

Modeling for the Design of Adaptive Systems

Abstract

A *monitoring and response system* (MRS) is a security mechanism intended to help protect an underlying system from attack, or to help identify and reverse compromises when they occur. While an MRS attempts to discover potential threats, which other security mechanisms may not detect, the MRS may have vulnerabilities of its own which go undiscovered. Alternatively, if an adversary can coax a component to provide misleading or distracting information, the MRS may lead security administrators to miss an attack facilitating an adversary's work.

The purpose of this report is to summarize a substantial line of work intended to develop principles for analyzing and appraising an MRS either after implementation or during the design stage. In particular, this line of work consists of three separate areas: a grammar-based model for MRS, a graph-based model for MRS, and building assurance cases for MRS. First, we explored how to specify an MRS based on an inventory of the components it comprises; these we specified as a datatype by a context-free grammar. We found that this type of a catalog, while informative, did not emphasize the structure that would allow us to analyze whether its components were performing in a way required by the security architecture of the MRS. This motivated us to reinterpret the MRS descriptions as directed graphs where arrows flow in the same direction as information through the MRS. The derived graph structure bounds the ways that events can have causal effects on the different components of the MRS. We then explored using assurance cases to understand the component-level structure of an MRS since they are a piece of careful informal reasoning that aims to identify the important causal processes that may cause an MRS to fail to meet its security goals. To argue that an MRS resists adverse consequences, it must propose specific responsibilities for components at successive levels of decomposition.

To summarize, we explored a few techniques that enable a designer to build more secure MRS and an analyst to assess the security posture of existing MRS. Some of our techniques worked well and are lightweight and straightforward to use. The main contribution of our work was developing an improved methodology that makes use of MRS models, provides an ability for a designer or analyst to navigate between many different views of those models, and a technique to keep track of both the views of and arguments made in support of the MRS meeting its intended security goals.

REPORT DOCUMENTATION PAGE					Form Approved OMB No. 0704-0188	
The public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Department of Defense, Washington Headquarters Services, Directorate for Information Operations and Reports (0704-0188), 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302. Respondents should be aware that notwithstanding any other provision of law, no person shall be subject to any penalty for failing to comply with a collection of information if it does not display a currently valid OMB control number. PLEASE DO NOT RETURN YOUR FORM TO THE ABOVE ADDRESS.						
1. REPORT DATE (DD-MM-YYYY) 07/01/2021		2. REPORT TYPE Technical Report			3. DATES COVERED (From - To)	
4. TITLE AND SUBTITLE Modeling for the Design of Adaptive Systems				5a. CONTRACT NUMBER W56KGU-20-F-0009		
				5b. GRANT NUMBER		
				5c. PROGRAM ELEMENT NUMBER		
6. AUTHOR(S) Banuelos, Cervando A., II; Burgin, Kelley W.; Cheuvront, Mike; Ellis, Devon P.; Guttman, Joshua D.; Loscocco, Peter A.; Rowe, Paul D.; Van Bruggen, Dirk C.; Vohaska, Brian M.;				5d. PROJECT NUMBER		
				5e. TASK NUMBER		
				5f. WORK UNIT NUMBER		
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) The MITRE Corporation 7515 Colshire Drive McLean, VA 22102				8. PERFORMING ORGANIZATION REPORT NUMBER PRS-21-0457		
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)				10. SPONSOR/MONITOR'S ACRONYM(S)		
				11. SPONSOR/MONITOR'S REPORT NUMBER(S)		
12. DISTRIBUTION/AVAILABILITY STATEMENT DISTRIBUTION STATEMENT A. Approved for public release: distribution unlimited.						
13. SUPPLEMENTARY NOTES						
14. ABSTRACT The purpose of this report is to summarize a substantial line of work intended to develop principles for analyzing and appraising a monitoring and response system (MRS) either after implementation or during the design stage. In particular, this line of work consists of three separate areas: a grammar-based model for MRS, a graph-based model for MRS, and building assurance cases for MRS.						
15. SUBJECT TERMS Information Security Architecture; Information Security Operations; Computer Security DAG; OSSEC; MRS; Wazuh; monitoring and response system; OPM; Object Process Methodology; directed acyclic graph						
16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES 80	19a. NAME OF RESPONSIBLE PERSON Susan Carpenito	
a. REPORT	b. ABSTRACT	c. THIS PAGE			19b. TELEPHONE NUMBER (Include area code) 781-271-7646	

Modeling for the Design of Adaptive Systems

Cervando A Banuelos II¹ Kelley W. Burgin¹ Mike Chevront¹
Devon P. Ellis¹ Joshua D. Guttman¹ Peter A. Loscocco²
Paul D. Rowe¹ Dirk C. Van Bruggen² Brian M. Vohaska¹

¹The MITRE Corporation

²Laboratory for Advanced Cybersecurity Research / NSA

July 1, 2021

Abstract

A *monitoring and response system* (MRS) is a security mechanism intended to help protect an underlying system from attack, or to help identify and reverse compromises when they occur. While an MRS attempts to discover potential threats, which other security mechanisms may not detect, the MRS may have vulnerabilities of its own which go undiscovered. Alternatively, if an adversary can coax a component to provide misleading or distracting information, the MRS may lead security administrators to miss an attack facilitating an adversary's work.

The purpose of this report is to summarize a substantial line of work intended to develop principles for analyzing and appraising an MRS either after implementation or during the design stage. In particular, this line of work consists of three separate areas: a grammar-based model for MRS, a graph-based model for MRS, and building assurance cases for MRS. First, we explored how to specify an MRS based on an inventory of the components it comprises; these we specified as a datatype by a context-free grammar. We found that this type of a catalog, while informative, did not emphasize the structure that would allow us to analyze whether its components were performing in a way required by the security architecture of the MRS. This motivated us to reinterpret the MRS descriptions as directed graphs where arrows flow in the same direction as information through the MRS. The derived graph structure bounds the ways that events can have causal effects on the different components of the MRS. We then explored using assurance cases to understand the component-level structure of an MRS since they are a piece of careful informal reasoning that aims to identify the important causal processes that may cause an MRS to fail to meet its security goals. To argue that an MRS resists adverse consequences, it must propose specific responsibilities for components at successive levels of decomposition.

To summarize, we explored a few techniques that enable a designer to build more secure MRS and an analyst to assess the security posture of existing MRS. Some of our techniques worked well and are lightweight and straightforward to use. The main contribution of our work was developing an improved methodology that makes use of MRS models, provides an ability for a designer or analyst to navigate between many different views of those models, and a technique to keep track of both the views of and arguments made in support of the MRS meeting its intended security goals.

Contents

1	Introduction	1
1.1	The Importance of an MRS Working Correctly	1
1.2	Differing Perspectives on MRS Analysis	2
1.3	Goals of MRS Analysis Tools	3
1.4	Summary of Our Work	4
2	Background / Motivation	5
2.1	Monitoring and Response Systems	6
2.2	Motivating Goals	9
2.3	Abstract MRS Component Model	10
2.4	MRS Case Studies	13
2.5	Summary	18
3	A Grammar-based Model	18
3.1	Syntax	18
3.2	Example 1: Decomposed Thermostat	19
3.3	Example 2: Single-Component Thermostat	20
3.4	Example 3: Temperature Aggregation Thermostat	21
3.5	MRS Execution	22
3.6	Summary	22
4	Graph-based Model Manipulation	23
4.1	Lexicon	23
4.2	Graph Transformations	32
4.3	A Tree of Views	33
4.4	Summary	34
5	Graph-based Example: Weather Service System	35
5.1	Correctness of the Weather Service	36
5.2	Correctness of the Weather Stations	42
5.3	A Global Assurance Argument	46
5.4	Summary	48
6	Building an Assurance Case	48
6.1	Background	48
6.2	Assurance Cases In Practice	50
6.3	Summary	54
7	Open Problems	54
7.1	Extending our Work	55
7.2	Arguments and Modeling	55
7.3	Tooling Support	55
8	Conclusion	56

Appendices	59
A Existing Tools / Techniques	60
A.1 OPCAT: Object Process Methodology	60
A.2 GROOVE - Graph Rewriting	62
A.3 OSATE - Architecture Analysis and Design Language (AADL)	69
A.4 Summary	76

1 Introduction

A *monitoring and response system* (MRS) is a security mechanism intended to help protect an underlying system from attack, or to help identify and reverse compromises when they occur. It accumulates information about security-relevant conditions and events; it makes that information available to a decision-maker who can determine appropriate responses; and it delivers those responses back to the affected parts of the system to parry the actions of an adversary. Very frequently, an MRS is used to protect a distributed system, in which case it generally involves *probes* or *agents* at the end components of the system, and a *management component* at which security administrators obtain an overall view and select the responses necessary to protect the underlying system.

1.1 The Importance of an MRS Working Correctly

Another way to understand the intended functionality of an MRS is to consider the effectiveness of security mechanisms utilized in a system. We know that there is no set of security mechanisms that will provide perfect security against any adversary. As a result, operational systems employ security mechanisms that will attempt to monitor a set of likely security imperfections, attempt to detect security events, and respond to these events given some behavior according to some security model. For example, an application may be monitored by an anti-virus program to mitigate known threat behavior. Other tools such as network traffic inspectors monitor network traffic and identify threats. These example security mechanisms may also have gaps. When existing security mechanisms contain some security gap, an MRS can be employed to help fill that gap by looking for behaviors that have some possibility of being malicious and responding to those potential threats. MRS are in a way a meta-security mechanism. From a design perspective, an MRS is intended to buttress confidence that existing security mechanisms such as anti-virus software are working as intended. If those systems are not working in a correct and sound manner then the MRS should detect this, alert in some way, and respond accordingly. With this additional security assurance, an MRS can build confidence that the system it monitors had not been compromised under the security model on which the MRS has been configured. As such, it is important to ensure that an MRS is designed appropriately and that design can be validated against a security model.

While an MRS attempts to discover potential threats, which other security mechanisms may not detect, the MRS can have vulnerabilities of its own which may go undiscovered. MRS are usually made of three parts: a client, an orchestrator or manager, and a response system; each of these components must necessarily interact with each other. Vulnerabilities in one of these components can also be disastrous. Indeed, the situation may be much worse than if an MRS was not added to the system.

While application security vulnerabilities such as buffer overflows can be identified using tools such as symbolic analysis or signature-based code analysis, these tools may not be well suited to detect vulnerabilities that are caused by the MRS design itself. For instance, insecure communication from agent to manager may allow an adversary to replace alerts with messages saying that the status is harmless. Insecure communication from manager to agent may allow an adversary to insert commands that undermine rather than restore the security of the endpoint. Alternatively, if an adversary can coax a component to provide misleading or distracting information, the MRS may lead security administrators to miss an attack and facilitate the adversary's work. As such, operators of MRS may wish to treat

MRS as critical infrastructure.

An MRS typically resides in a trusted environment with administrator like privileges. For example, the probes or agents associated with an MRS might retain root or admin previewers as they run on a client machine; the manager might run on a machine with network administrator privileges which could open up potential adversary vectors in the case of compromise.

If the MRS were to be compromised and the system were to heavily rely on MRS detection and response, an adversary might be able to achieve a deep level of stealth as the system that should be detecting adversarial behavior is unable to do so. This scenario might be worse than the case when an MRS is not employed as other security mechanisms would likely be employed to compensate.

Furthermore, if an adversary were able to compromise the MRS manager, that adversary could instruct the MRS agents to perform some malicious action; this action would be determined to be valid as it came from the manager which is authorized to authoritatively send instructions.

When there are operational or design flaws in an MRS, these flaws may be catastrophic to the strength of the security (without an MRS) due to the privileges and scope with which these systems are required to run. Therefore, it is important that an MRS itself be analyzed and measured to build confidence that the MRS' functionality is not likely to be compromised and that the MRS is operating as intended.

1.2 Differing Perspectives on MRS Analysis

Because MRS are complicated and large, ad hoc methods of analysis such as fuzzing or pen testing may be insufficient to build a level of internal security confidence that one may require from an MRS given its highly privileged operational environment. While no system is flawless, there are techniques and tools that one can use to better discover these flaws and mitigate them as necessary. At the design level, any such tool or technique must enable an interested party to,

1. Describe the components of the system and how they relate to each other
2. Track how each component affects others
3. Apply measurable security claims to system components
4. Validate these security claims with evidence
5. Provide a means by which one can effectively convey that the analysis achieves the desired security goals

How one performs this analysis might differ based on the security goals or model one might take when analyzing an MRS. For example, a penetration tester (pen tester) might be interested in discovering vulnerabilities on a particular component or set of components of the system. The pen tester might also desire to understand how that vulnerability affects the security of that system with respect to remote code execution. The pen tester may use a fuzzer in conjunction with other automated tooling to perform a vulnerability analysis of the system. The pen tester might explore the system using source code, reverse engineering, or mentally given previous experience. This pen tester might also use some note taking tool to record their work and track what they have done.

A systems designer may be interested in understanding how components of the MRS work together so that the availability of the system is maximized or that the system fails safely. This system designer may utilize model checkers such as STRIDE to validate a collection of properties they wish to impose on the system. The designer might also utilize a methodology such as threat modeling to aide in structuring an analysis. Like a fuzzer in pen testing, model checkers often probe the state space of model inputs to check for deviation from expected safe conditions. The model the designer built may be the means by which they explore the system and tracking of work is typically done by saving the output of the model checker.

We describe a view of a system as an understanding of how that system works given a focus on one part of that system. For example, we may consider a view of an internet browser with a focus on communication security. In that view, we might choose to model how the browser interacts with web pages or other content within the context of communication security. We might also choose to focus on how each component of the browser might affect communication security. Often, an analyst will generate flow diagrams for each view of a system and may choose to use a text file or diagramming tool to visualize and track how this diagram may be affected by some action. While this is useful, it is difficult to track and connect relationships between different system views through the use diagrams together. For instance, understanding the relationships, interactions, and consequences of changing one system component on some other part of the system may not be easily represented in either component view or system view as a whole. Furthermore, there is often little or no consistency checking to ensure there are no errors in the diagrams. Importantly, tracking the impact of threats identified in one diagram with those in another can be a manual process and tedious process as that information is not tracked as part of the collection of diagrams.

