be helpful to show which behaviors of the model have been reached. For Brahms systems, this thesis defines a coverage requirement list as composed of workframe coverage requirements, communication coverage requirements, and schedule coverage requirements. Each item on the list is a requirement that can be marked when it is witnessed in some simulation of the model.

This thesis presents a tool that given a system as input, can calculate the coverage, see Figure 1.1. The first part of the tool is the parser, which uses static analysis algorithms to automatically generate coverage requirements. The static analysis determines shared attributes in the system and the agents that interact with those shared attributes. Because it is difficult to know how agents and objects will interact statically, the work in this thesis

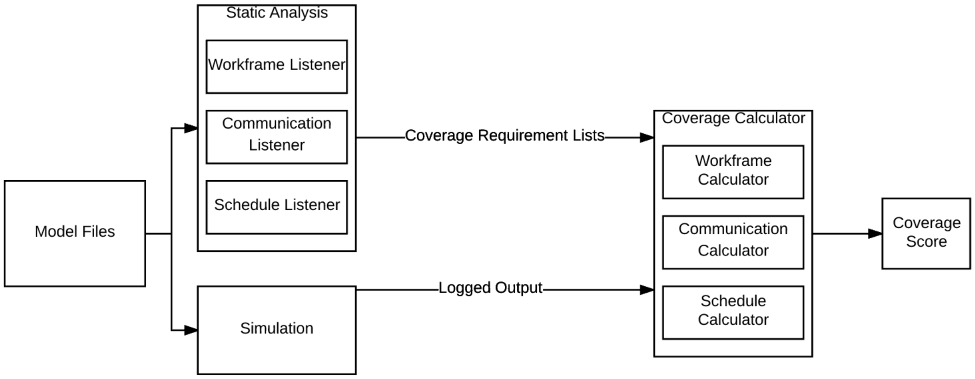


Figure 1.1: Coverage Calculation Process

over-approximates the true set of feasible coverage requirements. As a result, there may be events on the list of requirements that are impossible to observe in any execution of the system.

In order to observe the feasible coverage requirements, the system is instrumented according to the functional logger so that coverage measurement can be automated as the simulation runs. The coverage calculator then takes the logged output from the simulation and compares it to the coverage requirements from the static analysis to calculate the coverage for the system.

Several empirical studies were run on Brahms models, as well as a more detailed case study of NASA's Small Aircraft Transportation System (SATS), which was modeled in Brahms. These studies showed how to use the coverage metric in a practical sense. The coverage metric shows what behaviors the scenarios have not yet explored, and when scenarios are written to meet the coverage requirements more of the model behavior is revealed. The coverage metric is especially useful when unusual, unexpected, or unintended model behavior is revealed. With the confidence the coverage metric provides that the model is behaving correctly, the model can then be verified according to existing conventions. The coverage metric is a necessary means to an end: verification of the system.

The following chapter surveys the existing literature on the verification of multi-agent systems. Coverage metrics of complex concurrent software are surveyed, as well as verification techniques of knowledge-based systems. Following that the Brahms agent-oriented language is presented in greater detail. The remaining chapters outline the three coverage requirements, an empirical study and a case study of the effectiveness of these metrics, and what research can be done on this foundation.

Chapter 2

Related Work

There is no existing coverage metric for multi-agent systems. Simulation is widely used to test these systems, but the simulation itself doesn't tell you if the model is behaving correctly, or even what parts of the model were activated during simulation. This coverage metric fills the knowledge gap between writing the model and verifying the model.

2.1 Verification of Multi-agent Systems

Numerous tools exist for the verification of multi-agent systems. The predominant approach to verifying any model is to translate the model into a form that can be inputted into one of these tools. One method presented in [3] automatically translates multi-agent systems written in AgentSpeak (a logic-based agent-oriented programming language) into Promela or Java, and from there, verification can be performed using SPIN or JPF respectively. One downfall of their approach is that they restrict the systems they work with to be finite state systems. Since modern models represent real world systems and affect real people, that restriction cannot be maintained.

A framework presented in [9] generates the behavior space of the system and then translates that intermediate representation to input for the existing verification tools, SPIN, PRISM, and NuSMV, so the translation to readable input for these tools is automatic. This work focuses on Brahms workframe and schedule coverage, assuming that reaching the states in this way is comparable to generating the whole behavior space. Generating the behavior space is much more expensive, and requiring more computation and time, whereas workframe and schedule coverage are much more compact and will produce similar results. A model that is the explicit state space is unwieldy.

2.1.1 Epistemic verification techniques

These existing verification tools (SPIN, PRISM etc) reason about the system and its properties, but only deal with LTL (logic which reasons about when certain properties will hold) or PCTL (which reasons about the probabilities of properties). There are also tools that verify MAS using epistemic properties, the beliefs and intentions of the agents and objects in the system. Penczek and Lomuscio define a framework in [14] that combines CTL with notions of knowledge in a system. They use interpreted systems to verify multi agent systems. MCK, a system introduced in [6], is also concerned with the knowledge flow in a model. It takes the agent environment, agent protocol (rules that determine the behavior and scope of the agent), and knowledge properties to be model checked as input. Another system [12], attempts to model check multi-agent systems by partial order reduction. The original model is mapped to a semantically equivalent (with respect to the property of interest) smaller model, and then the smaller model is checked for that property. It uses CTLK and LTLK logics, which are essentially CTL and LTL with the added knowledge modality.

This analysis of the knowledge flow is different from whether or not the system is behaving as intended. It operates under the assumption that the model is doing what it is supposed to be doing, and just checks what the agents know at a given point in time.

2.2 Concurrent Software

Since multi-agent systems are analogous to concurrent programs (multiple agents are making decisions at the same time), there is much related work in the concurrent program testing fields. Hong et al [7] present a method that achieves higher and faster coverage than random testing, using a thread scheduling technique that manipulates the thread schedules to cover more uncovered coverage requirements, which are calculated first in an "estimation phase" that dynamically builds a thread model of the program. The estimation phase computes and reports a set of synchronization-pairs, which are pairs of code locations that have lock statements executed on the same lock.

An empirical study done by Hong et al [8] illustrate the importance of a good coverage metric for concurrent systems. They found that though the metrics can be moderate to strong predictors of testing effectiveness, they also vary across programs and no metric consistently provided effective testing. Terrangi and Cheung [19] use an iterative approach to define their coverage metric for concurrent classes. Their coverage metric captures context-sensitive information using only single-threaded execution, and then they use an algorithm to iteratively explore the space of possible sequences of method calls to find the sequence that achieves the highest coverage (according to their metric). They also use a thread schedule algorithm to increase the context-sensitive interleaving coverage.

Jackson and Damon present the *small scope hypothesis* [10], which states that a high proportion of bugs can be found within a small scope of tests. The small scope in this case is able to limit the context switching to two threads for concurrent programs [13]. This idea helps to reduce the state explosion problem, because the tests generated are limited to a small scope of the test space. Another idea presented in [22] called the *value independence hypothesis*, states that a majority of concurrency bugs are triggered if the memory dependencies are exposed, no matter what the data values of the shared variables are. These two ideas in software testing can apply to testing models as well. We can successfully generalize the state space of a MAS by not concerning ourselves with the specific agents or agent interactions.

2.3 Multi-agent modeling languages

There are many different agent-oriented languages designed to model multi-agent systems. Allan reviews many different tools and languages for modeling multi-agent systems in [1]. There is also very useful chart comparing different tools for agent-based modeling on Wikipedia

[20]. Many of these languages are domain-specific. We have chosen Brahms [4], as it can serve as a modeling tool for many different domains.

Chapter 3

The Brahms Modeling Language

3.1 Brahms Language Definition

Brahms is an agent-oriented modeling language, originally developed at NASA [4]. It can be used for modeling interactions between concepts (agents or objects) in an environment. In this section is an illustrative example of a Brahms model.

3.1.1 Brahms as a BDI

Brahms is a belief, desire, and intent system (BDI). These types of systems are used to model the behavior of intelligent agents. It uses an agent's beliefs, desires, and intentions to solve problems. Beliefs characterize the state of the agent, desires reflect the objectives an agent has, and intentions are desires that have been carried out to some degree. Agents act according to their beliefs, even if what they believe is not a "fact" in the world. Events that occur in the system, either due to agents' actions or external sources, update the state of the agent.

BDI systems are used to simulate real scenarios, and are increasingly important as more and more systems are designed to handle high-level management tasks in complex environments [16]. Modeling these complex systems in an object-oriented environment is much more difficult, and the perspective shift that agent-oriented programming—particularly BDI systems—offers, makes modeling and verifying these systems easier.

3.1.2 Illustrative example

Throughout this section a simple example model is used to help explain the Brahms language. In this model there is a Cashier and three Orders. The Cashier attempts to complete any Orders that have not been completed, and any order that has not been completed alerts all Cashiers in the area (in this scenario, only two of the three have been completed). Once a Cashier is alerted to the existence of an Order, it creates a belief about that Order and it can then be processed.

3.1.3 Concepts

Brahms models are composed of concepts; the most interesting concepts are groups, classes, and their agent and object instances. There are other concepts, like areas, paths, and conceptual objects, but we restrict our focus to concepts that can act on each other and on the environment. agents and objects, the only two concepts that can act and are not just acted upon. For a more detailed discussion on these other types of concepts and their function, see [4]. Objects can be instances of a class, and agents can be members of a group. Much of the time, agents and objects are defined primarily in their respective classes or groups.

Listing 3.1 is an example class that defines "Order" objects. Objects can have attributes, relations, activities, and workframes. For the sake of brevity, attributes and relations will be referred to as attributes only. Attributes are used to describe the state of the object and can have primitive types (string, boolean, int, float, etc.) or can have more complicated types, like another agent or object, or a map. Objects operate according to facts that exist in the world.

The Order object has 3 attributes: an identifying number, a boolean value that describes whether or not the order has been completed, and a relation to a cashier the order has been completed by. When an Order is completed, it updates its attributes to reflect the change. As long as the order has not been completed, it will repeatedly “alert” all Cashiers.

Listing 3.1: Example Brahms Class: Order

```
class Order {
    attributes:
        public int number;
        public boolean completed;
    relations:
        public Cashier completedBy;
    activities:
        communicate alertCashier(Cashier c){
            max_duration: 2;
            with: c;
            about:
                send(current.completed = unknown);
        }
    workframes:
        workframe wf_alertCashier {
            repeat: true;
            variables:
                collectall(Cashier) cashier;
            when(
                knownval(current.completed = false) and
                unknown(current completedBy) and
                known(cashier.name)
            )
            do {
                alertCashier(cashier);
            }
        }
}
```

Agents are similar to objects in that they also have attributes, activities, and workframes. They can be members of classes, like the example Cashier class in Listing 3.2. Agents operate according to their own beliefs, rather than facts in the world, as an object does.

Listing 3.2: Example Brahms Group: Cashier

```
group Cashier {
    attributes:
        public string name;
    activities:
        primitive_activity workOnOrder(){
            max_duration: 5;
        }
        primitive_activity clean(){
            max_duration: 10;
        }
        communicate processOrder(Order order){
            with: order;
            about:
                send(order.completed = unknown);
        }
    workframes:
        workframe wf_workOnOrder {
            priority: 1;
            variables:
                foreach(Order) order;
            when (
                knownval(order.completed = false)
            )
            do {
                workOnOrder();
                conclude((order.completed = true));
                processOrder(order);
            }
        }
        workframe wf_clean {
            variables:
            when ( )
            do {
                clean();
            }
        }
}
```

A Cashier agent has only one attribute: it's identifying name.

Instances

Agent and object instances inherit all the attributes, activities, and workframes of their parent group or class, but are typically initialized with different initial beliefs and facts. Initial beliefs and facts can be instantiated at the general group or class level, but are typically left to the specific agents and objects. Agents and objects can also define additional attributes, activities, and workframes specific to themselves or override existing activities or workframes. Here are some examples of agent and object instances.

Listing 3.3: Example Brahms Agent: Bob

```
agent Bob memberof Cashier{
    initial_beliefs :
        (current.name = ``Bob'');
        (Order_1.completed = true);
        (Order_2.completed = false);
        (Order_3.completed = false);
}
```

Note that even though a Cashier agent only has one attribute (see Listing 3.2), it can have beliefs beyond its own attributes.

Listing 3.4: Example Brahms Objects

```
object Order_1 instanceof Order {
    initial_facts :
        (current.number = 1);
        (current.completed = true);
        (current completedBy Bob is true);
        (Bob.name = ``Bob'');
}
object Order_2 instanceof Order {
    initial_facts :
        (current.number = 2);
        (current.completed = false);
        (Bob.name = ``Bob'');
}
```

In order for these concepts to interact, they must be aware of each other, which is why the Order objects have fact initializations about the Bob agent, and the Bob agent has initial beliefs about the Order objects.

3.1.4 Activities

There are several different types of activities in Brahms, though we focus on only four types: primitive, communicate, broadcast, and Java.

- **Primitive** activities have no affect on the environment, they simply take time.
- **Communicate** activities can be of two types: send or receive. A send communicate activity sends information from the initiating concept to one or more concepts it is

bound to. The receiving concept(s) updates its beliefs according to the communication. In the example above (Listing 3.1, Listing 3.2), both the Order class and Cashier group have send communication activities. They are only bound to one other concept. When a Cashier communicates with an Order, it sends its own belief about that order's attribute "completed." The receiving order would then update its "completed" attribute to reflect that communication. The "unknown" distinction is necessary because depending on when that activity is called, the sent value will be different. "Unknown" allows the concept to send the most recent value of the attribute in question. A receive communicate activity receives information from the one or more concepts it is bound to, and then updates its own beliefs according to the communication.

- **Broadcast** activities are like communication activities, but rather than binding to concepts, they send or receive information from any concepts in the area. When no area is defined, all the concepts are in the same area by default, which is the case for models in this thesis.

- **Java** activities are defined outside of Brahms and are used when an activity needs more functionality than Brahms can provide. For example, the case study in chapter 8 uses a Java activity to increment an index mod 4, since Brahms does not support modular arithmetic. Java activities can also be used to simplify a class. Rather than creating a workframe for every possible condition, a single Java activity with conditional statements can be used. Java also has more complicated data types that can be useful. Using the Java API, these activities can access anything from the model at any time.

For all of these activities, you can specify the amount of time they take. If not specified, they happen instantaneously.

3.1.5 Workframes

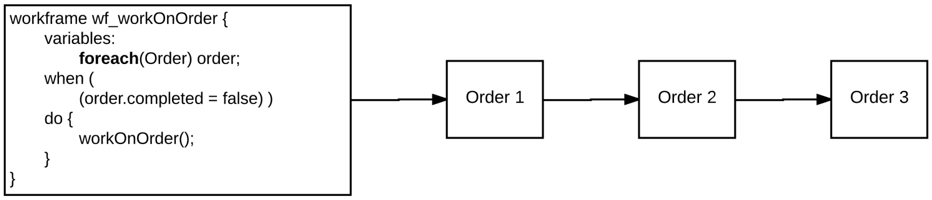
Workframes are the force behind the execution of a model. They activate if their preconditions, which depend on the state of the model, are met. Agents' workframes are activated according to their own beliefs, while objects' workframes are activated according to facts in the world. Objects can also activate workframes according to beliefs if the workframe is defined as a "data frame" rather than the default "fact frame". Once active, workframes can execute activities and conclude statements. Conclude statements update the beliefs of an agent and the facts of the world for an object. For example, the Cashier agent (see Listing 3.2) has a workframe "wf_workOnOrder" which activates when the Cashier believes that an order has not yet been completed ("order.completed = false"). When an order meets that criteria, the primitive activity "workOnOrder()" is called, and then the Cashier updates it's belief about the order's "completed" attribute in a conclude-statement. Once that belief has been concluded, it is communicated to the order, so that the order can update its attributes.

By default workframes execute only once, but they can set their "repeat" flag to true and execute infinitely many times. Workframes also have a priority that determines what order workframes will be called in. The default priority is 0, and the higher the number, the higher the priority. In Listing 3.2 the Cashier's "wf_clean" will always execute after "wf_workOnOrder" if "wf_workOnOrder" can be activated.

3.1.6 Variables

Variables in Brahms are the concepts or other data types that a workframe binds to before activating. There are three types of variable bindings:

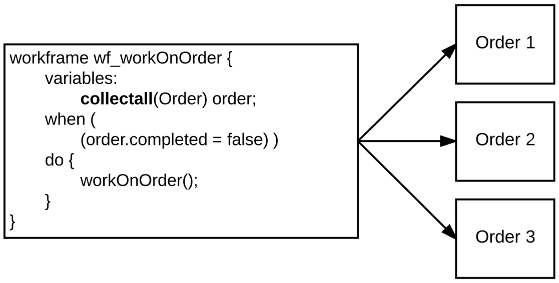
- For-each: creates separate workframe activations, each bound to a satisfying variable. These workframes are executed one at a time. See Figure 3.1. This can cause the first workframe to invalidate the following workframes, so the order is interesting. This is the type of binding that the Cashier has for each order it must work on in "wf_workOnOrder".
- Collect-all: similar to for-each, a separate workframe is activated for each satisfying variable, but these workframes are executed all at once. See Figure 3.2 for an example of the Cashier working on all Orders at the same time. The "wf_alertCashier" workframe



```
workframe wf_workOnOrder {
        variables:
                foreach(Order) order;
        when (
                (order.completed = false) )
        do {
                workOnOrder();
        }
}
```

Figure 3.1: For-each example

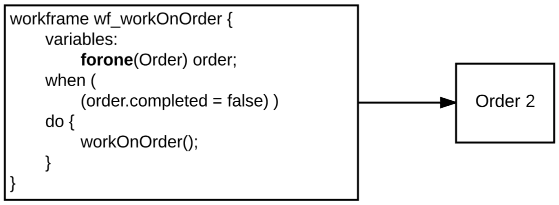
has a collect-all variable binding to alert all the cashiers at once that the order has not been completed.



```
workframe wf_workOnOrder {
        variables:
                collectall(Order) order;
        when (
                (order.completed = false) )
        do {
                workOnOrder();
        }
}
```

Figure 3.2: Collect-All example

- For-one: rather than binding to all satisfying variables, this picks a single satisfying variable at random to create an active workframe instance. See Figure 3.3. Often, this binding is used for introducing primitive values needed for comparison in the preconditions.



```
workframe wf_workOnOrder {
        variables:
                forone(Order) order;
        when (
                (order.completed = false) )
        do {
                workOnOrder();
        }
}
```

Figure 3.3: For-one example

These variables are bound according to the preconditions of the workframe. If the variable satisfies the preconditions, it can be bound to a workframe activation. Once bound, these variables can be passed into activities within the workframe and can be part of conclude statements.

3.2 Brahms Semantics

The semantics of Brahms allows for a great deal of concurrency. Each concept instance in the system runs on its own thread, and sometimes each workframe in each concept instance runs on its own thread. This allows for many workframes to activate at the same time. Workframes are activated according to their preconditions. If there are multiple workframes whose preconditions have been met, they are ordered according to their priority. If the workframes are the same priority, Brahms will activate them in a random order.

Because Java activities can access any part of the model via the Java API, we synchronize variable accesses from within the Java activity, so that any variable manipulation that happens inside the Java activity is uninterrupted. Without the synchronization, the variable the Java activity is working with could be manipulated by another workframe before the Java activity is finished.

Another intricacy of Brahms semantics is that Objects are aware of changes to the world state as soon as they happen, since they operate according to facts. Agents on the other hand would need these updates communicated to them in order to update their beliefs to reflect the facts, or they would need to update the world state themselves. So in reality, the Cashier object does not need to communicate to the Order after processing it, the Order is updated immediately upon completion; the communication is just a formality.

3.3 Restrictions

This thesis assumes Brahms models with these properties:

- There is no dynamic agent or object creation, so all agents and objects are known statically.

- Objects' workframes are all defined as fact frames, so objects always act only according to facts in the world.
- Neither agents nor objects have workframes with detectables, so no workframe will be interrupted by the execution of another workframe.
- Java activities are synchronized so there are no competing memory accesses within the activity
- When a workframe must reference itself, it uses the *current* keyword rather than binding to itself.
- Concepts are location-less, so broadcast activities communicate to all concepts in the system.

Even with these restrictions, Brahms is able to model complex and interesting systems, such as Departure Sensitive Arrival Spacing (DSAS)[11], which models air traffic around LaGuardia and increases departure throughput during their peak traffic hours. Brahms is also used to model Air Traffic Management in [17].

Chapter 4

Workframe Coverage

Workframe coverage is similar to statement coverage in software verification. For a workframe to activate, all of the preconditions of the workframe must be met. Each precondition is a statement for the "statement coverage." The preconditions illustrate different states of the model, or different beliefs of the agents and facts in the world. When a workframe activates, we know we have observed at least one state of the model that can activate that workframe. We cannot make any assumptions about the covered state space, because it might be infinite. Workframes that are not activated in a simulation run reveal that some states of the model are as yet unvisited. When every workframe is activated at least once, 100% workframe coverage is achieved.

4.1 Creating the workframe coverage requirement

In order to create the coverage requirements, we wrote a parser that decomposes the model code into pieces that can be analyzed. The parser decomposes the model code into pieces to be analyzed, including a set of concepts, instances, activities, workframes, and variables.

$$C = \text{ set of all concepts}$$

$$I = \text{ set of all instances}$$

$$A = \text{ set of all activities}$$

$$WFS = \text{ set of all workframes}$$

$$V = \text{ set of all variables}$$

Static analysis takes those pieces and determines what is needed for the coverage requirements. Because the Brahms grammar requires the “workframe” distinction before every workframe, workframes are easily identifiable.

Workframes are defined in group and class files most often, as these are the most general cases, but agent and object instances can have workframes that are unique to them as well. The workframe coverage requirement includes all workframes, the general and the specific.

The parser creates a map $WF : C \mapsto 2^{WFS}$ that maps each concept to the workframes of that concept, a subset of WFS. We define the set of workframe requirements RWF as:

$$RWF = \{(c, wf) \mid \exists c \in C, wf \in WF(c))\}$$

The concept name c will most often be the class or group name. Only in the cases where instances have workframes that are unique to them will c refer to an instance in the model. If an agent or object instance overwrites its parent workframe, then a workframe coverage requirement would be created for the instance itself in addition to the workframe coverage requirement for the parent concept. The workframe coverage requirements from the listings in chapter 3 would be *(Order, wf_ alertCashier)*, *(Cashier, wf_ workOnOrder)*, and *(Cashier, wf_ clean).*

The parser also creates a concept hierarchy for later coverage calculation. The hierarchy maps agents and objects back to their parent group(s) or class(es). This has direct correspondence to class diagrams. The general Concept can be broken down into two groups, Classes and Groups, which can be broken down into individual Objects and Agents. Agents can be members of multiple groups, but Objects can be instances of only a single class.

4.2 Instrumenting the code

Logging statements are included in every workframe to identify the workframes that have been activated. The instrumentation is of the form:

$$printToLog(String\ workframeName, Concept\ caller, String\ callerType, String\ loggerName)$$

The name of the workframe is passed in as well as the caller of the workframe, the type of the caller (its parent group or class), and the name of the logger, in this case, “workframeLogger”. The *callerType* is an addition for models in which objects or agents are created dynamically. This isn’t currently supported or explored in this work, but is ready to be implemented in the future. The Brahms code passes the parameters through to the Java activity which formats the output and sends it to an output file. The logged output is of the form *(ConceptType.InstanceName, WorkframeName)*.

The Cashier class from chapter 3 would be instrumented as shown in listing Listing 4.1.

As a result of running the simulation, the log would show *(Cashier.Bob, wf_ workOnOrder)* and *(Cashier.Bob, wf_ clean)* if the agent Bob had activated the two workframes *wf_ workOnOrder* and *wf_ clean*. See the log in Listing 4.2 for a run of the scenario given in chapter 3.

Listing 4.1: Example Brahms Group: Cashier

```
group Cashier {
   attributes: ...
   activities:  ...
      java printToLog(string message, Concept concept1, string concept2, string
          logName) {
            class: "BrahmsAccessToLogger";
      }
   workframes:
      workframe wf_workOnOrder {
         priority: 1;
         variables:
            foreach(Order) order;
         when (
            knownval(order.completed = false)
         )
         do {
            printToLog("wf_workOnOrder", current, "Cashier", "workframeLogger");
            workOnOrder();
            conclude((order.completed = true));
            processOrder(order);
         }
      }
      workframe wf_clean {
         variables:
         when ( )
         do {
            printToLog("wf_clean", current, "Cashier", "workframeLogger");
            clean();
         }
      }
}
```

Listing 4.2: Basic Cash Register workframe logged output

```
(Cashier.Bob, wf_workOnOrder)
(Order.Order_2, wf_alertCashier)
(Order.Order_3, wf_alertCashier)
(Order.Order_2, wf_alertCashier)
(Order.Order_3, wf_alertCashier)
(Order.Order_3, wf_alertCashier)
(Order.Order_2, wf_alertCashier)
(Cashier.Bob, wf_workOnOrder)
(Order.Order_3, wf_alertCashier)
(Order.Order_3, wf_alertCashier)
(Cashier.Bob, wf_clean)
```

4.3 Workframe coverage calculation

The workframe coverage calculator takes the output from the parser (the workframe coverage requirement list, *RWF*) as well as the workframes logged during the simulation *WFL* and calculates the total coverage for that particular simulation of the model. We compute coverage calculation as the ratio of coverage over requirements.

$$\frac{|(\mathit{WFL} \cap^* \mathit{RWF})|}{|\mathit{RWF}|}$$

Because the *WFL* is in terms of instances and *RWF* is in terms of concepts, we do not perform a true intersection of those two sets. We must ensure that the instance in the *WFL* is of the type that *RWF* requires. The logged output is searched for each workframe coverage requirement by the workframe name. If the workframe name is found, the workframe's caller is matched with the caller type from the coverage requirement. If the caller is of the required type, the workframe coverage requirement is met. The type of the caller is checked by the concept hierarchy the parser generated.

Once a satisfying instance of the workframe coverage requirement is found, it is added to a list of satisfying instances, and the counter for that instance goes up. This information is stored in a workframe coverage details file, which can be used to guide manual test creation.

Listing 4.3: Basic Cash Register: workframe coverage details file

```
(Cashier, wf_clean):
    (Cashier.Bob, wf_clean): 1
(Cashier, wf_workOnOrder):
    (Cashier.Bob, wf_workOnOrder): 2
(Order, wf_alertCashier):
    (Order.Order_2, wf_alertCashier): 3
    (Order.Order_3, wf_alertCashier): 5
Coverage Score (3/3): 1.0
```

It is clear to see in Listing 4.3 that 100% coverage is achieved; thus the scenario presented in chapter 3 was sufficient. In this example, you can see that Order_1 never alerted the cashier because it was already completed, but that two other Orders did so.

Even if only one instance meets the coverage requirement, we still consider the workframe coverage requirement to be satisfied. We are not interested in seeing every instance activate every workframe. This is not plausible. One instance satisfying the workframe is enough according to the *value independence hypothesis* [22], which states that a majority of concurrency bugs are triggered if the memory dependencies are exposed, no matter what the data values of the shared variables are.

Chapter 5

Communication Coverage

Communication activities within workframes of agents and objects are used to test the coverage of the bindings that occur in workframes. It is not the communication activities themselves that quantify the simulation effort; were that the case, workframe coverage would be sufficient since communication activities are called within workframes. What we are interested in is the bindings that must occur for communication to take place. Workframes can have the option to bind to multiple variables, and we are interested in all those bindings. Communication coverage requires all the possible bindings, and uses communication activities to report the bindings that occurred.