Many types of analysis techniques often struggle with the complexity of multi-level systems. For example, MRS are built of three major components (client, manager, and responder). If one were to use model checking, fuzzing, or symbolic execution to discover vulnerabilities in the MRS, one would find that it could be difficult to move between views of the system as well as track how one effect on a view will affect another view.

1.3 Goals of MRS Analysis Tools

Ideally, an analysis tool or strategy would enable a repeatable and measurable means by which to evaluate assurance claims as well as track how changes in one part of the system affect assurance properties in other parts of the system. Within the context of MRS, these tools, techniques, and strategies should inform the user if the MRS contains design flaws or other issues that will affect its ability to fulfil its role as a security tool.

In each of these cases, an interested party wishes to understand an assurance property of a system and understand if that assurance property holds if affected by other components of that system. For example, suppose we have a client and server which communicate frequently and only under authentication and encryption. If unauthenticated data is sent by this client to this server, we would expect an error to occur and possibly a connection reset. Now suppose that the server had an edge case that when logging is enabled the server logs the error but does not reset the connection. In this example, we expected the server to report the error and perform a connection reset – our assurance properties – and it did not.

Fundamentally, those performing this analysis need tools, techniques, and strategies that, in a simple and straight-forward manner, allow them to,

1. Represent components of the system

2. Track the complex interactions and relationships among components
3. Reason about and validate assurance properties imposed on the system

In the client-server example above, we needed to represent the server and client as separate entities, track any interactions and relationships they were required to maintain – namely that they are authenticated to each other, and reason about any issues that could affect the system from reporting an error and resetting a connection.

1.4 Summary of Our Work

The work presented in this technical report aims to address model complexity, difficulties with bookkeeping and tracking effects, and modeling how an effect on one part of the system affects another part of the system. Importantly, this work attempts to also explore how one might begin representing a system such as an MRS such that one can evaluate and measure assurance properties that a system claims to preserve.

Fundamentally, these approaches help guide developers to understand what important questions they should be asking about their system concerning security. Importantly, these methods help analysts document these questions, assumptions, and answers in a consistent manner. To that end, one might ask: What principles will allow an analyst to decide whether a proposed MRS is sound, so that it can protect itself; deliver accurate information about the endpoints; and ensure that it helps security administrators control attacks, rather than the reverse?

Ideally, this work will spark new investigations into useful techniques by which analysts and designers can more easily build better MRS. We also hope that this work sparks investigation into the techniques we present here. The purpose of this report is to summarize a substantial line of work intended to develop principles for analyzing and appraising an MRS either after implementation or at the design stage. In particular, this line of work had three main steps.

First, we explored how to specify systems based on an inventory of the parts that made them up; these we specified as a datatype by a context free grammar. These parts included sensors, actuators, decision-making components, as well as the input and output channels that connect them, and connect them to the outside world (see Section 3). We found that this type of a catalog, while informative, did not emphasize the structure that would allow us to analyze whether its parts were doing the job that the security architecture of the MRS required of them.

Second, the grammar research motivated us to reinterpret the descriptions as directed graphs in which the arcs are the channels (see Section 4). With this in mind, we wanted to understand how we might manipulate a graph and formalize those manipulations so that we might eventually be able to transition between views of the system by invoking these manipulations. This transition work is left to future work (see Section 7). Representing the system as a directed graph has many advantages, since the executions of the system are driven by the values that flow over the channels. The graph structure thus bounds the ways that events can have causal effects on the different components (graph nodes) of the system. Moreover, many types of analysis require taking a closer look at the ways that the major components are built up out of their parts, connected by channels that were hidden inside the major component.

But what are these component-level security goals, how do test these goals, and how do we track this logic? To understand this, we moved to our third main step of research, the

idea of an assurance case (see Section 6). An assurance case is a piece of argumentation that shows why a system will behave properly despite a range of adverse events. An assurance argument is not a piece of formal logic, but a piece of careful informal reasoning that aims to identify the most important causal processes leading from these adverse events to failures of the system to meet its goals. In other words, *an assurance case is a structured argument, supported by evidence, intended to justify that a system is acceptably assured relative to a concern (such as safety or security) in the intended operating environment*[26]. Assurance cases helped us structure how we might translate the security constraints on a system into a system of logic that we can validate through evidence. Using a lightweight interpretation of Goal Structuring Notation[14] we explored how this translation might take place, how to start tracking that reasoning about security constraints, and how one might test the validity of these arguments. We do not explore how deeply one should describe an assurance case but do show that applying minimal effort to an assurance case can yield unexpected and useful results. For this work, we use an open source MRS that provided rich documentation and ease of deployment. One of the key benefits of this process is that it makes assumptions explicit. Generally, security properties hold only when the “relevant” assumptions are met. For instance, does it so to speak void the warranty if a system administrator gives non-trusted personnel login access to a security configuration server? If this matters, the assurance argument should find it, in which case it will lead to an explicit warning to the system administrator not to do so.

In Section 2, we provide a more detailed presentation of our motivation, and the train of thought that led us through the path we have just described. In Section 5, we work out the graph-based approach in the case of a simple weather reporting system. In Appendix A, we discuss other approaches and available tools. Section 7 discusses open problems and future work, after which we conclude.

2 Background / Motivation

It is well known that static defenses for our cyber systems are insufficient. Adversaries continue to find ways around our defenses allowing them to achieve their tactical and strategic goals. In recognition of that fact, an important current trend is to equip our systems with software and hardware designed to detect lapses in our perimeter defense by monitoring for potentially malicious activity. The real promise of such technology is that it can allow cyber defenders the opportunity to respond based on a combination of what they detect and the greater context of their operational situation. That is, by adding monitoring and response features to a system, we can increase its adaptability and resilience.

Many natural questions arise in the context of MRS. What should we monitor for? What does normal system activity look like? What constitutes convincing evidence of adversarial behavior? What is the best way to respond to an adversary? How should we measure or rank our response successes and failures? While these questions are vital to address, they focus on the internal logic of an MRS in isolation. The focus of the current work is complementary. The goal is not to aid the design of secure MRS, but rather to help design secure systems that have an MRS capability.

Anti-virus (AV) software serves as an illustrative example. As new malware is constantly being developed, it is of vital importance to ensure the use of up-to-date virus signatures. However, the AV vendors have realized this is insufficient. If the AV software runs as a simple user space process, advanced malware that has managed to subvert the underlying

operating system can easily evade detection even if the AV has a valid signature capable of identifying the malware. This is a major reason why AV software acts a little like a rootkit; it must “get below” the malware it is trying to detect. The success of the AV depends not only on the internal logic of the software (e.g. up-to-date signatures), but also on the manner in which that software is embedded into the larger system. That is, the internal logic is sound only if the (often implicit) assumptions used are somehow satisfied by the surrounding system.

Much work has gone into building MRS. The growing interest in them is indicative of the potential value they offer in maintaining the desired security posture of the systems into which they are incorporated. MRS systems tend to be complex systems with many interconnected components. It is difficult to reason about the effectiveness of these systems, and because there is no single definition of what constitutes an MRS, the reasoning that is done can only be in the context of single systems with no possibility of generalization across classes of systems. Research continues to advance the state of the art of MRS by developing new capabilities that might be incorporated with existing systems, but currently there is nothing to focus the research on tasks that might lead to better, more general improvements.

Our work seeks to discover a way to describe MRS in general so that it becomes possible to assess the strengths and weakness of any given MRS. Toward this end, we seek to identify different dimensions along which any given MRS might be assessed. We hope to be able to identify general properties in each of the identified dimensions that can be used to assess strengths and weaknesses of specific MRS. From this we hope to be able to better evaluate existing systems and to identify research areas that can lead to improvements to MRS in general.

2.1 Monitoring and Response Systems

In the field of cybersecurity there are many different mechanisms that can be used to defend computing systems from attack. One such mechanism used for many years has been intrusion detection systems (IDS) in which audit or other information is collected and analyzed for misbehavior. In this work, we refer to a class of systems called “Monitoring and Response Systems” or MRS which not only collect and analyze data but also include the ability to execute responses as well in support of security operations. Gartner has defined a similar set of systems called Security Orchestration, Automation, and Response (SOAR) platforms and provide the following definition of the area [16]:

SOAR refers to technologies that enable organizations to collect inputs monitored by the security operations team. For example, alerts from the SIEM system and other security technologies — where incident analysis and triage can be performed by leveraging a combination of human and machine power — help define, prioritize and drive standardized incident response activities. SOAR tools allow an organization to define incident analysis and response procedures in a digital workflow format.

At their core, these platforms enable analysts and operators a centralized interface to view and analyze data about what is happening and also execute automated or manual actions to respond to security threats across their network. These may be commercial offerings or home-grown solutions that tie together various other pieces of the security operations infrastructure into an easier to use interface.

Whereas SOAR platforms may refer to the centralized framework that glues all of the various components together, MRS refer to the broader class of systems that also include sensors, analytics, and response tools that can be controlled by SOAR platforms. We have identified four main functions of these tools:

- | | |
|---------------------------------|--------------------------------|
| 1. Sensing / Collection of data | 3. Reasoning / Analysis |
| 2. Data storage | 4. Responding / Taking actions |

Examples of sensing include monitoring network traffic, collecting host logs, or even ingesting threat intelligence information. Data storage can be the obvious repositories such as SIEMs but also includes other storage such as temporary caching before data is transmitted and the storage of configuration information. Reasoning and Analysis is the broadest category and represents any sort of computation that may take the collected data as input. Finally, response actions may take the form of changing firewall rules, killing processes on endpoints, or adjusting access control policies. Through the combination of these four functions, MRS can be constructed to create various different types of adaptive systems.

2.1.1 Adaptation Loops

While adaptation and control loops have been studied in many different application domains, we would like to highlight two areas that influenced our thinking about MRS: autonomic computing and military strategy. In [18], Kephart and Chess discuss the vision IBM had for the new domain of “Autonomic Computing”, which was created to address the growing complexity of computing systems. This vision was for “computing systems that managed themselves given high-level objectives from administrators” and included abilities for self-configuration, self-optimization, self-healing, and self-protection. Figure 1 depicts a central piece of this vision known as the adaptation loop or Monitor-Analyze-Plan-Execute-Knowledge Base (MAPE-K) loop. This adaptation process also included the idea of sensors and effectors for interacting with other parts of the computing system and also included notions of many adaptation loops operating together in a self-organized fashion. While the overall goals of the vision may have been lofty, the adaptation loop concept has strong alignment with MRS concepts as described above.

Military strategist and U.S. Air Force Colonel John Boyd developed the OODA loop theory of decision making drawing on lessons learned as a fighter pilot. This theory says that decisions are made in a four-phased cycle of Observe-Orient-Decide-Act and is depicted in Figure 2. In addition there is also a strong emphasis on incorporating feedback between the different phases as well as outside information such as cultural traditions and unfolding circumstances. Beyond dogfights Col. Boyd advocated for application of this decision strategy in many tactical and strategic situations.

These two concepts have a similar four phased approach to interacting with environments but have somewhat different focuses with MAPE-K focusing on adapting IT systems and OODA having its origins in adversarial engagement. This combination of origins aligns well with the goal of security operations in which IT systems much adapt in response to attacks. Based on these areas, we have developed our own abstract model of MRS that is briefly described in Section 2.3 below and in further detail in Section 3. Next we describe the motivating goals of this work and how the study of adaptive systems can help us improve the design and implementation of MRS.

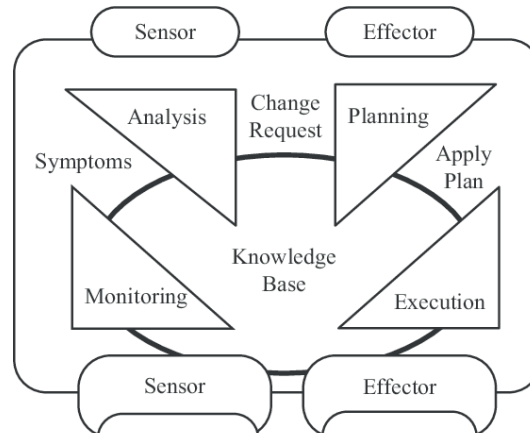


Figure 1: The MAPE-K Control Loop[18]

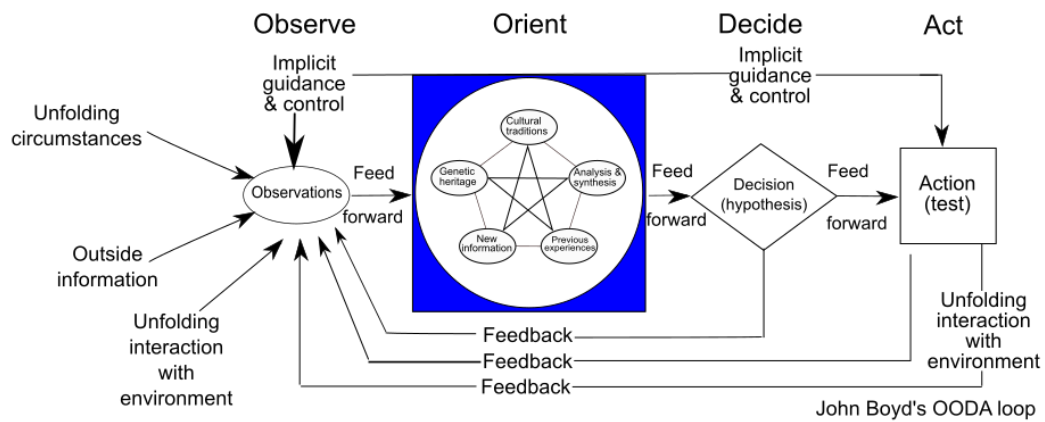


Figure 2: Diagram of the OODA Loop[24]

2.2 Motivating Goals

We have developed two motivating ideas that we hope will facilitate discussion and analysis of MRS in general. By combining these two ideas it should be possible to create abstract descriptions of MRS that lead to common ways to describe MRS, their core functions, and architectural design. Ultimately, we hope it might enable objective evaluation of MRS.