5.1 Static Analysis

In order to cover these communication activities, each possible communication needs to be listed. To keep with the basic cash register example presented in chapter 3, the Cashier has a communication activity with an Order (*processOrder*). If there are three Order instances in the model, then a communication between Bob the Cashier and all three Orders are required.

In order to create all these coverage requirements, the parser creates a map from a concept to its communication activities, as well as maps from an instance to its parent concept(s), and from a communication activity to the concept type it communicates with.

$$CA : C \mapsto 2^{A}$$

$$\mathit{concept} : I \mapsto C$$

$$\mathit{with} : CA \mapsto C$$

At the end of the analysis, each instance of a concept that has communication activities is paired with every instance of the concept it communicates with to create the set of communication requirements *RCM*.

$$\mathit{RCM} = \{(i, ca, i') \mid \exists ca \in \mathit{CA}(\mathit{concept}(i)), \mathit{concept}(i') \in \mathit{with}(ca)\}$$

Communication activities involve two agents, but they are not reflexive. An agent or object A communicating with an agent or object B is different from the agent or object B communicating with agent or object A. The list of coverage requirements is built by every possible communication that takes place between instances. An element on this list is a tuple with the unique names of the communicating instances, and the communication activity that linked the two instances: (i, ca, i'); the first listed instance is the instance that initiated the communication. See Listing 5.1.

Listing 5.1: Basic Cash Register: communication requirements

```
(Order_1, alertCashier, Bob)
(Order_2, alertCashier, Bob)
(Order_3, alertCashier, Bob)
(Bob, processOrder, Order_2)
(Bob, processOrder, Order_3)
(Bob, processOrder, Order_1)
```

5.2 Instrumentation

In order to know which bindings took place, the Brahms code is instrumented to output the names of the agent or object initiating the communication, as well as the agent or object with whom the communication was initiated. For each workframe, there will only be a pair of concepts that communicate with one another. The Java activity that handles the logging is of the form:

$$printToLog(String\ commName, Concept\ concept1, Concept\ concept2, String\ loggerName)$$

Where *commName* is the name of the communication activity, *concept1* is the agent or object initiating the communication activity, *concept2* is the agent or object with which *concept1* is communicating, and *loggerName* refers to the name of the logger handling this information, in this case “communicationLogger”. See Listing 5.2.

Listing 5.2: Basic Cash Register: communication instrumentation

```
class Order {
    attributes: ...
    relations: ...
    activities: ...
        java printToLog(string message, Concept concept1, Concept concept2, string
             logName) {
                class: "BrahmsAccessToLogger";
        }
    workframes:
        workframe wf_alertCashier {
            repeat: true;
            variables:
                forone(Cashier) cashier;
            when(
                knownval(current.completed = false) and
                unknown(current completedBy) and
                known(cashier.name)
            )
            do {
                alertCashier(cashier);
                printToLog("wf_alertCashier", current, cashier, "communicationLogger");
            }
        }
}
```

The communication logger in the Java activity then takes the information passed to it and formats a logged output of the form *(concept1Name, commName, concept2Name).* For the scenario given in chapter 3, the logged output looks like Listing 5.3.

Listing 5.3: Basic Cash Register: communication output

```
(Order_2, alertCashier, Bob)
(Order_3, alertCashier, Bob)
(Order_2, alertCashier, Bob)
(Order_3, alertCashier, Bob)
(Bob, processOrder, Order_2)
(Order_3, alertCashier, Bob)
(Order_2, alertCashier, Bob)
(Order_3, alertCashier, Bob)
(Order_3, alertCashier, Bob)
(Bob, processOrder, Order_3)
```

5.3 Coverage Calculation

In order to calculate the communication coverage, we parse the logged output and build a set *CML*, the logged communication activities, and calculate the coverage as the ratio of coverage over requirements.

$$\frac{|(CML \cap RCM)|}{|RCM|}$$

The communication coverage calculator is the simplest of all the coverage calculators. Because the communication coverage requirement list lists all possible communication pairs by their instance, and because the logged output logs the same format as the coverage requirement, the logged output can simply be intersected with the communication coverage requirement list.

A coverage details file is created with each communication coverage requirement mapped to how many times it was met by statements in the log. An overall coverage score is calculated and reported at the end of the file. See Listing 5.4 for an example.

Listing 5.4: Basic Cash Register: communication coverage details file

```
(Bob, processOrder, Order_1): 0
(Bob, processOrder, Order_2): 1
(Bob, processOrder, Order_3): 1
(Order_1, alertCashier, Bob): 0
(Order_2, alertCashier, Bob): 3
(Order_3, alertCashier, Bob): 5
Coverage Score (4/6): 0.6666666666666666
```

Chapter 6

Schedule Coverage

The concurrent nature of multi-agent systems makes schedule coverage an especially interesting problem. Workframes that share variables can experience data-race. Data-race in these models means that as the workframe activation order changes, so does the behavior of the model. It is useful to test these workframes with shared variables in each possible order to ensure that no bugs are introduced when the scheduling of the workframes is changed. A schedule coverage requirement consists of a schedule of two workframes that experience data-race. There are two types of schedule coverage: intra-concept and inter-concept. Intra-concept schedule coverage deals with two workframes within a single instance that have data-race. Inter-concept schedule coverage deals with two workframes from separate instances that have data-race.

6.1 Schedule Coverage Requirement

In order for the coverage of the MAS to be calculated, the schedule coverage requirement needs to be defined. We use the hypothesis that the majority of bugs can be found between two threads [13], so we only look at pairs of workframes. In order for two workframes to be considered "interesting" and to make the schedule coverage requirement list, they need to meet two requirements:

- The two workframes must be able to be active at the same time
- The two workframes must manipulate the same variables.

Here, "manipulate" means that at least one of the workframes is writing the shared variable. To find workframe pairs that meet these requirements we use static analysis.

6.1.1 Workframe Activation Order

Workframes are activated according to their preconditions. If there are multiple workframes whose preconditions have been met, they are ordered according to their priority. Workframes with higher priority are activated first, followed by workframes with lower priority. If the workframes are the same priority, Brahms will activate them in a random order.

Intra-Instance Schedule Coverage

For a single concept instance, two workframes can only be concurrently activated if they have the same priority. We are able to determine whether any two pairs of workframes share the same priority from the priority declaration of the workframe from the Brahms files.

Inter-Instance Schedule Coverage

For any two instances, workframes can be concurrently activated no matter what their priority is. This is because workframes are activated on a per-concept basis since each concept has its own thread. All workframes of a single instance are running separately from all workframes of any other instance. So priority is not a consideration when pairing workframes for inter-instance schedule coverage.

6.1.2 Variable Manipulation

Workframes bind to different variables via the *forone*, *collectall*, and *foreach* keywords. They can also use the *current* keyword to "bind" to their own attributes, which are variable in the context of workframes. The binding takes place in the precondition of the workframe. If the variable satisfies the preconditions, it can be bound to a workframe activation. For an agent, the satisfying variables in the precondition reflect the beliefs of that agent. For an object, the satisfying variables in the precondition reflect the facts in the world.

Once bound, these variables can be passed into activities within the workframe and can be part of conclude-statements, where the variables are updated. There are two types of variables within a workframe, variables that are read and variables that are written. As a part of the preconditions, the variable is considered a "read" variable. As part of a conclude-statement or communication activity, the variable is considered a "write" variable. There are also variables that are read and written as part of Java activities, but these variables are ignored for the scope of this thesis.

Preconditions

Read variables are the variables in the preconditions. These variables are read when a workframe attempts to activate. The parser adds the variables in the preconditions to the read set of a workframe. These variables are beliefs (or facts) about other agents and objects, or about the concept itself.

In Listing 6.1 there is only one read variable: *order.completed*, which is the *completed* attribute of the bound Order.

Conclude-statements

Workframes update or write variables as part of conclude-statements. Variables in conclude-statements are added to the write set of a workframe. Adding every variable that is concluded over-approximates the actual written variables in the case of agents. Objects are always aware of conclude statements because they update facts in the world. Agents on the other hand are only made aware of the variable manipulation in a conclude statement if they are concluding the variable themselves, or if it is communicated to them. So the variables written in a conclude statement are only visible to objects, but we still include the variables written in conclude statements in the write set for the workframe.

In Listing 6.1, there is one write variable in a conclude statement: *order.completed*.

Communication activities

Concepts communicate their variable updates to update the beliefs of other agents. Send communication activities update the variables of the agents they are communicating with, and receive communication activities update the beliefs of the initiating concept. When there is a communication activity in a workframe, the variables being communicated are added to the set of write variables for that particular workframe. This is an unnecessary addition for objects since in order for an updated variable to be communicated, it must be updated in a conclude-statement first.

It is still necessary to treat communication activities as separate writes because they can communicate information about the initiating instance itself, even if the instance did not update the communicated variable in that particular workframe.

In Listing 6.1 there is one communication activity, *processOrder*, which sends the value of *order.completed* as it was concluded in the line before the communication takes place.

Listing 6.1: Example workframe: read and write variables

```
workframe wf_workOnOrder {
    priority: 1;
    variables:
        foreach(Order) order;
    when (
        knownval(order.completed = false)
    )
    do {
        workOnOrder();
        conclude((order.completed = true));
        processOrder(order);
    }
}
```

Java activities

Java activities can both read and write variables of the model via the Java API. These variables are hard to keep track of because they don't need to be passed in to the activity as parameters and can go beyond the binding of the workframes. For the scope of this thesis, we ignore variable manipulation in Java activities.

6.1.3 Schedule Requirement Format

In addition to the sets described in chapter 4 and chapter 5, parsing returns two maps that each map a workframe to a set of reads and a set of writes (each subsets of the variable set), as well as a map from a workframe to its priority.

$$R : \mathit{WFS} \mapsto 2^V$$

$$W : \mathit{WFS} \mapsto 2^V$$

$$P : \mathit{WFS} \mapsto \mathbb{N}$$

Using the sets and maps returned by the parser, we define a function *conflict* that takes two workframes as input, and determines if they share a variable. In essence, *conflict* approximates data-race.

$$\mathit{conflict}(\mathit{wf}_1, \mathit{wf}_2) = \begin{cases} 1 & \text{if } (R(\mathit{wf}_1) \cap W(\mathit{wf}_2)) \cup \\ & \quad (W(\mathit{wf}_1) \cap R(\mathit{wf}_2)) \cup \\ & \quad (W(\mathit{wf}_1) \cap W(\mathit{wf}_2)) \neq \emptyset \\ 0 & \text{otherwise} \end{cases}$$

The schedule requirement list RS is made up of the intra-concept RAC and inter-concept ERC schedule requirements $RS = RAC \cup ERC$. An intra-concept schedule coverage requirement is defined as follows: $(c.\mathit{wf}_\alpha, c.\mathit{wf}_\beta) \in RAC$ iff

- $\exists \mathit{wf}_\alpha, \mathit{wf}_\beta \in \mathit{WF}(c)$
- $conflict(\mathit{wf}_\alpha, \mathit{wf}_\beta)$
- $P(\mathit{wf}_\alpha) = P(\mathit{wf}_\beta)$

An inter-concept schedule requirement is similar to the intra-concept coverage requirement, except that the two concepts calling the workframe are not the same, and the priorities of the workframes do not matter. $(c_1.wf_\alpha, c_2.wf_\beta) \in ERC$ iff

- $\exists wf_\alpha \in WF(c_1), \exists wf_\beta \in WF(c_2)$
- $conflict(wf_\alpha, wf_\beta)$
- $c_1 \neq c_2$

The schedule coverage requirement is of the form $(c_1.wf_\alpha, c_2.wf_\beta)$ in addition to its complement $(c_2.wf_\beta, c_1.wf_\alpha)$, so that both orders of the workframe pair are required. In the case of intra-instance schedule coverage, $c_1 = c_2$. There is also an intricacy for inter-concept schedule coverage when there are two different instances of the *same* concept that have a workframe pair. In order to distinguish c as two different instances, *_1* and *_2* are appended to the concept name.

For the scenario presented in chapter 3, there is only one schedule coverage requirement: *(Cashier.wf_workOnOrder, Order.wf_alertCashier)* and its complement *(Order.wf_alertCashier, Cashier.wf_workOnOrder)*. This is because *wf_workOnOrder* writes the variable *Order.completed*, and *wf_alertCashier* reads that variable.

Note that we are concerned with the schedule coverage at the general concept level, not for every instance. As long as one instance (in the case of intra-instance schedule coverage) or two instances (in the case of inter-instance schedule coverage) meet the requirement, we are not interested in any other instances that meet the requirement. This in accordance with the hypothesis in [22] which states that not every instance needs to be explored in order for a bug to be found.

6.2 Instrumentation

In order to calculate the schedule coverage after a simulation, we need to know when variables are being read and when they are being written. Because we didn't have a way of accessing the Brahms simulation internally, we had to come up with a work-around solution involving instrumenting the Brahms code to create meaningful log files that can later be parsed for coverage calculation.

The logged output is of the form *(time:caller.wfName,variable)*. This output goes to a log file which is then passed to the coverage calculator.

6.2.1 Variable Logging

Because the concurrent nature of Brahms results in reads and writes of one workframe intertwining with the reads and writes of other workframes, it is impossible to associate the manipulated variables with the appropriate workframe from a log since the JAPI has no way of knowing the parent workframe of a variable being accessed. For this reason, we could not keep track of the variables individually.

When creating the schedule coverage requirements, the parser creates lists of read and write variables for each workframe. We know that if one of these workframes is logged, then the variables that workframe maps to were read or written as the case may be. So rather than variable logging, we focused on workframe logging. We treat the workframes as atomic blocks; even though that is not the case, it is a better approximation than none at all. It is important to note that even if the log shows an order of workframes, it does not mean the frames were not interleaved when they appear in the same time step.

6.2.2 Workframe Logging

Because workframes are always trying to activate by binding with variables so that the preconditions are satisfied, it is impossible to know exactly when the preconditions of a workframe (read variables) are being read. We instead settle for a logged output as soon as the workframe is activated. At that point we know the variables in the preconditions have all been read. But the name workframe and the concept that called the workframe is not enough to know what variables were actually read when there is a binding involved. To solve this, we also pass a variable binding to the log.

The instrumentation is of the form:

$$printToLog(String\ wfName, Concept\ caller, Concept\ binding, String\ logger)$$

The name of the workframe is passed in, the caller of the workframe is passed in, a variable binding is passed in, as well as the name of the logger (in this case, "scheduleLogger"). Because not every workframe will bind to a different concept, you can pass in the caller again as the second concept, which will take care of that case. We assume that the workframe does not bind to itself, but rather uses the *current* identifier in Brahms, as per the restrictions in chapter 3. The name of the logger is passed in to differentiate the schedule report from the workframe and communication reports, since only one Java activity was necessary to handle all the logging. See Listing 6.2 for an example of how to instrument the code, and Listing 6.3 for logged output for the scenario given in chapter 3.

Listing 6.2: Basic Cash Register: schedule coverage instrumentation

```
group Cashier {
    attributes: ...
    activities: ...
        java printToLog(string message, Concept concept1, string concept2, string
             logName) {
                class: "BrahmsAccessToLogger";
        }
    workframes:
        workframe wf_workOnOrder {
            priority: 1;
            variables:
                foreach(Order) order;
            when (
                knownval(order.completed = false)
            )
            do {
                printToLog("wf_workOnOrder", current, order, "scheduleLogger");
                workOnOrder();
                conclude((order.completed = true));
                processOrder(order);
            }
        }
        workframe wf_clean {
            variables:
            when ( )
            do {
                printToLog("wf_clean", current, current, "scheduleLogger");
                clean();
            }
        }
}
```

Listing 6.3: Basic Cash Register: schedule logging output

```
0:(Bob.wf_workOnOrder,Order_2)
0:(Order_3.wf_alertCashier,Bob)
0:(Order_2.wf_alertCashier,Bob)
2:(Order_2.wf_alertCashier,Bob)
2:(Order_3.wf_alertCashier,Bob)
4:(Order_3.wf_alertCashier,Bob)
4:(Order_2.wf_alertCashier,Bob)
5:(Bob.wf_workOnOrder,Order_3)
6:(Order_3.wf_alertCashier,Bob)
8:(Order_3.wf_alertCashier,Bob)
10:(Bob.wf_clean,)
```

6.2.3 Timing the logged output

The logging unfortunately doesn't let us know when exactly something happened. So we need to know the time at which the workframe was called. This is not passed through as a parameter to the activity, but is rather accessed through the JAPI. The JAPI accesses the internal clock of Brahms so we can know which workframes are executing in which Brahms time steps.

6.3 Coverage Calculation

The coverage calculator takes in several different inputs: the output file from the instrumented simulation, the list of schedule coverage requirements, and the concept hierarchy from the parser.

6.3.1 Satisfying Instances

Schedule coverage requirements are in a very general form because we're not concerned with each instance in the model meeting all the schedule requirements with every other instance. When a workframe that is part of a schedule requirement is output, we first need to make sure that the instance calling that workframe is of the proper type. This is where we use the concept hierarchy. The schedule coverage calculator uses this hierarchy to ensure that the caller of the satisfying workframe matches the caller type of the requirement. If the workframe and the caller match, it is a satisfying instance of the schedule coverage requirement.

6.3.2 Bindings

In order for the schedule coverage requirement to be met, the bindings of the two satisfying workframes must match. If a workframe has no variable bindings that matter, and therefore no reported binding, then we know that it is only bound to itself. Any workframe that has no reported bound variables will manipulate its own attributes, or else it has no bearing on the state of the model at all. For a subsequent workframe to "match," it must bind to the instance that calls the first workframe, or be the same instance that calls the first workframe.

6.3.3 Timing

Since we don't have access to the Brahms simulator, we cannot know when exactly code is executing, even with the log statements. The purpose of the log statements is to know when variables are read and written, but it is possible that by the time the log statement has executed, those variables may have already had an intervening write. So within the same time stamp, if there are multiple workframes that manipulate the same variables, there is no way to know in what order they occurred. Because schedule coverage is concerned with the order of the workframes, we have to be very careful when we find the satisfying instances of the schedule coverage requirement.

If the program reported satisfying instances where none existed, it would lose validity. On the other hand, if it misses satisfying instances because it is too restrictive, the validity is still preserved. The results of the coverage metric would not be correct, but at least it would still search for those satisfying instances, rather than "finding" them and potentially missing bugs. In order to avoid this, the coverage calculator for schedule coverage must ensure that no two workframes that manipulate the same variable appear in the same time stamp. The first satisfying instance of the schedule coverage requirement must share any variables with the other workframes in the same time stamp. The log file is parsed by time stamp, so we obtain a map that maps an integer representing the time stamp to the set of workframes and their instances that appear in that time stamp.

$$T : \mathbb{N} \mapsto 2^{I}.2^{WF(concept(I))}$$

Once the first satisfying workframe instances is found, the second satisfying instance can be searched for. If there are any intervening writes, we know that the first instance that was found is not going to satisfy the schedule coverage requirement. Similarly, if the second satisfying instance to the workframe is found in a time stamp where there is another workframe that shares the same variable, that second satisfying instance cannot be counted for the schedule coverage requirement. In order for a schedule coverage requirement to be met, the first workframe must appear in a timestamp where there are no other conflicting workframes. The second workframe must appear in a subsequent timestamp and match the bindings of the first workframe, and there cannot be an intervening write in any timestamps between the first and second workframes: $met(i_1.wf_\alpha, i_2.wf_\beta) = 1$ iff $\forall k \ni i \leq k < j$

- $i_1.wf_\alpha \in T(i) \wedge$
- $\forall i.wf \in T(i),\ \neg conflict'(i_1.wf_\alpha, i.wf)$
- $\forall i.wf \in T(k),\ \neg conflict'(i_1.wf_\alpha, i.wf)$
- $\exists! i.wf' \in T(j),\ conflict'(i_1.wf_\alpha, i.wf')$
- $i.wf' = i_2.wf_\beta$

$$met(i_1.wf_\alpha, i_2.wf_\beta) = 0 \text{ otherwise}$$

The *conflict*$'$ function goes a step beyond *conflict* in that it takes the instance and its associated bindings into account.

We search for each satisfying instance of the schedule coverage requirement by time stamp. If one time stamp doesn't have a satisfying instance, we move on to the next one. See Listing 6.4 for the schedule coverage calculator pseudocode.

Listing 6.4: Schedule coverage calculator psuedocode

```
calculateCoverage(String logFile, String covList){
    For every time stamp, get each logged line
    For every requirement in the covList, get each wf that shares variables
    For every coverage requirement req
        For every time stamp i
            If the first half of the req is found (with no other competitors)
                for every time stamp i+1
                    if the second half of the req is found (with no other competitors)
                        the requirement is satisfied
                        add the found requirement to the details file
                        break
                if the requirement is satisfied
                    break
}
```

6.3.4 Details File

These satisfying instances are stored in a schedule coverage requirement details file, and are used in the overall calculation of schedule coverage.

$$\frac{\sum_{(i.wf,i'.wf')\in RS} met(i.wf,i'.wf')}{|RS|}$$

If a schedule requirement is met once, we do not seek to meet it again with other instances, according to the value independence hypothesis [22]. The file in Listing 6.5 is an example of the details file for the scenario from chapter 3. It is especially helpful in seeing how this coverage metric was met when looking at Listing 6.3.

Listing 6.5: Basic Cash Register: schedule coverage details file

```
(Cashier.wf_workOnOrder, Order.wf_alertCashier):
    (Bob.wf_workOnOrder, Order_3.wf_alertCashier)
(Order.wf_alertCashier, Cashier.wf_workOnOrder):
    (Order_3.wf_alertCashier, Bob.wf_workOnOrder)
Coverage Score (2/2): 1.0
```

Chapter 7

Empirical Study

To understand the usefulness of these metrics, we ran a few experiments on several small models we wrote and the existing model that ships with the Brahms code (ATM model). We wanted to understand if it is hard to instrument existing models for coverage, how much slowdown during simulation is caused by the instrumentation, if the metric shows anything other than 100% coverage, and if it is possible to increase coverage with more runs or more scenarios. These questions help us to learn how useful and viable the coverage metrics are.

7.1 Model Descriptions

We wrote several small models to perform our tests on. Tables 7.1 and 7.2 help us to understand the complexity of each of these models. Since there isn't a real measure for the complexity of Brahms models, we used the metrics in the tables to illustrate the complexity. Table 7.1 reports the number of workframes, attributes, and communication activities for each model, as well as the average number of read variables and write variables for each workframe.

Another good complexity metric is the number of generated coverage requirements for each model. Table 7.2 shows the number of workframe, communication, and schedule coverage requirements generated by the Parser. Note that the number of workframe coverage requirements is the same as the number of workframes in the system, and that the schedule coverage requirement list can be quite extensive. The number of schedule coverage require-

Table 7.1: Model details for empirical study

Model Details					
Model	Workframes	Attributes	Comm. Activities	Av. Read Variables	Av. Write Variables
Basic Cash Register	3	4	2	1.33	0.67
Roommates	1	2	1	2.0	2.0
Objects	2	2	0	2.5	1.5
Agents	4	2	1	2.5	0.75
ATM	39	75	10	8	4.31

Table 7.2: Requirements for each coverage metric

Model Coverage Requirements			
Model	Workframe	Communication	Schedule
Basic Cash Register	3	6	2
Roommates	1	4	0
Objects	2	0	2
Agents	4	4	14
ATM	39	40	494

ments for ATM tells us that there are a lot of dependencies and potential for data-race in the ATM model.

7.2 Instrumentation

We wanted to know how hard it is to instrument existing models. Each workframe in each model required at least two log statements, one for workframe coverage, one for schedule coverage. Workframes with communication activities required at least one log statement for each call to the communication activity. The number of log statements each model needs is n where $2 * wfs \leq n \leq 2 * wfs * commActs$. The worst case scenario is that every workframe calls every communication activity. For a model like ATM with 39 workframes and 10 communication activities (see Table 7.1), instrumenting the code was tedious. For the smaller models, it was relatively easy. The best approach is to instrument the model as it is being written.

Table 7.3: Running time for each model

Model running time (in milliseconds)		
Model	With Logging	Without Logging
Basic Cash Register	6.92	5.28
Roommates	5.20	5.31
Objects	5.31	5.37
Agents	5.29	5.19
ATM	10.74	9.875

In order to see if the instrumentation had any effect on the runtime of the simulation, we ran each model without the logging statements, and with the logging statements at least 20 times. Each simulation run could have a different running time because of the nature of multi-agent systems, so we reported the average. The scenarios were kept consistent for each run. The results of this experiment are in Table 7.3.

Including the log statements for the instrumentation doesn't have too much of an effect on the running time of the models. The Objects and Roommates models actually ran faster on average with the log statements. This we presume just implies that the effect of the instrumentation is negligible for these two models.

The biggest difference in running time was for the Basic Cash Register model, a difference of 1.64 seconds. The more complicated ATM model had a difference of 0.86 seconds. We concluded that even though that increase in running time is not negligible, the time that the coverage metrics save by providing information to generate better tests far outweighs the extra second or two gained by running the model without the instrumentation.

7.3 Coverage Results

To get a sense of the usefulness of each coverage metric, we wrote typical scenarios for each model and ran each one through our tool. The reported coverage for most of the models was consistent across all runs of the scenario. The coverage score reported with an asterisk is an average across the 20 or more runs of the single scenario. See the scores in Table 7.4. We know that coverage can be improved by running a single scenario more than once, because

Table 7.4: Results for each coverage metric

Model Coverage Calculation			
Model	Workframe	Communication	Schedule
Basic Cash Register	1.0	0.6667	1.0
Roommates	1.0	1.0	NA
Objects	1.0	NA	0.0
Agents	1.0	1.0	0.3047*
ATM	0.9923*	0.439*	0.5876*

agents in one run of a scenario may make different choices in another run of the same scenario. Coverage can also be improved by writing new scenarios that take the coverage metrics into account and try to improve upon them.

7.3.1 Workframe coverage results

It was easy to achieve 100% workframe coverage on all the models. We attribute this to the fact that we only care about general concepts activating every workframe rather than every instance activating every workframe. The ATM model was the only model for which the one scenario didn't cover all its workframes in the 20 simulation runs. Of the 20, only one run didn't cover every workframe. We were able to achieve 100% schedule coverage for the ATM model just by running the simulation multiple times.

7.3.2 Communication coverage results

Total communication coverage was also easily achievable with our simple models. The basic cash register model did not achieve 100% communication coverage because the scenario we had written precluded any communication for one of the objects. With one tweak, we were able to give that object a more active role in the scenario and achieve 100% coverage. This will not always be the case. There will be models whose scenarios necessarily preclude certain communication activities from happening. This is particularly true for models that have workframes for concepts that are conditional.

For example, the SATS model discussed in the next chapter involves several planes in an airspace over time. Planes that are active in the scenario at the beginning of the simulation may "land" and therefore will no longer communicate with planes that are flying. The plane that landed early in the scenario may never have the chance to communicate with a plane that entered the airspace later in the simulation run.

Because the communication coverage requirements require communication between every pair of instances rather than the general case, the communication coverage score has the potential to be the worst of the three scores. This proves to be the case for the ATM model. After inspecting the communication coverage details files for the ATM model, it was clear that the way the scenario was written 100% communication coverage was not possible. The model involved many communications between a bank and a student. This particular scenario involved two student instances and two bank instances, and each student was a member of only one bank. The parser creates requirements for communication between both students and both banks; because each student was only a member of one bank, the student will not communicate with the other bank in the model. Because of this our best communication coverage score for the 20 runs of this scenario was .5, half of the communication activities.