The two motivating ideas for creating abstract descriptions of an MRS are:

1. **MRS as a Maintainer of Beliefs:** The first idea is that it is possible to generalize the idea of an MRS system with a belief system abstraction. An MRS can be considered a system that maintains a collection of beliefs about the world outside it. All actions taken by the system are motivated by its held beliefs. It senses the outside world to test held beliefs, reflecting on their validity and modifying them as needed. Alternatively actions to modify the outside world might be necessary to ensure that it continues to conform to strongly held beliefs.
2. **MRS as a Collection of Generalized Components:** The second idea is that a generalized monitoring and response component can be defined by identifying common functionality found in all MRS. With such a component, it becomes possible to reason about MRS abstractly without having to fully consider the complex range of possible systems that might be identified as MRS. As it turns out, by creating a simplified general monitoring and response component, it becomes possible to describe complex MRS as systems of these interconnected generalized components, each specializing as some portion of the larger system. We believe this will enable better reasoning about both individual parts of the system as well as the system as a whole.

In our exploration of these two ideas, we identified three main requirements that needed to be developed. These requirements correspond to the three major contributions of this work. With these three items, we believe we can analyze the complex nature of MRS without being constrained to specific evidence generation techniques and independent of implementation specific details.

The three requirements needed to analyze complex MRS are:

1. **Abstract model:** The first requirement was to have an abstract model of an MRS to reason over. This comes directly from the ideas above as an abstract model will allow us to generalize concepts in MRS and allow more broad application of analysis and findings. The idea of abstract models of computing systems is not new, with [20] and [4] presenting methods for documenting complex software systems and creating multiple views of the system to better understand and reason about its design. Additionally, in Appendix A we cover the area of Model-based Systems Engineering which takes this concept further. Overall, we require a model of a system in order to be able to reuse our analysis techniques across different MRS. A high-level overview of such a model is described in Section 2.3 and an in-depth description is provided in Section 3.
2. **Navigation:** Second, we needed a way to navigate through this model to support reasoning about different portions of a system. As we looked at MRS implementations, we found that while two systems may look very similar at a high-level view, small changes in low-level details may result in drastically different security guarantees. Therefore we needed the ability to transform the model as we zoomed into low-level

details, or obscured design details that were not important to the current analysis at hand. Section 4 discusses our application of graph-based modeling techniques to allow for navigating a model of an MRS.

3. **Bookkeeping:** Finally, as we were interested in asking many different questions of an MRS at many different resolution of views, we needed a method of bookkeeping to keep track of both the different views we were using but also the assurance arguments each view supported. Section 6.2 discusses our exploration of Assurance Cases as a way to perform this bookkeeping by documenting the arguments that were constructed as we navigated and analyzed the model.

2.3 Abstract MRS Component Model

We view the generalized MRS component as a generic component that describes any part of the MRS. It is our intention that some instance of this component could be used to describe the whole system or any portion of the system. It can be useful to think of this component as a view of some logical portion of the system. That portion should be able to be viewed as a maintainer of a belief system relevant to that view of the system. Processing that occurs internal to that component is only relevant to that component's point of view. Information may flow in or out of that component to interact with other components that encompass other portions of the MRS. A notional diagram is given in Figure 3.

This way of thinking about the MRS is intended to be a logical breakdown of the system to enable reasoning about that portion of the system that the component represents. It is not intended that the breakdown should correspond to physical system components or interactions between them even if sometimes it might be useful to describe the larger system in that way. As such, the internal structure of any instance of a generalized component might represent portions of the physical system that spans across physical boundaries and interactions between components, and the flow of information between or within components should be considered discrete events rather than data traversing physical communications paths.

2.3.1 Interactions with the component

Interactions with the generalized component, considered boundary crossings, can be classified in three ways: a data transfer, a control signal transfer, or direct interaction with the world external to the entire system. Data transfers can be bidirectional, either reporting or ingesting data to or from other components. Data transfers represent exchanges of knowledge between components. The source component is reporting information about its current beliefs whether they are directly observed facts, hearsay from other components, or statements of beliefs contained in its knowledge base. The receiving component sees the transferred data as information that it can add to its knowledge base or perhaps use to influence updates to its current set of beliefs about the world beyond its boundary.

Transfer of control signal information is similarly bidirectional. Control signals entering a component are recommendations, or perhaps commands, to take some action. Control signals being sent are directions to other components that action on its part is required.

The last class of interactions is reserved for those interactions not involving other components. They are interactions directly with the world beyond the entire system. These too are bidirectional. The component may ingest data in the form of direct observations about the world. It may also take actions the directly impact the outside world.

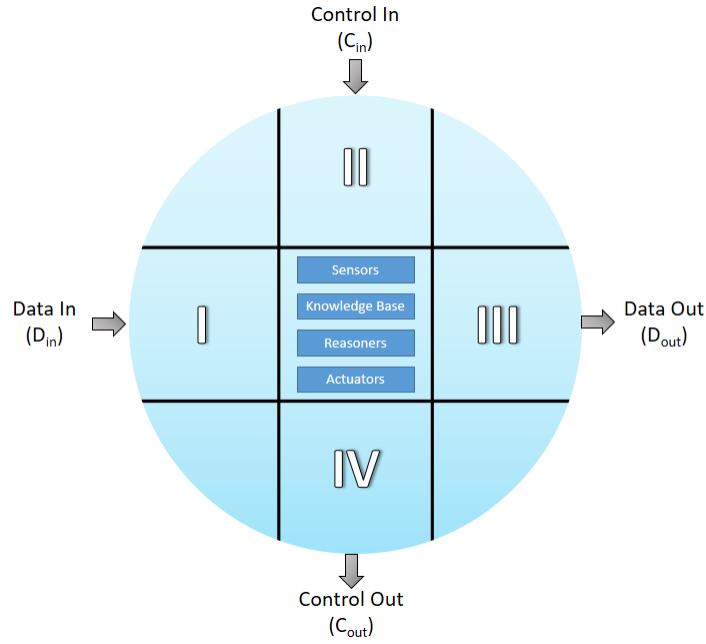


Figure 3: Abstract MRS component

Although similar in nature to the first two classes of interactions in that they are data/control interactions with the same relationship to the component's knowledge base, it is useful to call separate attention to the third because they indicate independence from any other components of the system. They best correspond to the specific idea of physical sensors and actuators found in MRS.

2.3.2 The component's belief system

As mentioned earlier, each component manages a belief system about the world beyond its boundaries, a representation of the world from its point of view. It uses interactions with other components and directly with the world to maintain its beliefs. It is the idea of maintaining beliefs that drives the function of the component. Data/control ingested tests/modifies beliefs; data/control exported shares a point of view or attempts to modify the external world to conform to its beliefs.

The components have functions to perform which may affect their beliefs. The following are some examples of the relationships between a component performing its function and the component's beliefs. For example, ingesting data from other components will cause the data to be added to the component's beliefs. Also, forwarding data/reporting will cause that data/reporting to be added to the receiving component's beliefs.

2.3.3 Knowledge base

Each component's belief system is contained in a knowledge base. This is a logical notion that is not necessarily intended to correspond to any physical data store. Rather it is

the logical embodiment of everything that the component needs to know to perform its functions. An example of a knowledge base's contents are:

- raw data
- processed data
- configuration data
- rules governing reasoning
- ingested control information
- data needs
- planning information
- pending actions

Interactions with the component are discrete interactions with some portion of the knowledge base. Any reasoning that the component might do is about information contained in the knowledge base and results in possible updates to the knowledge base.

2.3.4 Reasoning

The component has a general reasoning capability that processes information contained in the knowledge base. The reasoner considers the current state of its beliefs and plans a course of action. Reasoners might draw conclusions to update beliefs. They might also plan a course of action including such things as taking direct action to impact the outside world, directly seeking more information about the world through observation. More information might be requested from a connected peer or a notification might be made to some peer about observations. All these go to the idea of maintaining its beliefs.

Specific functions of the components can be thought about as specific types of reasoning done with respect to specific portions of the knowledge base. Reasoners may be complex requiring lots of data about the world and extensive rules about how to interpret it or so simple as to represent hard coded decisions about how to process a single thing. To create a generalized MRS component it was useful to generalize the idea of reasoning to include every decision process, no matter how trivial, as a distinct type of reasoner.

Some examples of the different types of reasoners are below.

- goal planners
- action planners
- things that govern interactions
- true analytics on data
- decisions to accept commands.

2.3.5 Beliefs and Reasoners

We mentioned above the possibility of decomposing beliefs and reasoners into different types. In this section, we explore a more concrete proposal for such a decomposition. Following the observation from the above discussion, we identify a type of belief that contains “unprocessed data” which results from copying signals from the input channels or sensors. Similarly, we identify a belief type that contains “intentions” which are copied to output channels or actuators. These correspond to the Data and Action classes in the above discussion. As for the intermediate Beliefs, we identify a preliminary set of belief types with the aim of helping to organize types of reasoners. Including the Data and Action types, the full preliminary decomposition of types of belief is the following:

- | | |
|-----------------------------------|------------------------------------|
| 1. Unprocessed data | 6. State/configuration information |
| 2. Desires/goals | 7. Processed/annotated data |
| 3. Beliefs about environment | 8. Plans |
| 4. Beliefs about (sub-)components | 9. Intentions |
| 5. Beliefs about data | |

The above decomposition of an MRS component is depicted in Figure 3. The figure also identifies a high-level classification of types of reasoners that can exist. We think of the processing pipeline as implementing an OODA loop. The simple input and output reasoners that copy from and to channels, sensors, and actuators represent the Observe and Act phases. Reasoners that create or update beliefs of types 2—8 represent the Orient phase. Reasoners that update the intentions represent the Decide phase.

To obtain a complete decomposition of types of reasoners, we need to establish a minimal set of function types among the beliefs that is sufficient to represent any possible function type. That is, we are looking for a kind of “basis set” of function types from which any function type can be obtained through composition. “Atomic” reasoners would be those whose type is an element of this set, and we could impose a restriction on MRS models that forces each MRS to enumerate the reasoners of each of these types (many would be empty). We hypothesize that each atomic reasoner type will fit cleanly in one of the OODA types. Identifying a complete decomposition of reasoner types is still an open problem that we are investigating.

2.4 MRS Case Studies

Throughout the rest of this document, we explore various techniques for improving the analysis of MRS. A series of three case studies are used to demonstrate the application of these techniques. The first two case studies explore systems that are not related to cybersecurity. This was an intentional decision as it helps to demonstrate the concepts on simpler systems that do not require deep understanding of security monitoring and response operations. However, this also illustrates that monitoring and response systems are not limited to just the security domain, though that is the primary focus of our interests. These case studies are described here to set out the initial systems and how they relate to the tasks of monitoring and response. Future sections may reference these initial descriptions.

2.4.1 Thermostat

The first case study we explore is a simple heating and cooling system. We use this example because of the small number of sensor types (i.e. Thermometer), actuator types (i.e. Heater, Cooler), and reasoning (i.e. Thermostat) required. With this simple set of components we can generate a range of examples that span simple (e.g. a single thermometer) to complex (e.g. multi-room measurement and control).

The main goal of the system is to take a desired temperature setpoint as input and then use the heater and/or AC to ensure the current ambient temperature stays close to this desired temperature setpoint. Figure 4 shows the four main components of the system. The thermometer acts as sensor from the external environment. Raw readings are used by the Thermostat to make decisions about what actions to take. Once the decisions are made,

the Heater and Cooler are toggled in order to achieve the desired temperature. While this can be a simple system with only a single instance of each of the components, it can also be easily modified by adding additional components such as multi-room zones. Section 3.2 will use this case study to demonstrate the grammar-based model we have developed and will modify the system as needed.

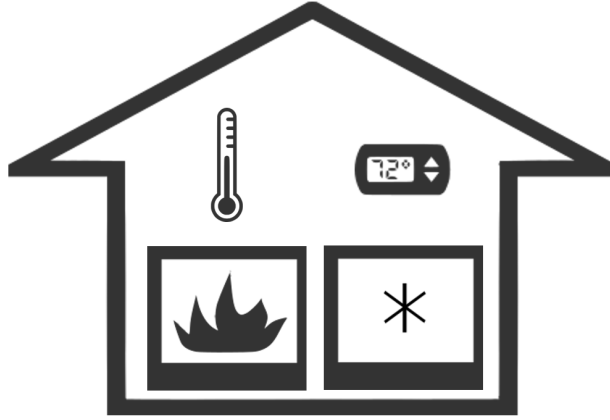


Figure 4: A Simple Thermostat System

2.4.2 Weather Service

We use a weather service system as a case study for demonstrating our technique in analyzing the properties of an MRS. We do not claim the design presented here is faithful to that of, say, the National Weather Service. The weather service system is comprised of two major components, the weather service center and the local weather stations that communicate data to and from the weather service center. Each component is responsible for performing some specific high-level functions. The weather service center is responsible for ingesting data from the weather stations, using the data it has collected to create weather forecasts for each weather station location, and reporting the forecasts back to the weather stations. Each weather station is responsible for collecting weather data from its sensors and reporting this information to the weather service center, ingesting the weather forecast information from the weather service center and extracting the forecast for the location of the weather station (the weather service center is assumed to broadcast forecasts for all its weather stations to simplify communications), and displaying the weather forecast.

Each weather station collects data about its location that it sends to the weather service for processing. The weather service uses the data from all the weather stations along with some additional information to create a weather forecast for each weather station location. The weather service then broadcasts the bundled forecasts for all stations, and each station ingests the broadcast and extrapolates its local forecast which is then displayed. A high-level view of the weather service system showing the basic interactions between the weather stations and the weather service is depicted in Figure 47c. This represents the most abstract or zoomed-out view of the system, and it highlights only the highest-level dependencies among the components. Section 5 will explore this system in further detail using a graph-based approach.