The communication coverage for the ATM model could be improved if the scenario was changed to involve only one student instance and one bank instance. Then we can achieve 100% communication coverage. Of course, with only one student instance and one bank instance the model interactions would not be as interesting. The more interesting the model, the less likely 100% coverage would be.

7.3.3 Schedule coverage results

Only one model in our study reported 0 for schedule coverage. The score for the Objects model was so low because both workframes in the model (there are only two) always appeared together in the same timestamp. Since there is no way of knowing the ordering of workframes within a timestamp, the schedule coverage calculator could not use the two workframes to satisfy the schedule coverage requirements.

Intervening writes is the primary culprit of poor schedule coverage. The only way around this is to access the Brahms scheduler, which would need to be done from a custom simulation tool. This was not within the scope of this thesis, but if implemented would greatly improve schedule coverage for models as it would remove the restriction that satisfying instances of schedule coverage cannot be found in the same time stamp.

Another factor for poor schedule coverage is the over-approximation of schedule coverage requirements. The list of schedule coverage requirements is composed of workframe pairs that have the potential to share variables, but the actual simulation run may not share variables at all because they are bound to different instances.

The parser also adds workframe pairs to the list of schedule requirements even if at runtime they are strictly ordered. For example, in the ATM model, one of the generated schedule requirements is *(Atm.wf_ askPin, Student.wf_ insertBankCard)*. For a proper ATM, this order of workframes would never occur. If this schedule was covered, that would reveal strange model behavior that would need to be corrected.

Despite over-approximating the possible schedules of a system, the schedule coverage metric can still reveal strange model behavior, and help us to write better scenarios that attempt to cover more schedules.

Chapter 8

Case Study

In order to evaluate the contribution of the coverage metric defined in this thesis, we conducted a case study. We wanted to see how the coverage metric would be used practically, especially during the development of the model.The model for the case study had to have some degree of complexity, so we decided to use NASA's SATS concept of operations.

8.1 SATS Concept

The Small Aircraft Transportation System (SATS) [5] was developed in 2004 to increase throughput of aircraft in small airports without communication towers during instrument meteorological conditions—conditions in which the pilots rely on their instruments rather than their eyes. It is a system of data communications between pilots, planes, Air Traffic Control, and the Airport Management Module (AMM). The AMM is an automated unit at the airport that monitors the Self Controlled Area (SCA) and manages the order in which planes can enter the airspace by withholding or granting clearance [15].

The SCA is an airspace volume that includes up to 3000ft above the airport. See Figure 8.1 for an example SCA including sample fixes. A fix is a 2D point that aids aircraft in navigation.

Once the planes have entered the SCA, there is no manned communication tower to help them maintain safe separation. Pilots are responsible for their own separation in the airspace around the runway (the SCA), which can be difficult during instrument meteorological conditions. SATS provides rules of entry that help maintain safe separation.

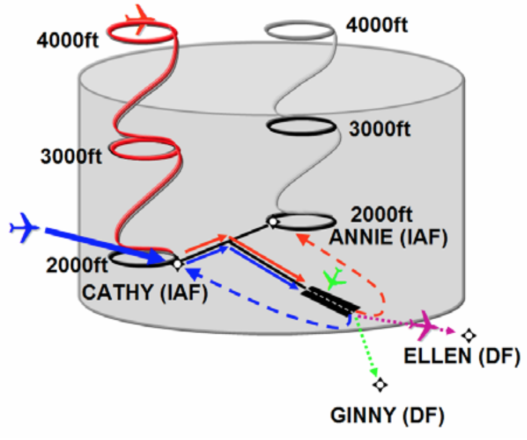


Figure 8.1: Sample SCA

These rules of entry are defined by flight procedures, on-board navigation, and the AMM. SATS aims to accommodate multiple aircraft in the SCA during instrument meteorological conditions.

8.1.1 SATS walk-through

As a plane approaches the SCA, it seeks permission from the AMM to enter. If there are no other planes in the SCA, the AMM allows the plane to enter and assigns it a Missed Approach Holding Fix (MAHF) that corresponds to the side it entered on. If there is another plane or more in the SCA, the AMM grants entry to the plane only if there are no aircraft at the initial approach fix (IAF) on that side, and that no planes have been assigned MAHF on that side. If these requirements are not met, the plane must reroute and hold above the SCA.

Planes that must hold above the SCA do so at the lowest available altitude. If there are no other planes holding on that side, they hold at 4000ft. If there are planes holding at that side, they must hold 1000ft above the lowest plane.

The planes holding above the SCA are granted entry when there are less than two planes at that IAF or assigned that MAHF. Once entry is granted, the AMM assigns the plane a MAHF opposite that of the entering plane's leader, the plane that entered the SCA just before. After the MAHF is assigned, the plane proceeds from the IAF, to the intermediate fix (IF), to the final approach fix (FAF), to the runway. See Figure 8.2. If the plane misses their runway approach, they proceed to their assigned MAHF (which corresponds to one of the IAFs) and attempt landing again.

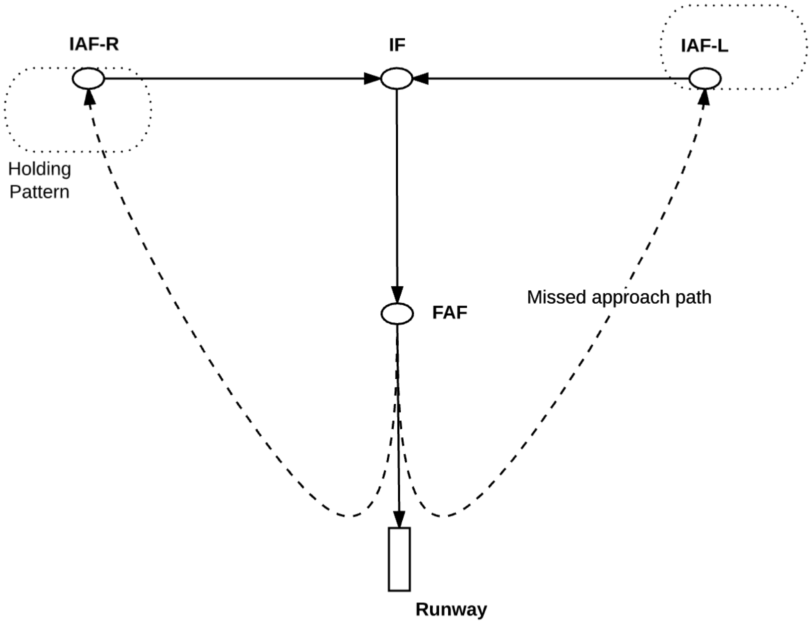


Figure 8.2: SCA from above

These rules when obeyed help the pilots keep safe separation in the SCA.

8.2 SATS model

In order to efficiently model SATS, we defined five different classes (AMM, flightPlan, flightSegment, waypoint, theClock), and one group (AFO - Abstract Flying Object). See Figure 8.3 for an abstract diagram illustrating how they interact.

8.2.1 Waypoints

Waypoints, or fixes, are 2D objects that characterize the SCA and define flight segments, which define flight plans. They are defined by their 3 attributes: latitude, longitude and name. Waypoints do not have any activities or workframes. There are five major waypoints in the SCA:

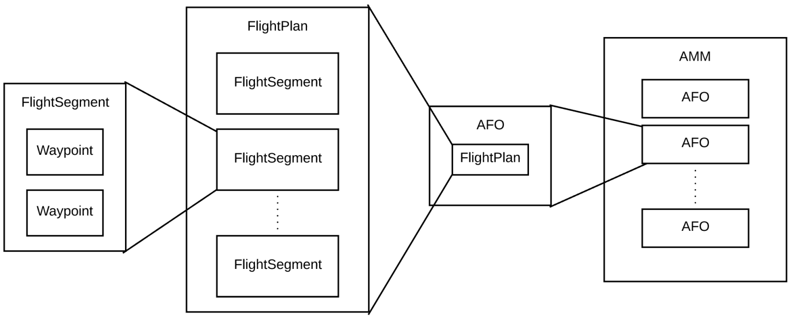


Figure 8.3: SATS components

- **IAF-R, IAF-L:** The Initial Approach Fix Right and Left. These are also known as the Missed Approach Holding Fixes, right and left (MAHF-R,L). Note in Figure 8.1 that the holding patterns at 2000ft, 3000ft and 4000ft on the same side rely on the same IAF, irrespective of their altitudes.
- **IF:** The Intermediate Fix
- **FAF:** The Final Approach Fix
- **RUN:** The Runway

For this particular model, nine additional fixes have been added to the SCA. See Figure 8.4. These additional fixes aid in defining the flight plans, especially the holding patterns, which were not defined in [5].

- **PIAF-R, HOLD1-R, HOLD2-R:** The fixes facilitating holding on the right side before the initial approach. The holding pattern goes from the Pre-IAF, to the IAF, to HOLD1 to HOLD2 and then back to the PIAF.
- **PIAF-L, HOLD1-L, HOLD2-L:** The fixes facilitating holding on the left side before the initial approach. The sequence is the same as on the right.
- **ENT-R, ENT-L:** Entry Right and Left facilitate entry into the SCA. When the to-waypoint of the plane is an entry fix, it requests entry to the AMM.

- **MAF:** Missed Approach Fix. This fix is used to facilitate a missed approach, when the plane cannot successfully initiate landing.

There are also four fixes modeled outside the SCA. All other fixes outside the SCA are not modeled for simplicity. We only concern ourselves with planes within a certain vicinity of the SCA and assume that their flight plans up to that point have been completed. Modeling all the waypoints outside the SCA is unnecessary.

- **DF-R, DF-L:** Departure Fix, right and left. These fixes are assigned to planes leaving the SCA.
- **OENT-R, OENT-L:** Outside Entry Right and Left. These fixes only serve as an origin point for planes created in the model.

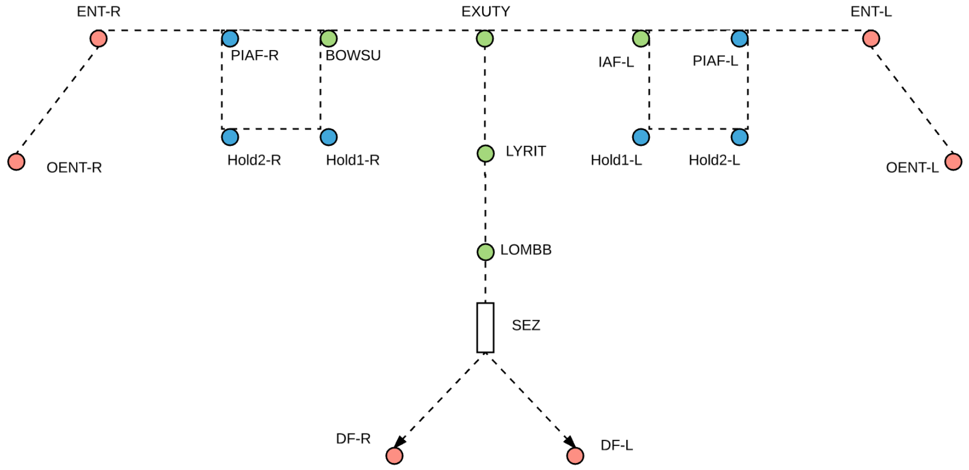


Figure 8.4: Fixes (waypoints) in and around the SCA in our model. These fixes are loosely based on the existing waypoints at and around the Sedona, Arizona airport (SEZ). BOWSU is the IAF-R, EXUTY is the IF, LYRIT is the FAF, and LOMBB is the MAF. Not to scale.

Our SATS model has 18 modeled waypoints.

An example plane entering the SCA (on the right or the left) will proceed from the outside entry fix (or waypoint) (OENT) to the entry point (ENT). While on the approach to the entry point, the plane requests entry from the AMM, and if granted, proceeds to the PIAF on its respective side. If holding is unnecessary, the plane proceeds to the IAF, IF, FAF, MAF and finally to the runway as long as the plane didn't miss the approach. If entry is rejected by the AMM, the plane proceeds from the entry point to hold at the altitude 1000ft above the highest plane on that side of the SCA and attempts entry again when approaching the IAF at 4000ft. When a plane approaches the IAF from the PIAF and holding is necessary, it proceeds from the IAF to HOLD1 to HOLD2 and then back to the PIAF, to perhaps repeat the cycle again. Approaching the MAF from the FAF, if it is determined that the plane cannot land, the plane flies from the MAF to its assigned MAHF, which corresponds to one of the IAFs.

8.2.2 Flight segments and flight plans

The paths between waypoints are flight segments. A flight segment is defined by two waypoints. SATS procedures as illustrated in [5] ensures that a safe distance is maintained between pilots by using specific "zones". In our model, flight segments have the same purpose. For instance, rather than checking if an aircraft was in the base-right zone, planes are checked to see if there are any whose current flight segment was from IAF-R to IF.

Flight segments also have an associated altitude. Because of this, there are many more flight segments than there are waypoint pairs. In our model, there are 110 flight segments. This number is due primarily to modeling the holding flight segments above the SCA. For instance, just between the waypoints HOLD1 and HOLD2, there is a flight segment at 2000ft, 3000ft, descending from 3000ft to 2000ft, 4000ft, descending from 4000ft to 3000ft etc. to 7000ft. We did not model any higher altitudes than 7000ft, so our model can only support 4 planes holding over the SCA per side. Flight segments do not have any workframes or activities; their attributes are *fromWaypoint, toWaypoint, end_altitude*, and *end_speed.*

Sequences of flight segments make up a flight plan for an aircraft. Flight plans are made up only of continuous sequences of flight segments. In our model there exists a flight plan from any modeled flight segment preceding the IAF to the runway. Our model has 34 flight plans. As planes attempt to enter the SCA and be cleared for the runway are rerouted, their flight plan changes from a direct path to the runway to hold either inside or outside the SCA. Planes can enter either laterally or vertically into the SCA. Vertical entry only occurs for a plane if lateral entry was initially denied to the plane and it moves to hold above the SCA. Flight plans do not have any workframes or activities, and their only attribute is a map of flight segments.