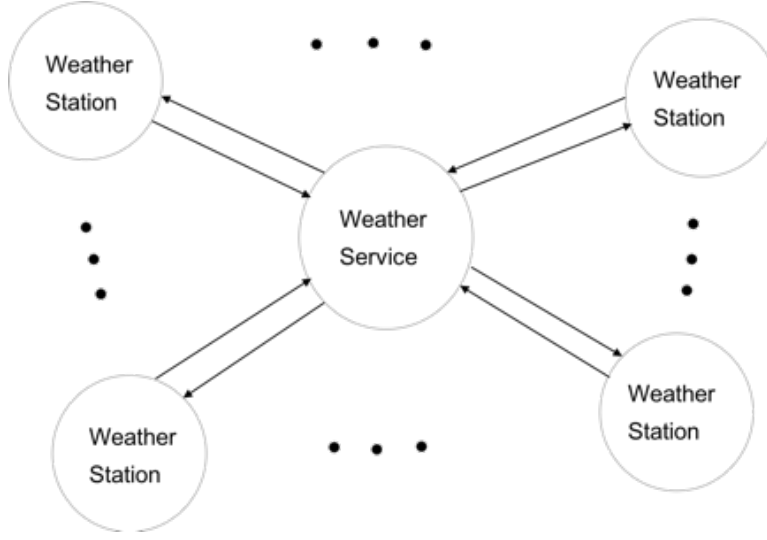


Figure 5: High-level Weather Service System View

2.4.3 OSSEC / Wazuh

OSSEC [23] is an open-source host-based intrusion detection system (HIDS). It provides a means for collecting and analyzing log-based information from endpoint devices within a network. Further, it also includes an “Active Response” capability which allows for execution of various responses on endpoint devices driven by the central management server. OSSEC has been in development since at least 2005 and is used extensively throughout cybersecurity community.

However, in 2015 the Wazuh project [35] started as a fork of OSSEC, citing a lack of new features and mainly maintenance updates for bug fixes. Since then, the Wazuh project has been quickly adding new features and combining the core of OSSEC with additional tools in the cybersecurity community. For example, Wazuh makes heavy usage of the Elastic stack [10] to provide an updated interface for reporting and exploring the data collected by endpoint agents.

At a high level, the Wazuh system sits within a larger computer network, collecting data from individual computers and analyzing that data in a central repository. The major system components are therefore the centralized “Wazuh Server” and the “Wazuh Agent” which is deployed to many machines throughout the network. Figure 6 provides an in-depth look at the components within the Wazuh agent. For now, the main takeaway is that the agent is made up of multiple daemons which provide various functionality to the agent and allow it to perform the primary tasks of log collection, active response, and communication with the server. The agent can perform the abstract functions of a sensor by collecting data, an actuator by executing active responses, beliefs by storing data locally before forwarding, and a reasoner as it manages configuration information and adapts its actions accordingly.

Figure 7 provides an overview of the Server. Again, the server is made up of multiple daemons which provide various sub-functions that enable the server to communicate with agents, apply decoders and detection logic to the incoming data, and interact with external

tools such as SIEMs. Figure 8 zooms out a bit and shows the flow of data between agents, the server, and the Elastic stack tools.

Overall this system is a great reference for various MRS within the cybersecurity domain. As can be seen from this high-level introduction, these systems can be very complex and consist of many different pieces. This case study will be used in Section 6.2 to demonstrate the need for various bookkeeping techniques to track multiple views of the system as well as the different assurance arguments necessary to support the overall trustworthiness of the system.

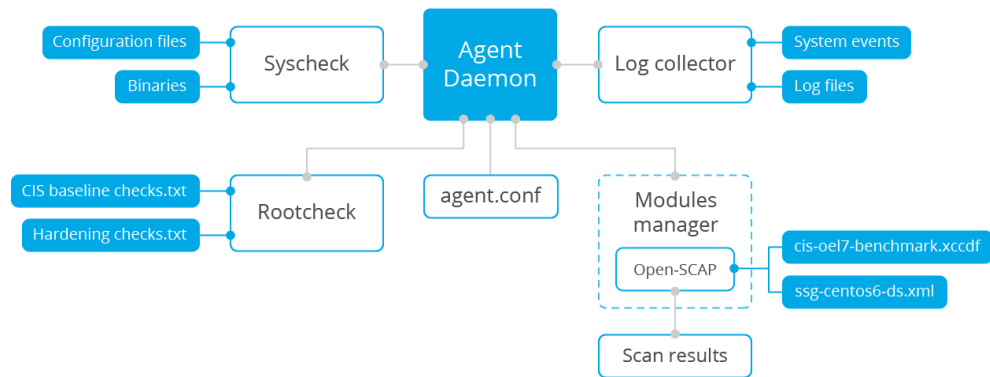


Figure 6: Wazuh Agent Overview [35]

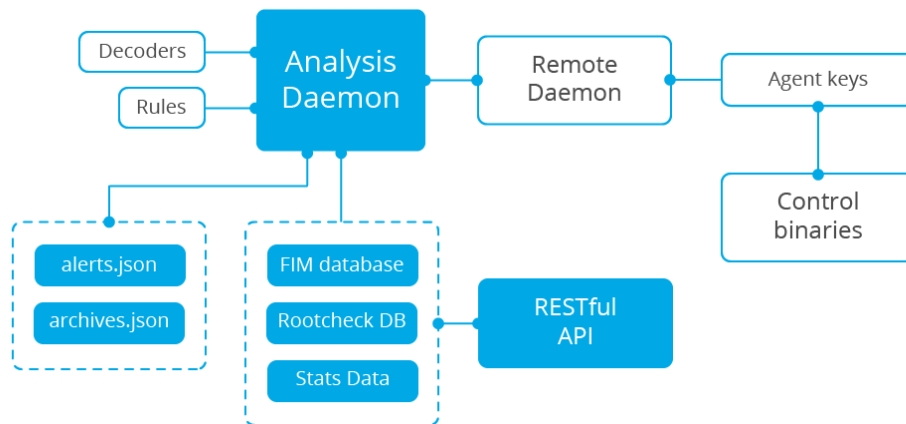


Figure 7: Wazuh Server Overview [35]

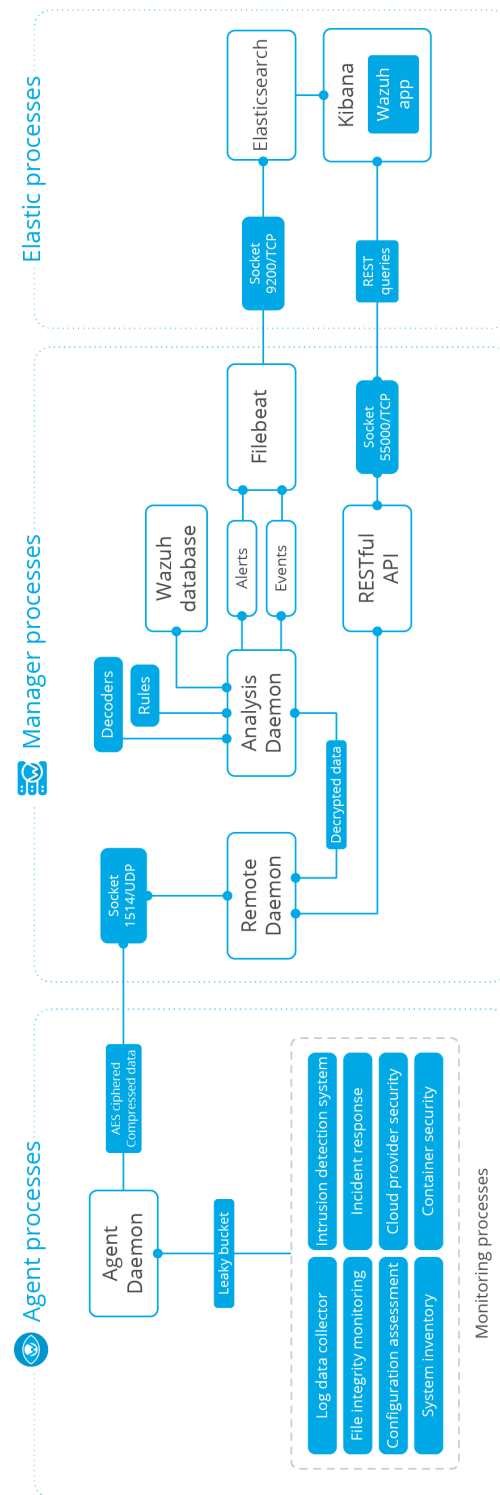


Figure 8: Wazuh Dataflow [35]

2.5 Summary

In this section, we have shown that MRS serve an important function keeping track of security telemetry information across an enterprise network and allowing security engineers to rapidly respond to threats that may be observed. In addition to the MRS of today, we have also looked at how adaptation loops can inform what those systems may look like in the future. Further, MRS represent a security-critical system both in the services provided and the additional attack surface that they add to the environment. Therefore, we advocate for an improved methodology that makes use of system models, an ability to navigate to many different views of those models, and a method to keep track of both the different views of and arguments made in support of the system. We then provide some background on our thoughts for what would make a useful model to achieve these goals with further details presented in later sections. Finally, we present three case studies that will be used throughout the paper as representative MRS.

3 A Grammar-based Model

This section discusses the grammar-based MRS model described in Section 2.3 and is illustrated with the thermostat example discussed in Section 2.4.1. In order to accommodate different scales of representation of the system under examination, we developed a grammar-based model for an MRS. This allows the flexibility of representing the same system at a coarse-grained (large) scale or at a fine-grained (small) scale. It will be important to maintain a sense of consistency among representations of a system at various scales. Roughly speaking, given a large-scale and a small-scale representation of the same system, we should be able to identify which sub-systems of the small-scale representation correspond to given (atomic) components of the large-scale representation. The representation should also be structured enough to permit (in the limit) a single representation at any given scale. The grammar-like representation of monitoring and response systems allows for some recursion enabling us to roll-up several subsystems into a larger one.

This work represents our first attempt to model an abstract MRS; it is not a finished product as our work evolved into a graph-based model (described later in Section 4) in order to better keep track of how changes in one part of the system affect other parts of the system.

3.1 Syntax

The following form the base components of an MRS as envisioned in Section 2:

```
C :: atomic_channel | MRS
B :: atomic_belief | MRS
R :: atomic_reasoner | MRS
S :: atomic_sensor | MRS
A :: atomic_actuator | MRS
```

We can further subdivide channels based on their type. Two main dimensions exist for describing channels: direction (input or output) and information type (data or control). For simplicity we will not differentiate between the two information types. It makes sense to think of atomic sensors and actuators as channels from and to the environment, respectively. However, as we will see below, an entire MRS subsystem might act as a sensor. So, we

identify atomic sensors and actuators as distinct from channels. Notice that we consider it possible for any of the components to be considered an entire system unto itself. For example, we could also envision a channel as being some complex information processing pipeline made of beliefs and reasoners connected by simpler channels. Similarly, complex reasoning tasks might be decomposed into smaller reasoning tasks that communicate among themselves. We can view the set of beliefs of a given component to be its knowledge base as envisioned in Section 2.

Any given MRS will not just have one of each type of component, but rather it will have a (possibly empty) set of them:

```

Cs ::  $\emptyset$  | C, Cs
Bs ::  $\emptyset$  | B, Bs
Rs ::  $\emptyset$  | R, Rs
Ss ::  $\emptyset$  | S, Ss
As ::  $\emptyset$  | A, As

```

We can then represent an MRS as a tuple of sets of components of these various types. That is, an MRS is represented by the following:

```
MRS :: (Cs; Cs; Bs; Ss; As; Rs)
```

The two sets of channels in the first two slots represent the input and output channels respectively. Written out more explicitly, an MRS is a tuple:

```
(Ins; Outs; Beliefs; Sensors; Actuators; Reasoners)
```

3.2 Example 1: Decomposed Thermostat

To exercise the above syntax, we begin with a very simple thermostat system initially described in Section 2.4.1. It might be thought of as consisting of 4 sub-MRS: **Thermometer**, **Thermostat**, **Heater**, and **Cooler**. These could be defined as follows.

```

Thermometer = ( $\emptyset$ ; temp_report;  $\emptyset$ ; temp_in;  $\emptyset$ ; copy_signal)
Heater      = (h_in;  $\emptyset$ ;  $\emptyset$ ;  $\emptyset$ ; heat; toggle_h)
Cooler      = (c_in;  $\emptyset$ ;  $\emptyset$ ;  $\emptyset$ ; cool; toggle_c)
Thermostat  = (temp_report, temp_set;
               h_in, c_in;
               desired_temp, h_state, c_state;
                $\emptyset$ ;
                $\emptyset$ ;
               set_temp, toggle_heat, toggle_cool)

```

In this representation, all the reasoners, sensors, and actuators are atomic. Syntactically, the atoms are just names. We need to give them meaning by defining how they are expected to act:

`copy_signal` reads a signal on `temp_in` and copies it to `temp_report`.
`toggle_h` reads signals on `h_in` and toggles the heat actuator.
`toggle_c` is completely analogous to `toggle_h`.
`set_temp` reads a signal on `temp_set` and updates the value of `desired_temp`.
`toggle_heat` reads the signal on `temp_report`, the value of `desired_temp`, and the value of `h_state`, and chooses (or not) to send a control signal on `h_in` and update (or not) the value of `h_state`.
`toggle_cool` is completely analogous to `toggle_heat`.

Abstractly, the reasoners are processing functions that consult (and possibly consume) information found in some of the other components (input channels, beliefs, sensors) and produce outputs that get placed in some of the other components (output channels, beliefs, actuators).

This level of description could inform an executable model in which the reasoners concurrently loop through their processing function. Some details would have to be worked out. As one example, when a reasoner uses a belief as input, it may make a difference whether that belief is consumed (i.e. erased) or remains for another reasoner to use later.