8.2.3 AFO

The AFO group has 6 workframes, 12 activities (4 of which serve for debugging and logging), and 25 attributes. These attributes describing the state of the plane include things like airspeed, latitude, longitude, bearing, altitude, flight plan, and toWaypoint, among other things. Our AFO group is based on NASA's AFO group used in modeling DSAS [11]. In order to make the AFO group work for SATS, we added the attributes *MAHF, entrySide, isHolding, holdingPt*, and *amm*. The rest of the attributes are used primarily to aid in flight path calculations. These calculations are executed in Java, where the AFO's state is updated and sent back to Brahms. There are two workframes that calculate the flight for the AFO, one that initializes the flight, one that deals with the state of the AFO's flight every time stamp. These workframes are based on the DSAS model. The other four workframes are for changing the flight plan when entry is refused, and when holding is necessary.

See Listing 8.1 for an example workframe. In order for this workframe to activate, the plane must be flying (*current.status = 1*), the AFO must be bound to the AMM that has been managing the AFO (*amm = current.amm*), the AFO has not yet been assigned a MAHF (*current.mahf = -1*), the AMM is not allowing entry on that side (*not(amm.allowLatEntry_ L = true)*), the AFO must be on its way to the entry fix on the left side (*current.toWaypoint = amm.entryPoints(3)*), and this must all occur when the global clock is ahead of the AFO's time stamp (*globalClock.time - current.timeStamp* $\geq$ *1*).

Listing 8.1: AFO: entryRefused_L

```
workframe entryRefused_L {
  repeat: false;
  priority: 2;
  variables:
    forone(AMM) amm;
  when (
    knownval(current.status = 1) and
    knownval(amm = current.amm) and
    knownval(current.mahf = −1) and
    not(amm.allowLatEntry_L = true) and
    knownval(current.toWaypoint = amm.entryPoints(3)) and
    knownval(globalClock.time − current.timeStamp >= 1))
  do {
    printToLog("entryRefused_L", current, "AFO", "workframeLogger");
    printToLog("entryRefused_L", current, amm, "scheduleLogger");
    //change flightplan to lowest available altitude outside the SCA
    denyEntryToSCA(current);
  }
}
```

It is only as planes approach the entry points that they check to see if entry is allowed. If this workframe activates, then the flight plan is updated so rather than proceeding to the runway as usual, the plane is rerouted to hold above the SCA at the lowest available altitude. The flight plan updating is taken care of in a Java activity. Most of the meaningful comparisons and calculations are done in Java because doing the same comparisons and calculations in Brahms can be much more inefficient. What Java can do in one simple function, Brahms might need several workframes to accomplish.

8.2.4 AMM

The AMM has 14 attributes, 9 activities, and 11 workframes to deal with its three responsibilities: granting/refusing entry, assigning MAHFs, and managing planes in the SCA.

- Granting and Refusing Entry to the SCA: These workframes check the entry booleans (attributes of the AMM) and communicate to the planes whether or not they are allowed entry at the particular entry point. See Listing 8.2 as an example. Note that before entry is allowed, the AMM must update its entry booleans (*current.entryBoolsUpdated = true*). Also note that because Brahms does not have modular arithmetic, we had to write a Java activity (*incrementIndexMod4*) to handle that simple calculation. The workframes that grant or refuse entry are as follows:

- **allowEntry_LatR**: Allows or denies entry for planes attempting to enter the SCA laterally from the right

Listing 8.2: AMM: allow lateral entry on the right

```
workframe allowEntry_LatR {
    repeat: true;
    priority: 2;
    variables:
        forone(AFO) afo;
        forone(int) index;
        forone(waypoint) toWaypoint;
        forone(int) newIndex;
    when(
        knownval(current.entryBoolsUpdated = true) and
        knownval(afo.status = 1) and
        not(current hasAFO afo) and
        knownval(index = current.latestPlane) and
        knownval(afo.altitude = 2000) and
        knownval(toWaypoint = current.entryPoints(2)) and
        knownval(afo.toWaypoint = toWaypoint) and
        knownval(current.allowLatEntry_R = true) and
        knownval(globalClock.time − current.timeStamp >=1))
    do {
        printToLog("allowEntry_LatR", current, "AMM", "workframeLogger");
        printToLog("allowEntry_LatR", current, afo, "scheduleLogger");
        conclude((current hasAFO afo));
        conclude((afo.amm = current));
        updateAFO(afo);
        printToLog("updateAFO", current, afo, "communicationLogger");
        incrementIndexMod4(index, newIndex);
        conclude((current.latestPlane = newIndex));
        conclude((current.scaQueue(newIndex) = afo));
        conclude((current.entryBoolsUpdated = false));
    }
}
```

- **allowEntry_LatL**: Allows or denies entry for planes attempting to enter the SCA laterally from the left
- **allowEntry_VertR**: Allows or denies entry for planes attempting to enter the SCA vertically from the right

  - **allowEntry_VertL**: Allows or denies entry for planes attempting to enter the SCA vertically from the left
  - **updateEntryBools**: This workframe looks at the position of the planes in the SCA to determine whether or not to allow entry at the four different entry points.

- Assigning MAHFs: Upon entry, the AMM assigns the aircraft a missed approach holding fix (MAHF-R, MAHF-L) which corresponds to the fixes IAF-R, IAF-L, respectively. This assignment is opposite the assignment of the leading plane, the plane just ahead of the entering plane in the sequence of aircraft. An entering plane that has no leading plane is assigned the MAHF that corresponds to the side they are entering on. See Listing 8.3 for an example. These workframes activate when the AMM "has" the AFO (a relation that exists only after entry is allowed), when that plane is flying, and it hasn't been assigned a MAHF. It also depends on the AMM's attribute *mahfOfLatestPlane*, which is just an integer 0 or 1 depending on the entry side; 0 corresponds to the right and 1 corresponds to the left. In the case of an initial plane entering the SCA, *mahfOfLatestPlane* will be -1, because there is no leading plane to follow.

  - **assignMAHF_R**: assigns a MAHF to a plane entering the SCA on the right when there is a leading plane to follow. See Listing 8.3
  - **assignMAHF_L**: assigns a MAHF to a plane entering the SCA on the left when there is a leading plane to follow
  - **assignMAHF_iR**: assigns a MAHF to the *initial* plane entering the SCA on the right
  - **assignMAHF_iL**: assigns a MAHF to the *initial* plane entering the SCA on the left

Listing 8.3: AMM: assigning the MAHF to a plane entering on the right

```
workframe assignMAHF_R {
    repeat: false;
    priority: 2;
    variables:
        forone(AFO) afo;
    when(
        knownval(current hasAFO afo) and
        knownval(afo.mahf = −1) and
        knownval(afo.status = 1) and
        knownval(current.mahfOfLatestPlane = 1))
    do {
        printToLog("assignMAHF_R", current, "AMM", "workframeLogger");
        printToLog("assignMAHF_R", current, afo, "scheduleLogger");
        conclude((current.mahfOfLatestPlane = 0));
        conclude((afo.mahf = current.mahfOfLatestPlane));
        assignMAHF(0, afo);
        printToLog("assignMAHF", current, afo, "communicationLogger");
    }
}
```

- Managing planes in the SCA
    - **updateMAHFWhenSCAEmpty**: when there are no more planes in the SCA, the MAHF attribute for the AMM must update so that the next plane coming in can be assigned the MAHF on its own entry side, rather than opposite the latest plane to enter the SCA.
    - **updateSCAQueue**: when a plane lands, the queue of planes the AMM manages is updated so that the index of the first plane points to the plane that no longer has a leading plane, and is next to be cleared for landing. This allows for continued entry if the SCA was at maximum capacity.

One of the AMM's crucial attributes is the *SCAqueue*, a data structure that keeps track of the planes that have been granted entry into the SCA. Because the only collection type in Brahms is a map, the queue is a map that maps an index to a plane. These indices always stay between 0 and 3, since only 4 planes are allowed in the SCA to preserve separation. Rather than shifting the planes in the map every time a plane lands, the AMM has another attribute *firstPlane* that is incremented (mod 4) so it can reference the plane closest to the runway. There is also an attribute *latestPlane* that is incremented (mod 4) when planes enter the SCA. These indices let the AMM know how many planes are in the SCA.

8.2.5 theClock

This object has a broadcast activity that announces the time to all agents and objects in the model. All the acting agents and objects in our model have a *timeStamp* attribute. This helps to control repeating workframes so that they cannot repeat an infinite number of times and block any lower priority workframes from activating. All workframes are set to activate only once in a time stamp. This is accomplished using handoff variables, a variable that must be a certain value in the preconditions to activate the workframe that is changed within that workframe, invalidating the preconditions. For example, Listing 8.2 relies on and updates the "entryBoolsUpdated" variable.

Within a time stamp, the ordering of workframes is controlled by the priority of the workframes. The clock's "announceTime" activity is in a workframe that has priority 0, so it will only activate after the higher priority workframes in the model have activated. In our model, all the workframes have a higher priority than 0 so the clock is the last thing to fire in every time stamp.

8.2.6 Java code

When planes are initialized, an equivalent plane object is created in Java. This plane object is cached within Java, so when planes need to be manipulated (their position is changed in flight, etc) rather than creating a new plane object for every calculation, the plane is pulled out of the cache by its unique identifier and updated accordingly. See Figure 8.5. This speeds up the simulation runtime considerably.

Because the Java activities are single threaded (see chapter 3), Brahms is essentially a scheduler, determining when these Java activities are called. The Java activities are called in a certain order; the corresponding threads in Java execute each activity one at a time.

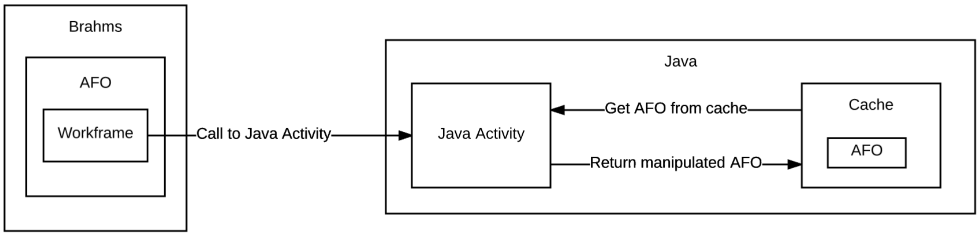


Figure 8.5: Brahms interactions with Java

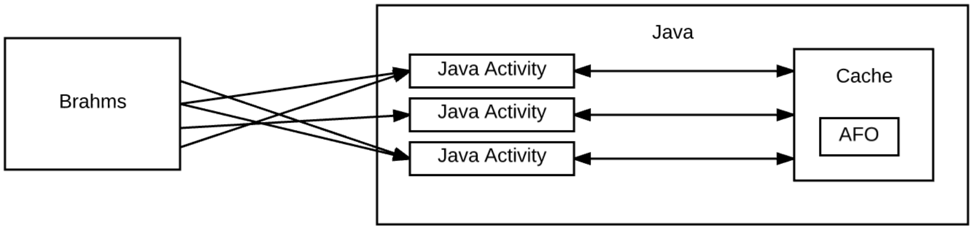


Figure 8.6: Brahms as a scheduler

Brahms is a convenient way to have concurrency. Note that even if two workframes are concurrent and both call Java, whichever calls into Java first happens first. In the case that both workframes are blocked, Java determines how to schedule the two workframes.

8.3 Existing Verification of SATS

Since SATS is an existing concept, it has been modeled and verified before. The initial model for SATS and basic verification of the model was proposed in [5] which describes an abstract model of the system. Since then, there have been several papers on the verification of SATS, many of them relying on simulation and human-in-the-loop (HITL) scenarios. HITL involves humans manually operating modules that behave according to simulations of models. Williams et al present their simulation in [21]. This is an incredibly effective way to verify the model, but also incredibly expensive. Once the model is in a state where an HITL simulation can be meaningful, adjusting the model requires much more time and money than earlier in development. These simulation methods also rely on choosing good test vectors, which likely may not be representative of the design space.

A more recent verification of SATS done by Sardar et al. appears in [18], in which the authors use a Discrete Time Markov Chain to create a probabilistic model that they check in PRISM. This improved upon the previous algorithm because it allows two planes in the self controlled area (SCA) simultaneously. Their model allows them to analyze collision risk, and it increases throughput, since more than one plane is allowed in the SCA. It also attempts to account for human error and off-nominal conditions. Rather than modeling plane dynamics as we do in our model, they break the SCA into "zones".

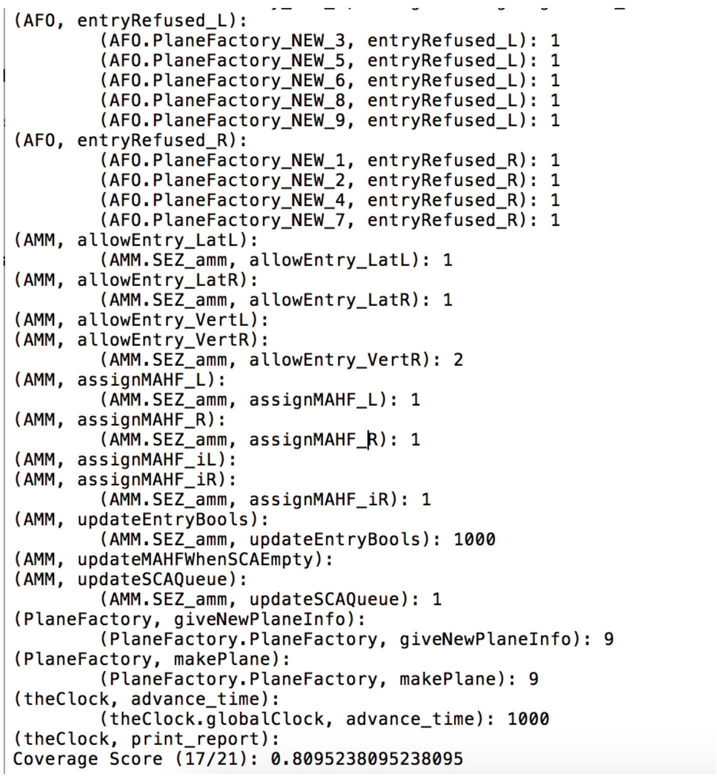
8.4 Case Study

The SATS concept was difficult to model in Brahms, and even more difficult to model correctly. The coverage metric defined in this thesis helped us to find and fix bugs expressed as strange behavior in the model. We explore examples of the coverage metric aiding in the debugging of SATS in this section.

8.4.1 Workframe coverage

The workframe coverage metric was helpful to reveal what scenarios needed to be written in order to more effectively explore the model space. We started testing our model with a scenario that had single plane, which resulted in only 42% workframe coverage. This is because our single plane in the scenario was entering laterally from the right, and no holding was ever necessary, so many workframes did not have the occasion to ever activate.

We added a second plane to the scenario to activate more workframes. While running scenarios with two planes, we realized that the second plane was not holding to allow the first plane to approach the runway with sufficient separation. Because of our attempt to reach 100% workframe coverage, bugs in our model were revealed. At this point, we still only had 58% workframe coverage. As we created more scenarios to achieve this total coverage (planes entering from both sides, too many planes for the SCA to accept planes, etc), more of our



```
(AFO, entryRefused_L):
        (AFO.PlaneFactory_NEW_3, entryRefused_L): 1
        (AFO.PlaneFactory_NEW_5, entryRefused_L): 1
        (AFO.PlaneFactory_NEW_6, entryRefused_L): 1
        (AFO.PlaneFactory_NEW_8, entryRefused_L): 1
        (AFO.PlaneFactory_NEW_9, entryRefused_L): 1
(AFO, entryRefused_R):
        (AFO.PlaneFactory_NEW_1, entryRefused_R): 1
        (AFO.PlaneFactory_NEW_2, entryRefused_R): 1
        (AFO.PlaneFactory_NEW_4, entryRefused_R): 1
        (AFO.PlaneFactory_NEW_7, entryRefused_R): 1
(AMM, allowEntry_LatL):
        (AMM.SEZ_amm, allowEntry_LatL): 1
(AMM, allowEntry_LatR):
        (AMM.SEZ_amm, allowEntry_LatR): 1
(AMM, allowEntry_VertL):
(AMM, allowEntry_VertR):
        (AMM.SEZ_amm, allowEntry_VertR): 2
(AMM, assignMAHF_L):
        (AMM.SEZ_amm, assignMAHF_L): 1
(AMM, assignMAHF_R):
        (AMM.SEZ_amm, assignMAHF_R): 1
(AMM, assignMAHF_iL):
(AMM, assignMAHF_iR):
        (AMM.SEZ_amm, assignMAHF_iR): 1
(AMM, updateEntryBools):
        (AMM.SEZ_amm, updateEntryBools): 1000
(AMM, updateMAHFWhenSCAEmpty):
(AMM, updateSCAQueue):
        (AMM.SEZ_amm, updateSCAQueue): 1
(PlaneFactory, giveNewPlaneInfo):
        (PlaneFactory.PlaneFactory, giveNewPlaneInfo): 9
(PlaneFactory, makePlane):
        (PlaneFactory.PlaneFactory, makePlane): 9
(theClock, advance_time):
        (theClock.globalClock, advance_time): 1000
(theClock, print_report):
Coverage Score (17/21): 0.8095238095238095
```

Figure 8.7: Workframe coverage details for SATS model

model behavior space was revealed and so were the bugs. Figure 8.7 shows a portion of a workframe coverage details file for the SATS model in a scenario with many planes.

The workframe coverage details file helps to answer the questions: which workframes are being activated? are there any workframes being activated too much? which agents or objects are activating which workframes? The simulation for this particular scenario activated 17/21 workframes. It is possible to achieve 100% workframe coverage, so we can see how to augment our scenario (or write a new scenario) in order to achieve this 100% coverage. The coverage details file in Figure 8.7 tells us that the AMM never experienced an empty SCA, and therefore never activated the *updateMAHFWhenSCAEmpty* workframe. Also, the *assignMAHF_ iL* workframe was never activated by the AMM because the initial plane entering the SCA came in on the right-hand side. These two workframes can be activated if the scenario was written in such a way that the simulation had time to let all the planes in the SCA land, thus activating the *updateMAHFWhenSCAEmpty* workframe, and then having a plane enter the SCA from the left side, this activating the *assignMAHF_ iL* workframe. The other two unactivated workframes for that particular scenario were *allowEntry_ VertL* for the AMM and *print_ report* for theClock. It is possible that *allowEntry_ VertL* would activate as a natural consequence of extending the simulation time, or it could be that there is a bug in the model, preventing that workframe to activate. The last workframe, *print_ report*, is a logging tool, and was not set to activate this particular run. We were able to write a scenario that did achieve 100% workframe coverage and fix several major bugs along the way.

Another useful feature of the workframe coverage details file, is the number reported next to each workframe. In Figure 8.7, you can see that the AMM activates the *updateEntryBools* workframe 1000 times. It may be the case that the *updateEntryBools* workframe needs its predicate strengthened so that it is not activating so often. In the case of our SATS model, this many calls to the *updateEntryBools* was necessary, but the reported numbers in the details file can help the model builder understand how the model is running.

An intimate knowledge of the workings of the model helps in understanding which workframes did not activate and why. It is possible for an outsider to look at the model and hypothesize how to change the scenarios in order to achieve 100% workframe coverage if the names of the workframes are expressive, but the coverage details files are more easily understood by those familiar with the model. That is why further research into automatically generating scenarios based on the workframe coverage metric would be valuable.

8.4.2 Communication coverage

It is easy to achieve 100% communication coverage when you have only one instance of every concept. Our first scenarios with a single plane always achieved total communication coverage. See Listing 8.4.

When we added more planes to the scenario to achieve 100% workframe coverage, we achieved 93% communication coverage. One of the communication activities was not happening as we expected. Particularly, even though the workframe allowing vertical entry on the right was allowed, the plane executing that entry was never assigned a MAHF. The communication coverage metric alerted us to this issue and we were able to fix it and achieve 100% communication coverage.

Listing 8.4: Communication coverage for one plane scenario

```
(LR_ENTRYPLANE, receiveAMMMap, SEZ_amm): 1
(LR_ENTRYPLANE, sendAFOData, SEZ_amm): 882
(SEZ_amm, assignMAHF, LR_ENTRYPLANE): 1
(SEZ_amm, updateAFO, LR_ENTRYPLANE): 1
Coverage Score (4/4): 1.0
```

8.4.3 Schedule coverage

We were never able to achieve 100% schedule coverage. This was expected after our empirical study, since there are so many intervening writes between two workframe with data-race. Every time stamp in our model hosts a communication activity that writes several variables (the altitude, toWaypoint, and holding boolean included) that are read as part of the preconditions of several different workframes. Because this communication takes place every time stamp, there is an intervening write for all workframes depending on those variables, and the schedule coverage requirement cannot be satisfied. Listing 8.5 is an example of a schedule coverage details file for one of our scenarios.

It is important to note that the success of schedule coverage depends on the success of workframe coverage. If not all the workframes are being activated in a scenario, then there is no way that the schedule coverage requirements involving the unactivated workframes will be met.

As mentioned in chapter 7, the best use for the schedule coverage requirement would be as part of an automated tool that has control of the Brahms schedule and can drive the model to explore the behavior space and ultimately achieve 100% schedule coverage if it is possible.

Listing 8.5: Schedule coverage for four plane scenario

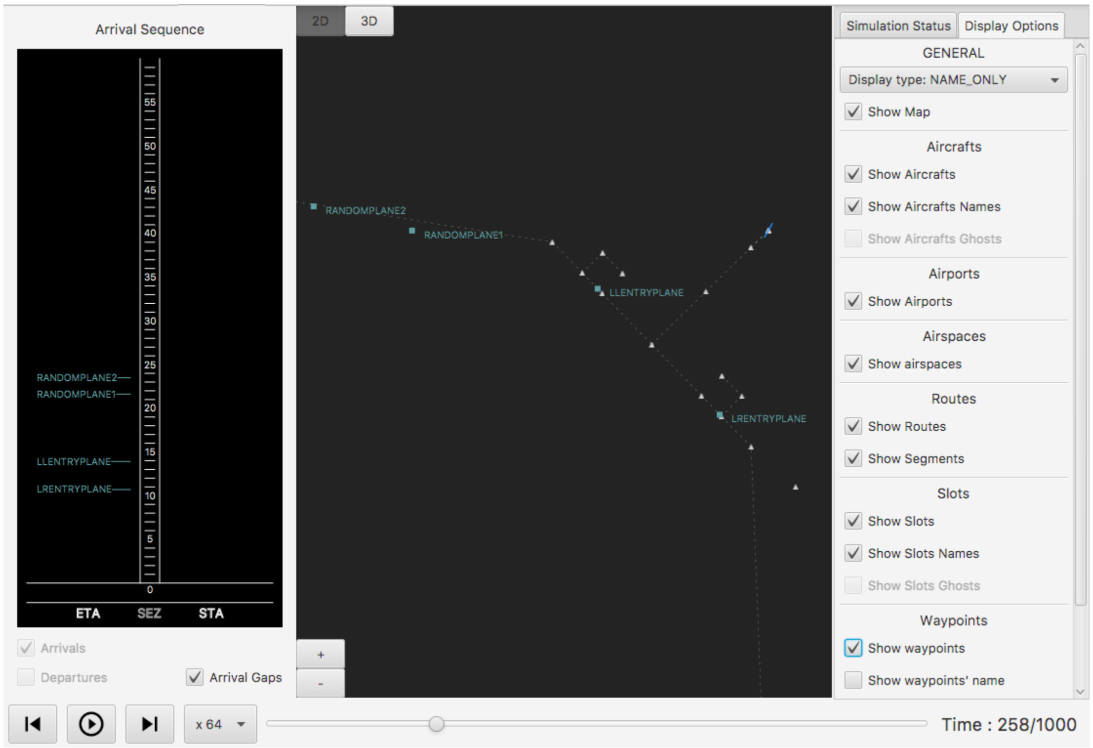
```
(AFO.Fly_To_Waypoint, AMM.allowEntry_VertL):
   (LR_ENTRYPLANE.Fly_To_Waypoint, SEZ_amm.allowEntry_VertL)
(AFO.Fly_To_Waypoint, AMM.allowEntry_VertR):
   (LR_ENTRYPLANE.Fly_To_Waypoint, SEZ_amm.allowEntry_VertR)
(AFO.beginFlying, AFO.Fly_To_Waypoint):
   (LR_ENTRYPLANE.beginFlying, LR_ENTRYPLANE.Fly_To_Waypoint)
(AFO.changeHoldingFlightPlan_L, AMM.allowEntry_LatL):
   (LL_ENTRYPLANE.changeHoldingFlightPlan_L, SEZ_amm.allowEntry_LatL)
(AFO.changeHoldingFlightPlan_L, AMM.allowEntry_VertL):
   (LL_ENTRYPLANE.changeHoldingFlightPlan_L, SEZ_amm.allowEntry_VertL)
(AFO.changeHoldingFlightPlan_R, AMM.allowEntry_LatL):
   (LR_ENTRYPLANE.changeHoldingFlightPlan_R, SEZ_amm.allowEntry_LatL)
(AFO.changeHoldingFlightPlan_R, AMM.allowEntry_VertR):
   (LR_ENTRYPLANE.changeHoldingFlightPlan_R, SEZ_amm.allowEntry_VertR)
(AFO.entryRefused_L, AMM.allowEntry_VertR):
   (LL_ENTRYPLANE2.entryRefused_L, SEZ_amm.allowEntry_VertR)
(AFO.entryRefused_R, AMM.allowEntry_VertR):
   (LR_ENTRYPLANE2.entryRefused_R, SEZ_amm.allowEntry_VertR)
(AMM.updateSCAQueue, AMM.updateMAHFWhenSCAEmpty):
   (SEZ_amm.updateSCAQueue, SEZ_amm.updateMAHFWhenSCAEmpty)
Coverage Score (10/126): 0.07936507936507936
```

8.5 SATS Visualization

An invaluable tool in this research was the visualizer we used. The visualizer was taken from [17]. We instrumented the SATS model to output the necessary information to the logs for the visualizer to parse. Complicated models like SATS are difficult to verify with logs alone, which is why using the visualizer to review the simulations was so effective. The visualizer shows the waypoints in the model, as well as the planes traveling between the waypoints. See Figure 8.8.

Another useful feature of the visualizer was the ability to select planes and see some of their more important attributes, like altitude and ETA, see Figure 8.9.