3.3 Example 2: Single-Component Thermostat

This decomposition of the total system into four separate components is natural, but it also makes sense to attempt to represent the whole system as a single MRS. The most natural and naïve attempt would be to take the union of each of the components of the tuples. That is, the set of input channels would be the union of the input channels of the subcomponents, etc. The result for the above example would be the following:

```

Whole_Thermostat =
(h_in, c_in, temp_report, temp_set;
 temp_report, h_in, c_in;
 desired_temp, h_state, c_state;
 temp_in;
 heat, cool;
 copy_signal, toggle_h, toggle_c, set_temp, toggle_heat, toggle_cool)
  
```

Unfortunately, although the approach is natural, the result is somewhat unnatural. For example, `h_in` becomes both an output and an input channel. It seems unnecessary to “externalize” this signal. Rather, this channel could be represented as a new belief variable in the combined system. This approach also does not take advantage of the recursive quality of the grammar-like representation. Put another way, although the above attempt represents the system as a single component, we have not really abstracted the system. The representation is still at the same scale as Example 1. We could try instead to call the thermometer from Example 1 a sensor, and the heater and cooler actuators. Then we would write something like:

```

Whole_Thermostat =
(temp_set;
 temp_report;
 desired_temp, h_state, c_state;
 thermometer;
  
```

```

heater, cooler;
set_temp, toggle_heat, toggle_cool)

```

If the thermometer, heater and cooler are given atomic definitions as primitive functions, then we performed an abstraction of Example 1. On the other hand, if those three components are names for the (sub)-MRSs they represent in Example 1, then this representation could be thought of as “expanding” into the decomposed version of Example 1. This, too, is a little unnatural. In a sense, it hides the channels from the sensor and to the actuators. It also does something funny to the types of the reasoners that are visible. It changes the type of data they work on. For example, in the decomposed version, `toggle_heat` reads from a data input channel. Here it really reads from the thermometer which is more of a sensor input channel. Details such as these bear on the consistency among different representations of a given system at different levels of abstraction. If one goal is to seamlessly navigate among different views of the system, then these details necessarily play an important role. We can think of them as defining the principles of abstraction and refinement of MRS. We could try to define a formal consistency condition and even prove that the (yet-to-be-discovered) principles of MRS abstraction and refinement yield consistent results. We do not undergo such a task in this document. Instead we present it as a challenge for future research.

3.4 Example 3: Temperature Aggregation Thermostat

As another example, consider a more complex thermostat system that receives signals from several different thermometers and aggregates their reported temperatures to create a belief of the current temperature before deciding how to respond. A decomposed representation of this system would have n thermometers of the form:

$$\text{Thermometer}_i = (\emptyset; \text{temp_report}_i; \emptyset; \text{temp_in}_i; \emptyset; \text{copy_signal})$$

The big difference from Example 1 is in the thermostat. It might be represented as follows:

```

Thermostat =
(temp_report_1, ..., temp_report_n);
h_in, c_in;
desired_temp, h_state, c_state, current_temp;
∅;
∅;
set_temp, toggle_heat, toggle_cool, aggregate_temps)

```

The reasoner `aggregate_temps` is an atomic reasoner that reads signals from each thermometer `temp_reporti` and updates the `current_temp` belief variable according to some function. Different functions are possible, and exactly which one we choose is of little importance for our current purposes.

The heater and cooler would remain the same as in Example 1.

Since we have expanded the functionality, it is not surprising that we have found need to add a new reasoner, viz. `aggregate_temps`. What is interesting, is that we now have to change the types of `toggle_heat` and `toggle_cool`. In Example 1, they read information from input channels. They now read the same information from a belief variable.

This suggests that we have not yet identified all the aspects of modeling choices that

are important to retain consistency among related MRS. To see this, consider an abstracted version of this MRS in which the collection of n thermometers would be collectively represented by a single sensor (and maybe the `aggregate_temps` reasoner). That is, it would look the same as the second representation in Example 2, but the atomic sensor functionality would be altered to account for the multiplicity of input signals. We would prefer such an abstraction to yield reasoners as the same type as those in Example 2, but for the reasons described above, we get reasoners of different types. Principles for this kind of modular replacement of subparts should also lead us to some notion of consistency of representation.

3.5 MRS Execution

The idea of using a grammar-like representation of MRS is most useful for statically describing the general paths of information through an MRS. It is important to keep in mind the importance of leveraging such descriptions to identify properties of systems and components that are required to support the goals of an MRS. An important piece of the puzzle is understanding how the description of an MRS translates to its underlying dynamics. The grammar-like description has no underlying operational semantics to define system behaviors. One could imagine augmenting the static descriptions of this section with an explicit execution semantics for atomic items. The grammar suggests principles for how such atomic execution semantics should compose with each other to generate global system dynamics. Many of the issues discussed above regarding consistency between system descriptions would be exacerbated by the need to keep the dynamics consistent as well. We believe that identifying a comprehensive descriptive framework that allows for consistency at all scales of representation could be a valuable line of research.

3.6 Summary

The grammar-based model for an MRS in this section facilitates representing an MRS at different scales and enables us to roll-up several sub-MRS into a larger one. We chose the grammar-based model for the capabilities it provides an analyst while examining an MRS. We attempted to model MRS in other ways, but the end result did not provide the desired MRS analysis capabilities. For example, we developed a model for MRS with a purely Belief–Desire–Intention [37, 36] focus, but decided not to pursue it further for this work because it was more complex and did not allow for representing an MRS at different scales.

In order to develop a grammar-based model for an MRS, we present a syntax that includes definitions of the five base MRS components. We use the base components to build a grammar-based representation of an MRS as a tuple of sets of base components. We divide the channel component into two distinct groups, input and output, based on the direction of flow of information through them. Grammar-based MRS representations are useful in general to describe general paths of information through an MRS. However, they do not provide the desired capability to navigate between representations of an MRS in order to explore some security property of the system.

Examples in Sections 3.2–3.4 illustrate a grammar-based representation of a thermostat system viewed as an MRS. The examples demonstrate multiple representations of the thermostat MRS which was shown to be represented as a single MRS and also as a collection of multiple sub-MRS.

As we were continuing to develop the methodology for building a grammar-based MRS model, we pivoted to graph-based MRS models once we realized there was a body of work

on graph-based representations and graph-rewriting (used on a particular individual system representation to transform it to another while keeping track of these transformations) that our existing research better resembled. We want to be able to focus in, pan across, and occlude parts of the system in a particular system representation; graph-based models fit these needs better than a grammar-based model. Additionally, we found a need for tools to create grammar-based MRS models that already exist for graph-rewriting (see Appendix A) which make graph-based MRS models more attractive. A graph-based MRS representation, as described in the next section, allows an analyst to move from one MRS representation to another.

4 Graph-based Model Manipulation

In this section we introduce an approach to modeling systems that aims to facilitate the construction of global security arguments for a fixed, static system design by combining local security arguments for different pieces of the system. The core idea is to navigate through various local graph-based representations called views (see Definition 13 in Section 4.1) of a given design in a structured manner in order to move around different parts of the model. Each view will highlight certain dependencies and interactions that suggest to an analyst reasons to believe the design achieves (or fails to achieve) important local security properties. Representing the system with different views allows an analyst to manage system complexity and focus only on the relevant pieces. For example, one view may highlight the dependency between an AV tool and its signature file, while another view may zoom into details about how that software gains read-access to the important parts of the file system. The manner in which views are combined is intimately connected to the manner in which one navigates among the views using some combination of transformations listed in Section 4.2.

To enable this kind of navigation between views, we explored the theory of graph transformations, also known as graph rewriting since grammars did not have this level of maturity. We planned to leverage existing graph rewriting tools and research to help with our work (see Appendix A.2). To make use of graph rewriting requires defining a graph that represents a particular view of the system. The theory then allows us to specify transformation rules that transform some specific substructure of the graph into a different structure while leaving the rest of the graph unchanged.

4.1 Lexicon

This section presents formal definitions that will allow us to be clear and precise in our graph-based representations of a system as well as how allowed transformations of the graphs affect the representation of the system under study. In particular, we pay close attention to the relationship between components (and groupings of components into larger “components”) representing different functional aspects of the system.

We assume some basic familiarity with the fundamental notions of directed graphs. All graphs and edges are assumed to be directed. We note up front that graphs and subgraphs are not required to be connected.

Definition 1 (Node Type) *The type of a node n is τ (written $n : \tau$) where τ is one of $\mathcal{T} = \{s, b, r, a, i, o, g, c\}$. s represents a sensor, b represents a blackboard, r represents a reasoner, a represents an actuator, i represents an input channel, o represents an output*

channel, g represents a “summary” of an arbitrary subgraph, and c represents a summary of a subgraph understood to be a component (Def. 7). An element whose type is either i or o is called a channel endpoint.

Definition 1 is straightforward. It simply introduces the types of nodes. Figures throughout the remainder of the document will use the representations of the node types shown in Figure 9. Sensors are trapezoids with the left and right sides parallel, blackboards are rectangles, reasoners are diamonds, actuators are trapezoids with the top and bottom sides parallel, and input and output channels are cylinders.

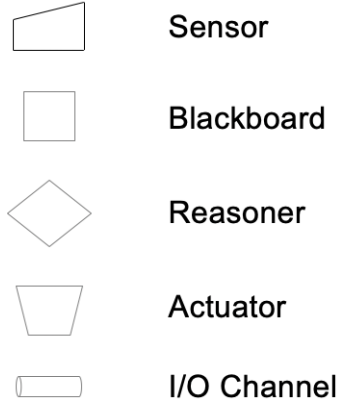


Figure 9: Node Type Representations

There are a couple things to note. First, compared with Section 3, we use the term *blackboard* instead of *belief*. This arises out of viewing it as a component that stores values. Second, we have added nodes of type g and c to the list of acceptable node types compared with those we found in Section 3. We generally use nodes $n : g$ to represent parts of the system whose internal structure is not relevant in a particular view. They will typically arise as a result of graph transformations that contract a subgraph into a single node (Def. 5). Nodes $n : c$ are similar to nodes of type g , but they will be used to represent subgraphs that form components as defined in Def. 7. This relates to the recursive nature of the grammar of the previous section.

Definition 2 (Edge Type) *The type of an edge $e = (n_1, n_2)$ between nodes n_1 and n_2 is $\tau_1 \rightarrow \tau_2$ (written $e : \tau_1 \rightarrow \tau_2$) where $n_1 : \tau_1$ is of type τ_1 , $n_2 : \tau_2$ is of type τ_2 , and $\tau_1, \tau_2 \in \mathcal{T}$. We say an edge e is a channel edge iff $e : \tau$ where $\tau \in \{o \rightarrow \tau', \tau' \rightarrow i, c \rightarrow c\}$ for $\tau' \in \mathcal{T}$*

The type of an edge is determined by the types of its source and target nodes. We identify channel edges as those edges coming from an output channel node $n : o$, going to an input channel node $n : i$, or otherwise going between two nodes of type c . We use channel edges in Def. 7 to help us define the notion of a component, roughly, the circles of Section 2.

Definition 3 (Hybrid Graph) *A graph $G = (N, E)$ is a hybrid graph iff each $n \in N$ has type in $\mathcal{T}$, and each edge $e = (n_1, n_2) \in E$ such that $n_1, n_2 \neq g$ has type $\tau_1 \rightarrow \tau_2 \in \mathcal{S}$ for*

$$\mathcal{S} = \{i \rightarrow s, s \rightarrow b, b \rightarrow r, r \rightarrow b, b \rightarrow a, a \rightarrow o, o \rightarrow i, o \rightarrow c, c \rightarrow i, c \rightarrow c\}.$$

There are no constraints on the types of nodes that can be incident with a node of type g .

Definition 3 is key. When we perform graph transformations, they will not always preserve the property of being a hybrid graph as defined here. This means that we may have to perform several transformations in series to ensure we end up with a hybrid graph that respects these type constraints. This is how the issues surrounding consistency of different system descriptions manifests in the graph-based approach.

Several other options are possible. For example, one could introduce a “null channel” consisting of an edge $e : i \rightarrow o$. An application of this type of edge is demonstrated in Figure 10 where it is used to avoid a channel traversing multiple components. However, in the absence of any compelling desire to avoid such a situation, we choose not to include the edge type as permissible.

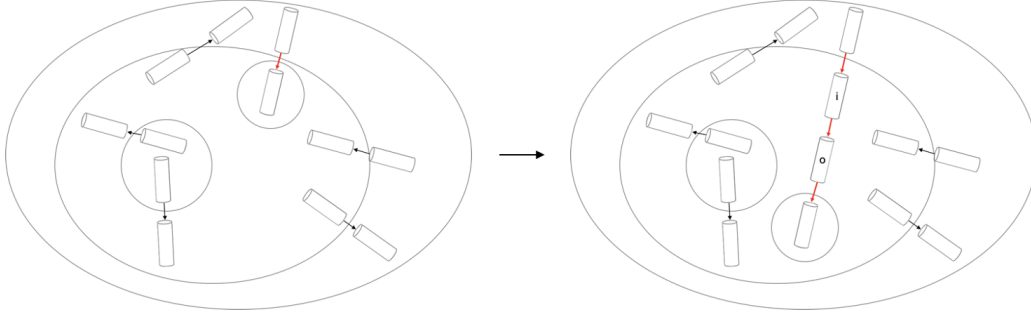


Figure 10: Example of a null channel to avoid traversing several components

Definition 4 (Element Graph) *There is a distinguished subset of node types $\mathcal{E} = \{s, b, r, a, i, o\}$ called elements. A hybrid graph $G = (N, E)$ is an element graph iff for every $n \in N$, $n : \tau$ with $\tau \in \mathcal{E}$.*

Element graphs have no nodes $n : g$ or $n : c$ that “summarize” some subgraph structure into single nodes. In this sense, they might represent “fully expanded” views of a system. This also explains the name of a hybrid graph: a graph whose nodes can be either elements or summaries of subgraphs of elements.

Definition 5 (Contraction, Visible/Occluded) *Let $G = (N, E)$ be a hybrid graph and suppose $N' \subseteq N$ is a subset of the nodes of G such that the induced subgraph defined by N' , $G' = (N', E')$, is connected. The subgraph contraction of G' is the process of replacing G' in G with a single element $n : g$ by contracting all the edges in E' . In this case, we say that G' is occluded in G . Otherwise, we say that G' is visible in G .*

Figure 11 shows an example where the endpoints of two channels have been occluded by two different subgraph contractions, thus losing the knowledge that the edges in the resulting graph were originally boundary crossings (they are now of type $e : g \rightarrow g$).