The visualizer serves as an oracle; it lets us know how our model is behaving. Without the oracle our coverage metric is of no use, because even if we know that all the workframes have been activated, say, that doesn't tell us if the model is behaving correctly. With small models, logged output can be oracle enough, but most meaningful models will be complicated



GENERAL

Display type: NAME_ONLY

☑ Show Map

Aircrafts

☑ Show Aircrafts

☑ Show Aircrafts Names

□ Show Aircrafts Ghosts

Airports

☑ Show Airports

Airspaces

☑ Show airspaces

Routes

☑ Show Routes

☑ Show Segments

Slots

☑ Show Slots

☑ Show Slots Names

□ Show Slots Ghosts

Waypoints

☑ Show waypoints

□ Show waypoints' name

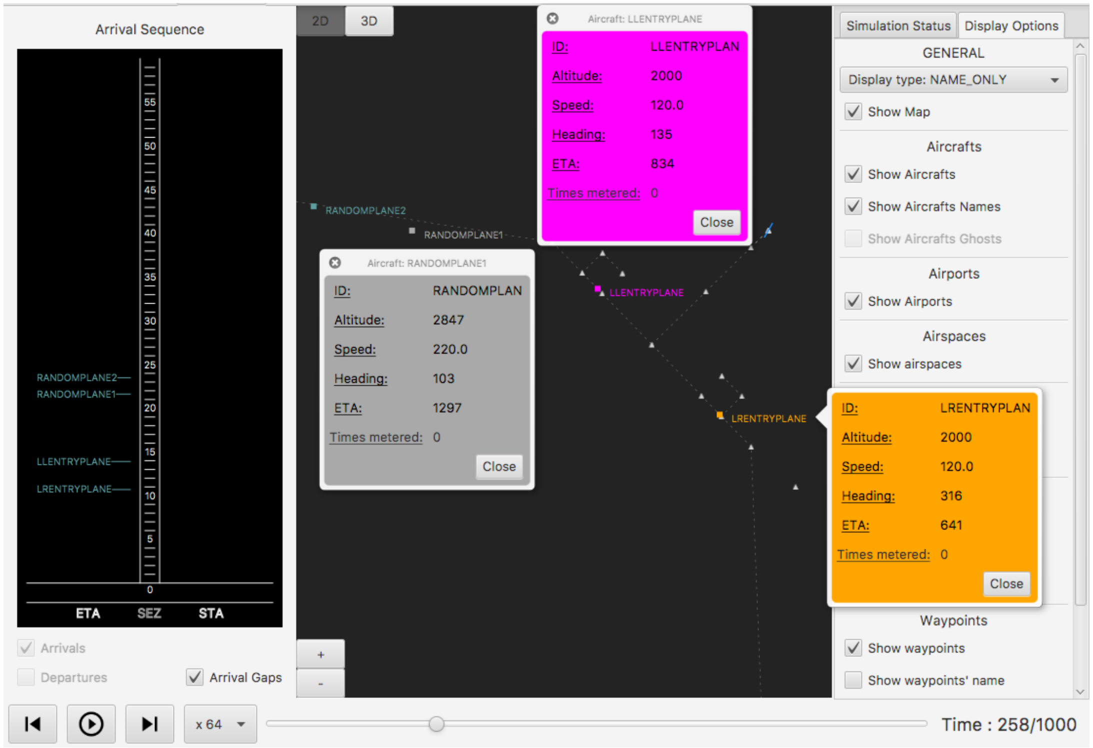
☑ Arrivals

□ Departures

☑ Arrival Gaps

Time : 258/1000

Figure 8.8: Visualizer for SATS



Aircraft: LLENTRYPLANE

ID:	LLENTRYPLAN
Altitude:	2000
Speed:	120.0
Heading:	135
ETA:	834
Times metered:	0

Aircraft: RANDOMPLANE1

ID:	RANDOMPLAN
Altitude:	2847
Speed:	220.0
Heading:	103
ETA:	1297
Times metered:	0

ID:	LRENTRYPLAN
Altitude:	2000
Speed:	120.0
Heading:	316
ETA:	641
Times metered:	0

GENERAL

Display type: NAME_ONLY

☑ Show Map

Aircrafts

☑ Show Aircrafts

☑ Show Aircrafts Names

□ Show Aircrafts Ghosts

Airports

☑ Show Airports

Airspaces

☑ Show airspaces

Waypoints

☑ Show waypoints

□ Show waypoints' name

☑ Arrivals

□ Departures

☑ Arrival Gaps

Time : 258/1000

Figure 8.9: Visualizer with additional insight

enough that a log file on its own will be too verbose to be easily parsed without an automated tool. Not all models will have a visualization tool, or the resources necessary to construct a visualizer. This is one reason why we chose SATS as our model for testing the coverage metric. We already had a visualizer that could easily model aircraft interactions [17].

The visualizer still isn't the best solution, since it still requires a human to watch for odd behavior. The best solution would be automated, but since Brahms doesn't have any assertions, that is difficult to orchestrate. It is clear if there are exceptions that the model isn't behaving correctly, but after the exceptions are gone we only have the oracle, in our case the visualizer, to show us model behavior.

Chapter 9

Conclusion

Testing multi-agent systems is hard, because the model attempts to capture the behavior of multiple decision makers working independently to reach their own goals. Though these agents are independent, they are affected by each other because they use information from the same environment, and their actions can change the environment. The only scalable solution to testing multi-agent systems is simulation, which is an inefficient way to test the model because many multi-agent systems are non-deterministic and that non-determinism is hard to witness without thousands, even millions of simulations if it is even witnessed at all. Simulation on its own offers no control of the testing, and may not cover the test space well.

This thesis defines coverage over three metrics: workframe coverage, communication coverage, and schedule coverage. Workframe coverage reveals what actions agents and objects in the model have taken. Communication coverage looks for every possible variable binding workframes can achieve. Schedule coverage requires every ordering of every pair of workframes that may experience data-race. We explored all three coverage metrics in an empirical study and a case study. Both these studies revealed the usefulness of each metric. Workframe coverage is especially useful for manual test creation since it is the only coverage metric consistently able to achieve 100% coverage. Communication and schedule coverage requirements often over-estimate the actual possible communications and schedules that can take place in the model, resulting in poor coverage scores. These metrics are still useful for guiding manual test creation, but, especially in the case of schedule coverage, would be most beneficial as part of an automated test effort with more control of the simulation.

Even though 100% coverage is not likely achievable across all three metrics, this is not an issue. We care more about how the coverage metric helps us than we care about achieving perfect coverage. The coverage details files offer information that can aid in test creation. Currently, the tests are created by the human tester, but in the future an automated system can be constructed to take the coverage details from a given simulation and generate tests for subsequent runs. Also possible is using the coverage metric in active test, so the coverage is being achieved as the simulation runs in one test.

9.1 Future Work

Because relatively little has been done in formally testing multi-agent systems, there are many directions that this work could go. Before taking this research and moving on, there are a few things that could be done to improve the tool. Firstly, Java activities can be dealt with rather than ignored. Because of limited time and the limited Java API, we decided to forgo taking into account the variables affected by Java activities. In an earlier attempt to handle these variables, it was discovered that no matter what the workframe binds to, the Java activity can access the variables of any other agent or object in the system. The fact that we ignore variable manipulation in Java activities is the biggest weakness of our coverage metric. Another way to improve the coverage tool is to allow for the input of a suite of tests, rather than one test at a time.

Once the coverage metric is perfected, it can be used as a tool for further improving the testing of multi-agent systems. The metric can be used to generate tests automatically based on the coverage calculation, and can even be used to guide active test.

9.1.1 Active Test

Active Test was the goal we had in mind when defining this coverage metric. Active test is used to guide the simulation to meet the coverage requirements on the list. Java Pathfinder (JPF) is a good tool for active testing, since it is a fully customizable Java virtual machine that is capable of backtracking execution to explore different thread schedules. Given Java bytecode, JPF executes the program, stopping at execution choices to allow the user to decide which path to take. Theoretically, JPF can execute all reachable paths through a program assuming the needed input is provided. It uses state matching to combat the state explosion problem: when it reaches a state similar to one it has already visited, it backtracks to the most recent decision point that still has unexplored choices. A JPF friendly Brahms runtime exists for JPF that is able to backtrack to any point of non-determinism in the model and explore all possible resolutions. These points of non-determinism are variable bindings for workframes and schedule orderings. The active test would make decisions at these points based on the coverage list.

Ultimately, the behavior space of the model is explored to ensure all possible behaviors are revealed. This strategy—driving tests into unexplored spaces—is known as active test in the literature. By activating workframes and binding variables to workframes according to the coverage requirements, many behaviors of the model will be revealed. As the simulator witnesses these behaviors, the coverage requirement resulting in these behaviors will be removed from the list. The initial size of the coverage requirement list is recorded and compared to the size of the list after the simulation. Ideally the size of the list after simulation will be zero, but because many of the coverage requirements obtained during static analysis are impossible since the static analysis over-approximates the coverage, there will be cases where the size of the list after simulation will be greater than zero.

9.1.2 Refinement

One advantage of active test is that it is dynamic, so as the simulation runs, it is possible to eliminate coverage requirements that are impossible. For instance, if two agents are never active at the same time (i.e. two planes that never fly at the same time), then any coverage entry in the list relating the two agents are impossible by definition. Another example of impossible coverage requirements is when workframes have a forced ordering. It is possible that after completion, one workframe will negate the preconditions of a workframe that was scheduled to follow it. These schedules are impossible to cover, since the subsequent workframe will never activate. This refinement of the coverage requirements can be incorporated into the active test to make the coverage results of the simulation more accurate by eliminating impossibilities on the list.

References

[1] Robert John Allan. Survey of agent based modelling and simulation tools. Technical report, Science & Technology Facilities Council, 2009.

[2] Christel Baier, Joost-Pieter Katoen, et al. Principles of model checking. *MIT Press Cambridge*, 26:58, 2008.

[3] Rafael H Bordini, Michael Fisher, Willem Visser, and Michael Wooldridge. Verifying multi-agent programs by model checking. *Autonomous agents and multi-agent systems*, 12(2):239–256, 2006.

[4] NASA Ames Research Center. Brahms, 1998. URL `http://ejenta.com/documentation`. [Online; accessed 9-November-2017].

[5] Gilles Dowek, Cesar Munoz, and Victor A Carreno. Abstract model of the SATS concept of operations: Initial results and recommendations. Technical report, NASA Langley Research Center; Hampton, VA, United States, 2004.

[6] Peter Gammie and Ron Van Der Meyden. Mck: Model checking the logic of knowledge. In *Proceedings of Computer Aided Verification*, volume 3114, pages 479–483. Springer, 2004.

[7] Shin Hong, Jaemin Ahn, Sangmin Park, Moonzoo Kim, and Mary Jean Harrold. Testing concurrent programs to achieve high synchronization coverage. In *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, pages 210–220. ACM, 2012.

[8] Shin Hong, Matt Staats, Jaemin Ahn, Moonzoo Kim, and Gregg Rothermel. Are concurrency coverage metrics effective for testing: a comprehensive empirical investigation. *Software Testing, Verification and Reliability*, 25(4):334–370, 2015.

[9] Josie Hunter, Franco Raimondi, Neha Rungta, and Richard Stocker. A synergistic and extensible framework for multi-agent system verification. In *Proceedings of the 2013 International Conference on Autonomous Agents and Multi-agent Systems*, pages

869–876. International Foundation for Autonomous Agents and Multiagent Systems, 2013.

[10] Daniel Jackson and Craig A Damon. Elements of style: Analyzing a software design feature with a counterexample detector. *Software Engineering, IEEE Transactions on*, 22(7):484–495, 1996.

[11] Paul U Lee, Nancy M Smith, Connie Brasil, Eric Chevalley, Jeffrey Homola, Bonny Parke, Hyo-Sang Yoo, Nancy Bienert, Abhay Borade, Nathan Buckley, et al. Reducing departure delays at LaGuardia airport with departure-sensitive arrival spacing (DSAS) operations. *Air Traffic Control Quarterly*, 23(4):245–273, 2015.

[12] Alessio Lomuscio, Wojciech Penczek, and Hongyang Qu. Partial order reductions for model checking temporal-epistemic logics over interleaved multi-agent systems. *Fundamenta Informaticae*, 101(1-2):71–90, 2010.

[13] Madanlal Musuvathi and Shaz Qadeer. Iterative context bounding for systematic testing of multithreaded programs. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 446–455, New York, NY, USA, 2007. ACM.

[14] Wojciech Penczek and Alessio Lomuscio. Verifying epistemic properties of multi-agent systems via bounded model checking. In *Proceedings of the Second International Joint Conference on Autonomous Agents and Multiagent Systems*, pages 209–216. ACM, 2003.

[15] M. Peters. Capacity analysis of the NASA Langley airport management module. In *Proceedings of the 24th Digital Avionics Systems Conference*, volume 1, pages 6–12, Oct 2005.

[16] Anand S Rao, Michael P Georgeff, et al. BDI agents: From theory to practice. In *Proceedings of the First International Conference on Multiagent Systems*, volume 95, pages 312–319, 1995.

[17] Neha Rungta, Eric G Mercer, Franco Raimondi, Bjorn C Krantz, Richard Stocker, and Andrew Wallace. Modeling complex air traffic management systems. In *Proceedings of 2016 IEEE/ACM 8th International Workshop on Modeling in Software Engineering (MiSE)*, pages 41–47. IEEE, 2016.

[18] Muhammad Usama Sardar, Nida Afaq, Khaza Anuarul Hoque, Taylor T Johnson, and Osman Hasan. Probabilistic formal verification of the SATS concept of operation. In *Proccedings of NASA Formal Methods Symposium*, pages 191–205. Springer, 2016.

[19] Valerio Terragni and Shing-Chi Cheung. Coverage-driven test code generation for concurrent classes. In *Proceedings of the 38th International Conference on Software Engineering*, pages 1121–1132. ACM, 2016.

[20] Wikipedia. Comparison of agent-based modeling software — wikipedia, the free encyclopedia, 2017. URL `https://en.wikipedia.org/w/index.php?title=Comparison_of_agent-based_modeling_software&oldid=803908747`. [Online; accessed 19-October-2017].

[21] Daniel M Williams. Point-to-point! validation of the small aircraft transportation system higher volume operations concept. In *Proceedings of the International Congress of the Aeronautical Sciences*, volume 25. International Council of the Aeronautical Sciences, 2006.

[22] Jie Yu, Satish Narayanasamy, Cristiano Pereira, and Gilles Pokam. Maple: A coverage-driven testing tool for multithreaded programs. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, volume 47, pages 485–502, New York, NY, USA, 2012. ACM.