The restriction that the subgraph G' be connected may be a little too strong. For example, consider a virtualized system with a hypervisor and several VMs as depicted in Figure 12. If we want to group all the VMs together in one subgraph G' to be occluded, G'

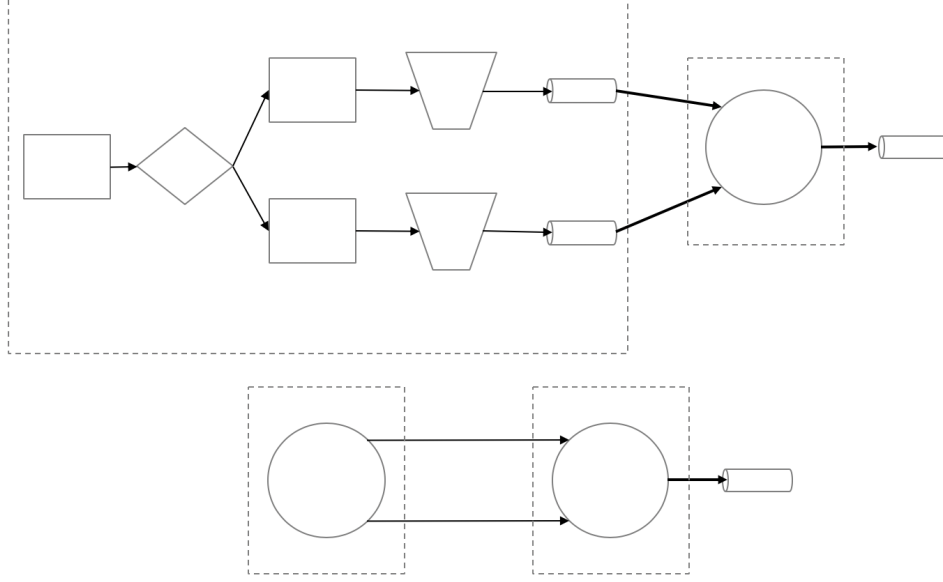


Figure 11: A contraction that “forgets” a boundary crossing

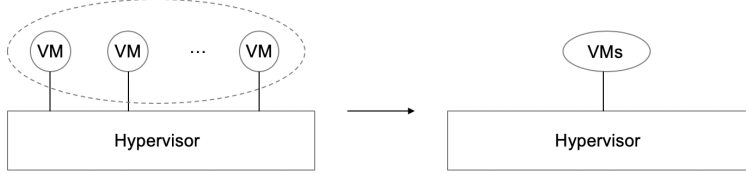


Figure 12: Example where we may want to occlude a non-connected subgraph

will not be connected and we will therefore not be able to occlude this subgraph according to the definition.

Figure 13 provides an example of a contraction that maintains both the component structure of G and the boundary crossings. It may be important to restrict contractions to include only those that “respect boundaries” in some way. We make this idea more precise below.

Separately, note that the hybrid graph resulting from occluding a subgraph will possibly be a multigraph. That is, a given pair of nodes may have more than one edge between them. (See Figure 14)

Definition 6 (Frontier) *Let $G = (N, E)$ be a hybrid graph, let $N' \subseteq N$ be a subset of the nodes of G , and let $G' = (N', E')$ be the induced subgraph defined by N' . (Note G' is not necessarily connected.) The frontier of G' (written $\text{frnt}(G')$) is the set of all $e = (n_1, n_2) \in E$ such that either*

1. $n_1 \in N'$ is a node of G' and $n_2 \in N \setminus N'$ is a node of G not in G' , or

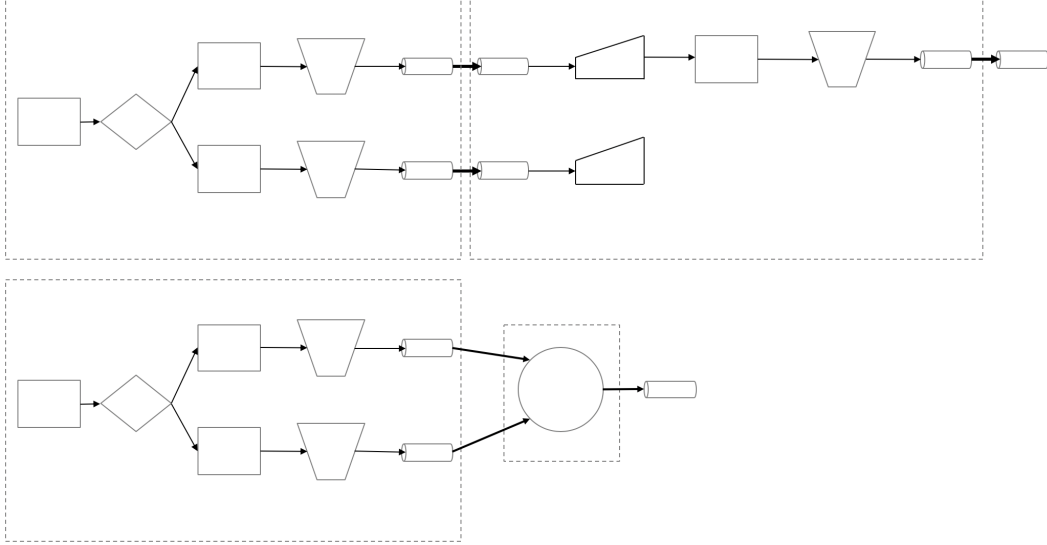


Figure 13: An acceptable contraction

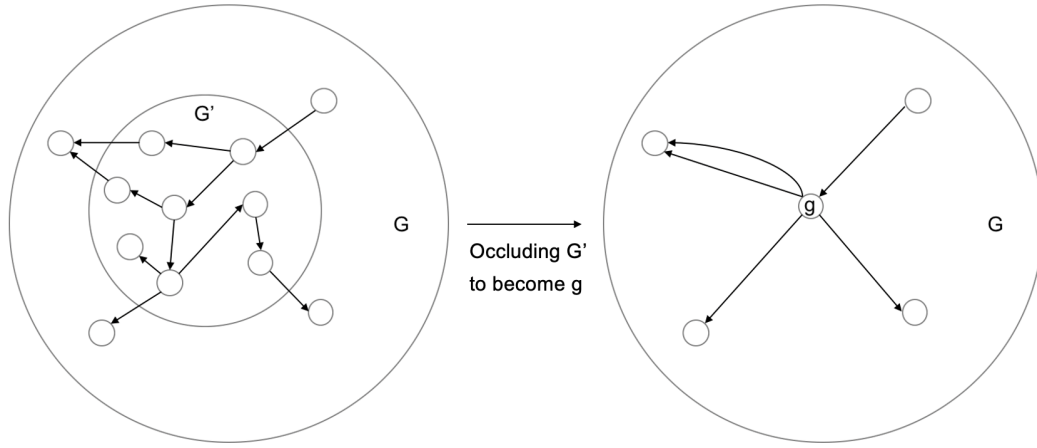


Figure 14: Occluding subgraph G' results in a multigraph

2. $n_1 \in N \setminus N'$ is a node of G not in G' and $n_2 \in N'$ is in G' .

Def. 6 recognizes that any collection of elements defines a frontier. These are the edges that have exactly one endpoint in the collection, and one endpoint outside the collection. We reserve the term *boundary* (Def. 8) to refer only to the frontier of a component (Def. 7), not to any arbitrary set of elements.

Definition 7 (Component, Subcomponent) Let $G = (N, E)$ be a hybrid graph, let $N' \subseteq N$ be a subset of the nodes of G , and let $C = (N', E')$ be the induced subgraph

defined by N' . We say that C is a component of G iff whenever $e \in \text{frnt}(C)$, then e is a channel edge (Def. 2). We say C_1 is a subcomponent of C_2 (written $C_1 \sqsubseteq C_2$) if C_1 is a subgraph of C_2 and both C_1 and C_2 are components of G . We write $N(C)$ for the node set of C and $E(C)$ for the edge set of C .

Definition 7 says that we can identify components as collections of elements whose frontier consists solely of channel edges. We recognize the possibility of a component having subcomponents. This allows for the possibility of a component having internal channels. This could be useful for describing a virtualized system. The physical machine defines a boundary to the outside world, but the hypervisor maintains boundaries between virtual machines on the same physical hardware. Such a possibility is depicted abstractly in Figure 15.

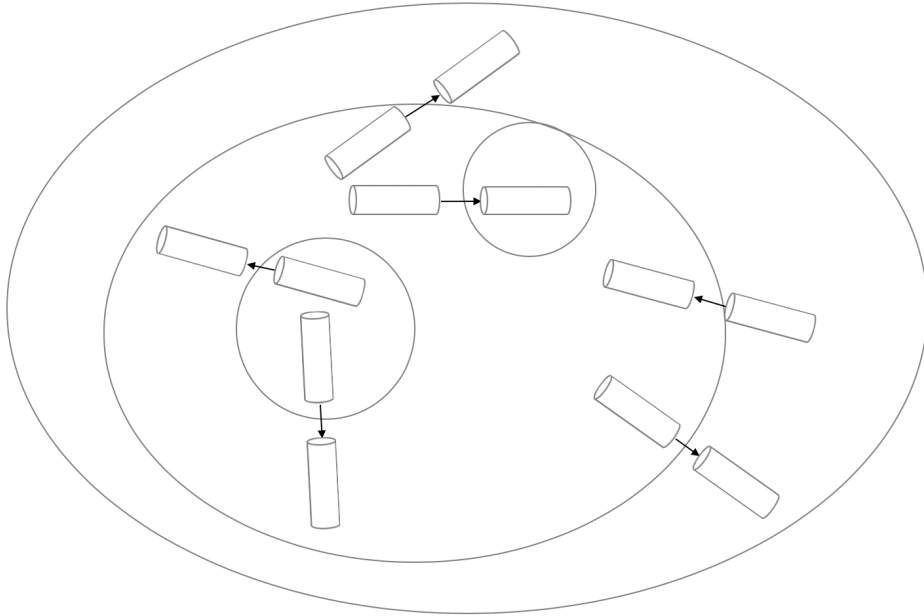


Figure 15: An abstract example of a virtualized system

Figure 16 introduces an interesting question about boundary crossings. The definitions currently allow an edge to cross several boundaries at once. If this is undesirable, the definitions will have to be adjusted.

Definition 8 (Boundary) *If C is a component of a hybrid graph G , then we call $\text{frnt}(C)$ a boundary. In this case we write $\text{bnd}(C)$ to denote the boundary of C . Each edge $e \in \text{bnd}(C)$ is called a boundary crossing.*

Note that $\text{bnd}(C)$ may be empty for certain choices of C . As examples, consider taking a single component $C = G$ consisting of the entire graph or taking C to be a subgraph that is disconnected from the rest of G .

Definition 9 (Atomic) *Let $G = (N, E)$ be a hybrid graph and let C be a component of G . We say that C is an atomic component of G if G has no proper subcomponents C' of C .*

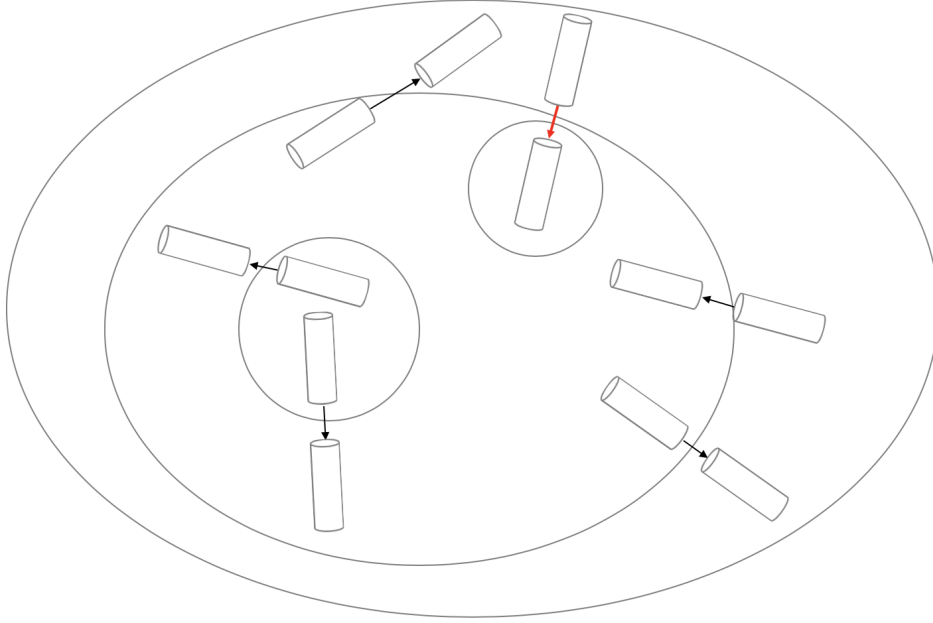


Figure 16: An edge crossing several boundaries at once

Defs. 7–9 refer only to individual components, not the full, modular structure of an MRS (Def. 11). Given a hybrid graph G , it may be possible to identify many components according to Def. 7. Not all such identifications are compatible. Nor are they all complete in the sense that every element belongs to at least one component. Figure 17, borrowed from Section 5, represents the internal details of a notional weather station. The elements in the green and blue enclosures represent components; the weather station oval could also be viewed as a component of the entire weather service example. The figure shows that not all component identifications are compatible, in that we would like to avoid intersecting components (other than proper subcomponents).

We revisit here the notion that occlusion should “respect a boundary” as mentioned following Def. 5. If the subgraph G' to be occluded is contained entirely within (or consisting of) an atomic component, then there are no problems. However, if G' spans multiple atomic components, we might have to expand the subgraph to contain all the atomic components which have elements in G' . Many of these choices for what kinds of transformations should be considered allowable depend on exactly how the transformations will be used. We believe this is a potentially fruitful area of research for modeling MRS.

Definition 10 (Component Structure) *Let $G = (N, E)$ be a hybrid graph, and let $\mathcal{C} = \{C_1, \dots, C_n\}$ be a collection of components of G . We say that $\mathcal{C}$ forms a component structure for G iff*

1. $\bigcup_{C_i \in \mathcal{C}} N(C_i) = N$: the union of the node sets of components is N ,
2. $\forall C_i, C_j \in \mathcal{C} . N(C_i) \cap N(C_j) \neq \emptyset \Rightarrow C_i \sqsubseteq C_j \vee C_j \sqsubseteq C_i$: either the intersection of node

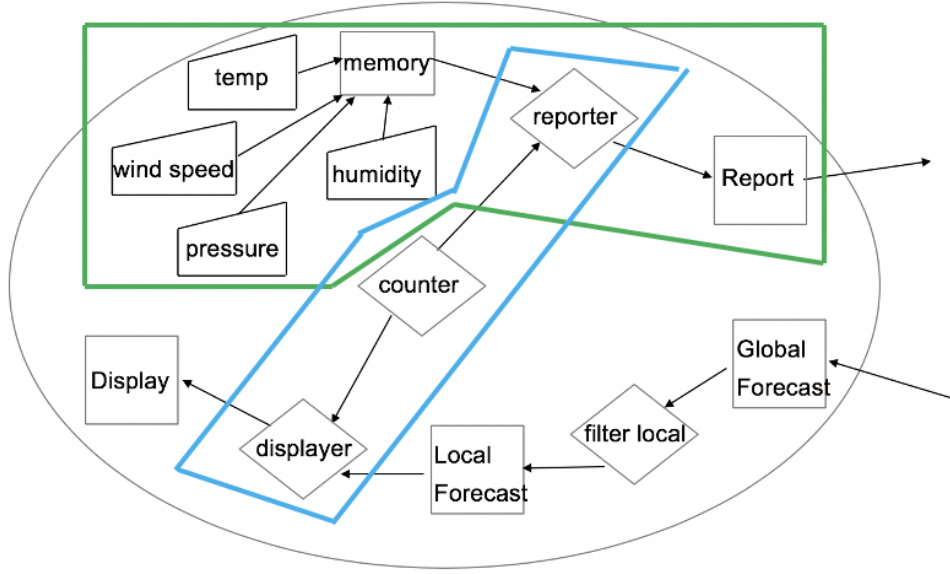


Figure 17: Some elements are not included in a component; the reporter reasoner is included in two components

sets of two components is empty or one component is a subcomponent of the other, and

3. $\forall e \in E$. if e is a channel edge, then $e \in \text{bnd}(C_i)$ for some i : each channel edge must be a boundary crossing of some component $C_i \in \mathcal{C}$.

Def. 10 addresses the issues identified in the paragraph previous to it. It stipulates three conditions. The first says that a component structure must assign each element to at least one component. This is not a strong restriction since the set of all elements can always be viewed as a component. The second condition states that the only overlap allowed is via subcomponents. The final condition says that every channel edge is part of the boundary of at least one identified component. This means that we do not have “extraneous” channels in our graphs. Figure 18 shows an example that violates this condition. The graph depicted there admits no component structure.

Definition 11 (MRS) *An MRS is a hybrid graph that admits a component structure.*

Definition 12 (Flat, Hierarchical) *Let $\mathcal{C}$ form a component structure for a hybrid graph G . We say that $\mathcal{C}$ is flat iff for every $C_i, C_j \in \mathcal{C}$, either $C_i = C_j$ or $C_i \cap C_j = \emptyset$: that is, every pair of components is either equal or disjoint. Otherwise, we say $\mathcal{C}$ is hierarchical if there exists some pair of components $C_i, C_j \in \mathcal{C}$ such that $C_i \sqsubseteq C_j$ with C_i a proper subcomponent of C_j .*

We conjecture at this point that if a hybrid graph admits a component structure, then it has a flat component structure consisting of atomic components.

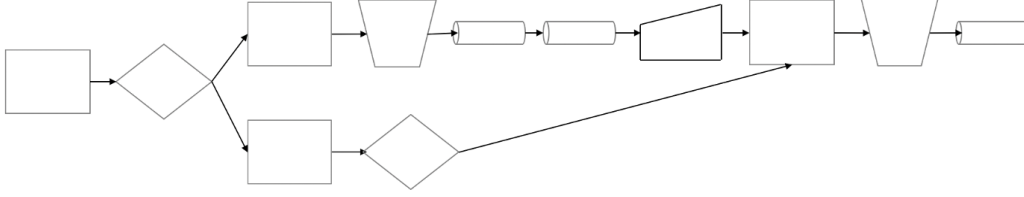


Figure 18: A graph that admits no component structure

Definition 13 (View) Let $G = (N, E)$ be a hybrid graph and let $\mathcal{C}$ form a component structure for G . Then we call $\mathcal{V} = (G, \mathcal{C})$ a view.

As we are probably only interested in graphs with a chosen component structure; we codify such entities in Def. 13 as a view. Note that a given hybrid graph G may admit several views because we may identify different component structures.

As we perform graph transformations on the underlying graph, we will frequently regroup the elements into different component structures as well. For example, a blackboard divide separates a single component into two. The definition of component (Def. 7) requires the introduction of a boundary crossing between the two blackboards that result. Performing this local transformation, may result in a graph that does not admit a component structure. For example, the graph in Figure 18 could have resulted from a blackboard divide in which the two blackboards on the top row were formerly a single blackboard. This blackboard divide does not result in a coherent view. In order to ensure the two blackboards end up in different components (in a new view) we must also transform the bottom branch, introducing a boundary crossing there.

These definitions allow us to distinguish between two potential purposes for identifying a collection of nodes as a unit. One purpose is to identify a component or a component structure in a fixed element graph G . In this sense, the definitions prohibit us from identifying arbitrary subsets of elements as useful collections. The constraints of what may be considered a component according to the existence of boundary crossings defines what is allowed. The other purpose is to propose a *new* component structure to guide a graph transformation. In this instance, it may be possible to collect arbitrary sets of nodes together. However, these collections may not correspond to boundaries in the current graph. We would then have to transform the graph G into some G' for which the new circles represent a component structure for G' .

One direction for future research in this area would be to develop graph transformation rules for exposing the hidden details of a sub-graph that has been previously occluded to allow us to move up and down between the extreme views where nothing is visible and everything is visible. For example, in order to invert a contraction, we must have a mechanism for remembering information about what was occluded.

Our approach in this section relies on graph-based representations of system components and their dependencies. This is a natural way to think about how various pieces of a system can influence each other. It also has the advantage that graphs do not impose a fixed granularity of representation. One may selectively increase the details in one part of a system while abstracting away the details elsewhere. We therefore envision a means for a human analyst to guide the transformation of one graph-based view into another based on

the need to highlight dependencies visible in the new view. This process creates a tree of graph-based system views in which children of one node of the tree are new system views generated in order to highlight relationships relevant to whatever system property is being explored at the time. These are the topics of the rest of this section.

4.2 Graph Transformations

As mentioned at the beginning of this section, our work using graph transformations was motivated by our desire to navigate views of a system (using the transformations listed below). The graph rewriting community had already solved many of the problems we were facing, including solutions for system engineering.

We briefly summarize here a set of six standard types of graph transformations (three sets of inverse pairs) that are useful in navigating among views. They are: expand and collapse, reveal and occlude, divide and merge. Each of the transformations performed in the examples in the following sections can be understood as an instance of one of these six types.

Expand: This type of transformation turns a single node into a larger subgraph. This amounts to “zooming in” on the details of the node in question. Since this type of transformation adds information not inherently contained in the current view, it must be informed by the details of the underlying system design.

Collapse: This type of transformation turns a subgraph into a single node and is the inverse of expand. This amounts to “zooming out” from the subgraph. This type of transformation only makes sense to be applied to a connected subgraph with a well-defined boundary between it and the rest of the system. Expanding a node and then collapsing the resulting subgraph returns the original view.

Reveal: This type of transformation embeds the current graph into a larger graph. This amounts to “enlarging the frame” around a given view. It differs from an expansion because it does not reveal any structure internal to any element already in view. Rather it reveals the structure of how the elements currently in view relate to surrounding elements. Since this type of transformation adds information not inherently contained in the current view, it must be informed by the details of the underlying system design.

Occlude: This type of transformation removes elements from view and is the inverse of reveal. It transforms a graph into one of its subgraphs. This amounts to “shrinking the frame” around a given view so that some portions are no longer visible. It differs from a collapsing because the elements that remain in view contain the same amount of information as they did in the original view. Revealing elements of a view and then occluding them returns the original view.

Divide: This type of transformation divides a node into two copies of itself with a “processing chain” between them. It differs from an expansion because the two resulting copies of the node represent the same element of the design. It is a logical separation, not a physical one. As such, although more information is added, this transformation need not be informed by the details of the design.

Merge: This type of transformation combines two identical elements and a processing chain between them into a single, unified element. This is inverse to a divide transformation. The goal of the graph-based approach to modeling MRSs is to continue to develop and improve upon the grammar representation of MRSs in the last section. In a graph-based MRS model, a tree is built where the nodes are the MRS components (or a collection of components that follow the relationship rules) and an edge from one node to the other if there is a relationship

between those components in the MRS being modeled.

Section 5 is dedicated to a demonstration of the methodology described in this section on a notional example system. Our aim is to provide the reader a better understanding of how this approach could be applied and how it can help expose the assumptions that often go overlooked when MRS capabilities are incorporated into a system. While we provide enough details to make the ideas concrete, the emphasis is on the methodology itself.

4.3 A Tree of Views

Applying a graph transformation to a view of a system results in a new system view. The relationships between views created by a graph transformation can be described visually as a tree, or more precisely a directed acyclic graph (DAG) where the nodes of the DAG are graph representations of a system view. Then a view obtained by applying a transformation to a first view produces a new node (graph representation of the resulting view) which is a child of the first view in the DAG. The resulting DAG after applying a sequence of transformations is called a *tree of views (TOV)*.

We used the technique of creating a TOV to understand how different parts of the system relate and to help track changes as a method of bookkeeping that provides potentially different information than assurance cases. TOVs help relate parts of a system to each other whereas assurance cases help ask security questions about those parts.

A TOV, an example of which is shown in Figure 19, allows an analyst to identify and track arguments and views throughout the process. A leaf node represents a view of the system where the analyst has made an assumption that the current view of the system is sufficient to explore a given system property.

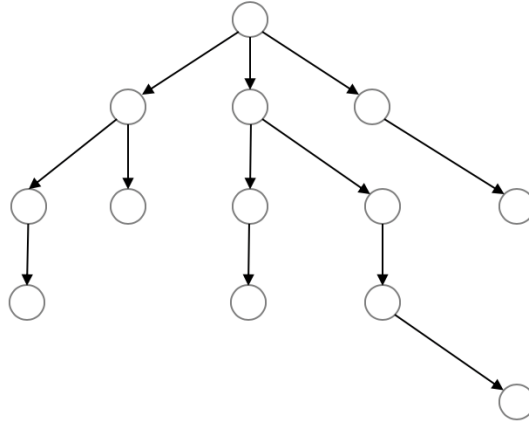


Figure 19: Notional tree of views

This tree of views can additionally serve as a means of keeping track of the various local arguments and how they fit together. Associated with each node in the tree would also be some local argument in favor of the system satisfying some property. That argument may rely on some assumptions that may appear as expanded arguments elsewhere in the tree.

This methodology is designed to be flexible and human-driven. It is unrealistic to expect

there to be a single, canonical set of system views that can help make all assumptions explicit. Rather, a human-guided approach to building the tree of views allows for fruitful discussion and debate about whether the views capture the assumptions and whether the arguments are convincing. Enabling such discussions based on a common understanding of the system structure is one of the key benefits of this methodology.

Nevertheless, since the relationship between arguments at different places on the tree of views should mirror the relationship between the graph-based views themselves, we anticipate a relatively small set of standard ways to transform one view into another. This will provide a framework for connecting local arguments into global ones. We have identified a small set of generic, graph-based transformations that we believe are sufficient to accommodate a wide-ranging set of view navigations. They are explained in more detail by example in Section 5.

4.4 Summary

The methodology of navigating through views (graphs of system representations) in this section supports system analysis by changing how the analyst views the system model while still representing the model in the same way. Also, we keep record of the various views of the model and the relationships between these views as a method of bookkeeping. We have not completely examined the base properties of a TOV as it is a difficult task that would make for an interesting topic of future research. However, we conjecture that, for a given system, there will always exist a “complete” view (i.e. an Element Graph per Def 4) that consists of only atomic elements and all other views could be derived from that. Thus we are not just creating many views of the system, but keeping an internal representation to ensure consistency.

We found that it would be useful to change representations of an MRS to focus on some local security property that provides evidence towards a global security argument for the system. The grammar-based model does not allow for this movement between representations of the same system since it produces a single representation for a fixed system design. The graph-based approach to modeling MRS improves upon the grammar representation of MRS in Section 3 by changing the way the base components are used to build a graph-based model and allowing an analyst to describe several representations of a system. This is the primary improvement of the graph-based model over the grammar-based model.

A graph-based representation of an MRS enables a human analyst to construct global security arguments for the MRS by combining local security arguments for subsets of the system. The analyst uses a graph-based representation to identify and track arguments for local security properties.

An analyst chooses the transformation from one graph-based view into another based on the need to highlight dependencies between components visible in the new view. The resulting tree of views highlights relationships relevant to the global system property currently being explored by the analyst. Each view of the tree is associated with a view of the system satisfying some local property. Combining the results of security analysis for each local views results in a global security argument for the entire system under study.

5 Graph-based Example: Weather Service System

Recall that our approach allows an analyst to selectively increase the details in one part of the system while abstracting away the details elsewhere. The tree of views created during this process documents the views chosen by the analyst to determine whether some high-level property of the system is true. Here we analyze the weather service system first described in Section 2.4.2 by decomposing this high-level property into more local properties that must hold in order for the high-level property to be true of the system as a whole. The high-level property we have chosen to address is:

Main Property: Each local forecast accurately reflects the forecast generated by the global model based on input data from the weather stations.

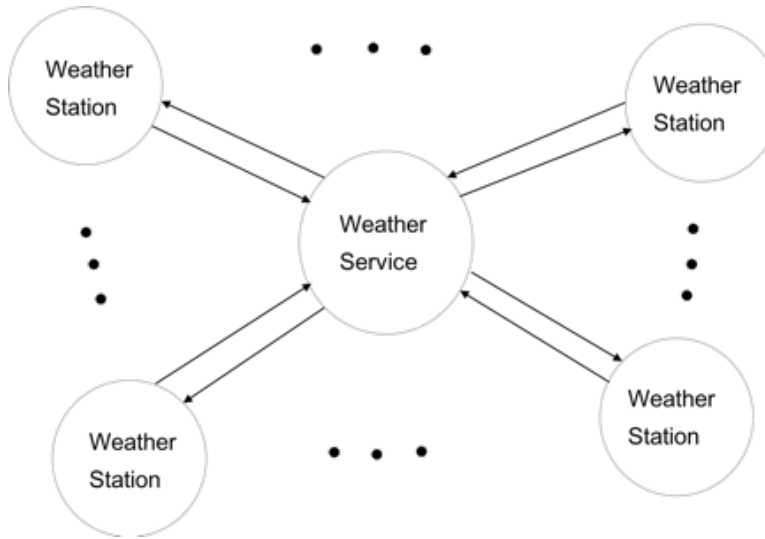


Figure 20: High-level Weather Service System View

How would we argue that this property holds? The system view of Figure 20 suggests an argument structure based on the visible relationships. Each weather station is potentially influenced by every other station as well as the weather service center. Thus, an argument that the main property holds could be based on

1. the correctness of the weather service
2. the correctness of the weather stations, and
3. the adequacy of communications.

We track our arguments in a tree of views. We start at the root node labeled “Main Property” which represents the main property of the system we are investigating. Arguing that the main property is true if the three properties above are true induces three children nodes in the tree corresponding to the three properties (with node labels corresponding to the list numbers) as is depicted in Figure 21. The graphs represented by the nodes will be presented in the following sub-sections.

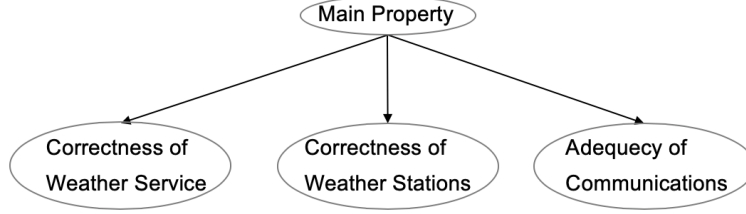


Figure 21: A tree of views corresponding to the three sub-properties of the main property

In our analysis, we will accept the risk in assuming that the communication channels are reliable and focus our analysis on the correctness of the weather service and the correctness of the weather stations. The flexibility in deciding which assumptions need to be discharged by further analysis and which ones represent acceptable risk is a key feature of our approach. It allows an analysis to bracket out certain considerations while also forcing the analyst to confront and document the assumption.

The decomposition of the main (global) property into three (local) properties of parts of the system suggests additional system views. That is, by decomposing the system property, we are naturally led to adjust our view of the system from that shown in Figure 20 in various ways. For example, when verifying the correctness of the weather service, we will need to expose more detail of the internal structure of the weather service, but we may be free to abstract away, or even ignore the weather stations. This process of adjusting system views to accommodate the argument structure can be repeated indefinitely until the assumptions are either obviously satisfied, or else the risk of them being false is acceptably low.

As we expose some details of the system, we will get to a view where we need to recall part of a “visual vocabulary” from Figure 9 in Section 4.1 to represent the generic type of an object or function. Square symbols represent “blackboards” which are elements that store data. Diamond symbols represent “reasoners” which are elements that process data. Trapezoid symbols represent “sensors” which are elements that collect information from the system environment and store it in a blackboard. “Actuators” are another type of element used in monitoring and response systems, but they are not used in our analysis below.

5.1 Correctness of the Weather Service

To argue for the correctness of the weather service, it is natural to consider a view that “zooms in” on the details of the weather service and ignores details of the individual weather stations. Such a view is depicted in Figure 22. In the tree of views generated by the argument that the main property holds if the three local properties hold, the view in Figure 22 corresponds to the node labeled “Correctness of Weather Service” in the tree.

The basic flow of data through the weather service is that data from all weather stations is received by the weather service and is stored in a Report blackboard. Then a Recorder reasoner processes the data by storing it in a Weather History blackboard that contains the weather history of all station locations. A model trainer reasoner then processes the weather history data to produce a Weather Model which is used by a Forecaster reasoner to create the Global Forecast for all locations. The Global Forecast is then broadcast to all stations. With this view of the weather service, an argument for its correctness depends on the correctness

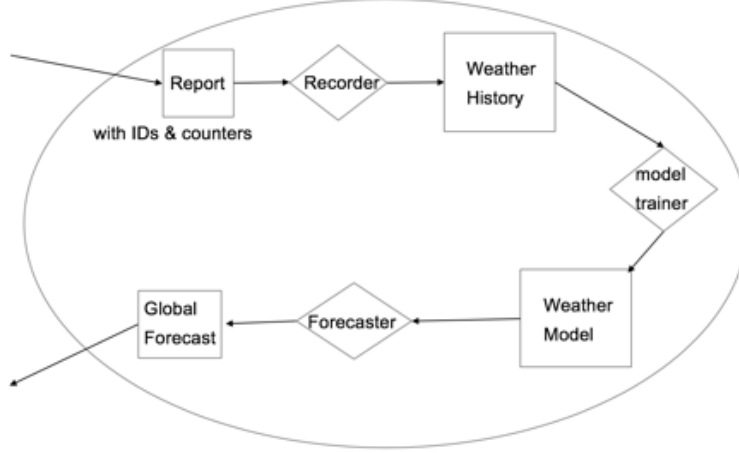


Figure 22: High-Level Weather Service View

of its component functions. Although this view identifies seven distinct components, it is perhaps more natural to consider three fundamental functions of the weather service center: data ingestion, weather forecasting, and reporting. An argument based on the correctness of these three functions suggests three new system views to explore. In contrast to the decomposition of the global system, these new views overlap in potentially interesting ways. For example, the Weather History blackboard is part of the ingestion function since it is the end of the ingestion pipeline. It is, of course, also fundamental to the weather prediction function.

5.1.1 Ingestion

The first weather service function we analyze is the ingestion of data from the weather stations. The view of the internal structure of the weather service needed to analyze the ingestion function is demonstrated on the left side (a) of Figure 23 as the part of the structure above the red line. We only need the Report blackboard, the Recorder reasoner and the Weather History blackboard, so we adjust our view in the right side (b) of Figure 23 to only keep those elements visible and occlude the rest.

In the tree of views shown in Figure 24, Figure 23(b) is labeled as "Ingestion" and is a child of the "Correctness of Weather Service" shown in Figure 22.

A full correctness argument would have to account for aspects such as de-multiplexing and de-duplication. If reports from the weather stations arrive with information that identifies the source and a counter for each source, then the Recorder should be responsible for inserting the reports into the Weather History in a manner that ensures all reports get recorded and there are no duplicates. This might involve exposing more detail about the Recorder reasoner to better understand how it achieves these properties.

One could imagine zooming into code-level views of the Recorder and arguments for correctness based on formal analysis of the code. For the purposes of this exposition, however, we note that such a detailed analysis is not always necessary. In order to conclude that the ingestion function is correct, we may be willing to rely on the fact that it has

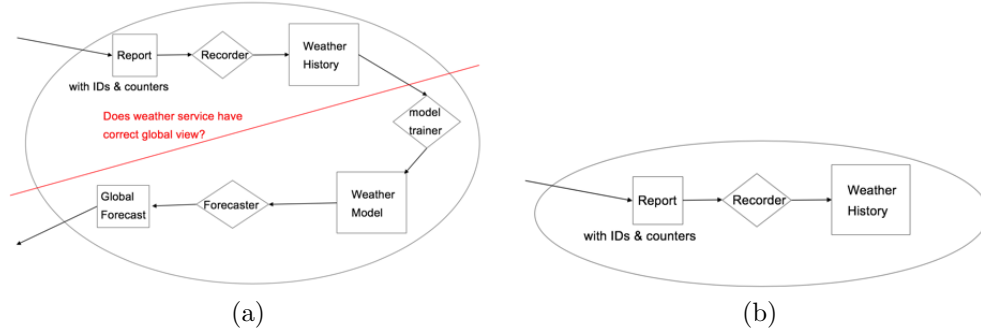


Figure 23: Internal Weather Service details: (a) before and (b) after changing views

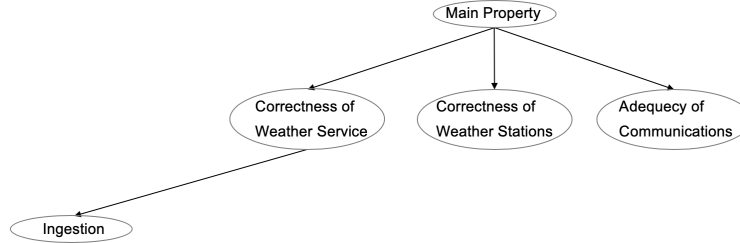


Figure 24: Ingestion is a child of the Weather Service

undergone typical testing and review. Such an argument is much weaker than a more detailed formal analysis. Its likelihood of being falsified is much greater, but it still identifies reasons for believing the property in question.

Consider now, however, the Weather History blackboard. One implicit property of the history is that once an entry is made by the Recorder, that entry remains unchanged over time. That is, the integrity of the data over time is an important property. A further change in view (from Figure 25(a) to Figure 25(b)) would help us determine if there are any dependencies that have remained invisible until now. For example, the Weather History could be located in a shared data store. This would uncover a dependency between the Weather History and all other entities that can access the data store. Such hidden dependencies are frequently the source of security problems that creep into designs. As a standalone function, the data ingestion may be perfectly sound, but once it is integrated into a larger system, the new dependencies may undermine previously established properties.

Exploring the correctness in the view on the right may expose further detail within the rest of the structure. Given a particular view of the system, an analyst can evaluate whether a desired system property holds. For example, the analyst may have a code review process for the Recorder reasoner to ensure that it is not subject to buffer overflows. The analyst may also have some technical mitigation such as using SELinux as a Mandatory Access Control mechanism to protect a file system in order to reduce the risk of assuming the property holds to an acceptable level.

Depending on the design, the correctness could depend on the correctness of all other

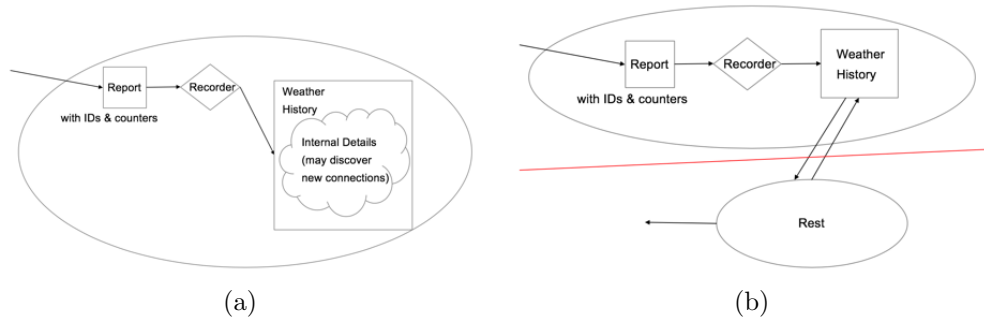


Figure 25: View when Weather History has connections to the rest of the internal structure

components connected to the Weather History, or it might only depend on a localized filter that is part of the (currently hidden) details of the design of the Weather History itself.

5.1.2 Forecasting

The next function of the weather service that we analyze is the weather forecasting for each of the weather stations. The view of the internal structure of the weather service relevant to the forecasting function is demonstrated in Figure 9 as the part of the structure to the right of the red line. Note that Weather History is in the intersection of the internal structure needed for ingesting and forecasting.

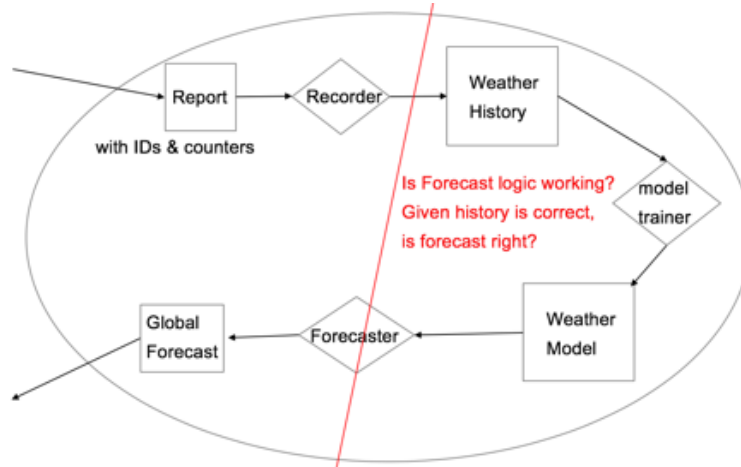


Figure 26: Internal Weather Service Details of Forecasting Function

The red line drawn in Figure 26 goes through the Forecaster reasoner, so we expose further details of the Forecaster to determine whether it is functioning correctly and where exactly the red line crosses. When exposing the details of the Forecaster reasoner, we discover that it is really two reasoners, a Forecast Generator and Forecast Displayer, connected with a Forecast Output blackboard in between as shown on the left of Figure 27. We still

have the issue that the red line goes through an element, namely the Forecast Output blackboard. We can now perform a “blackboard divide” on Forecast Output as seen on the right side of Figure 27. The new symbols represent elements of an MRS that are not important to this discussion, so we do not describe them here.

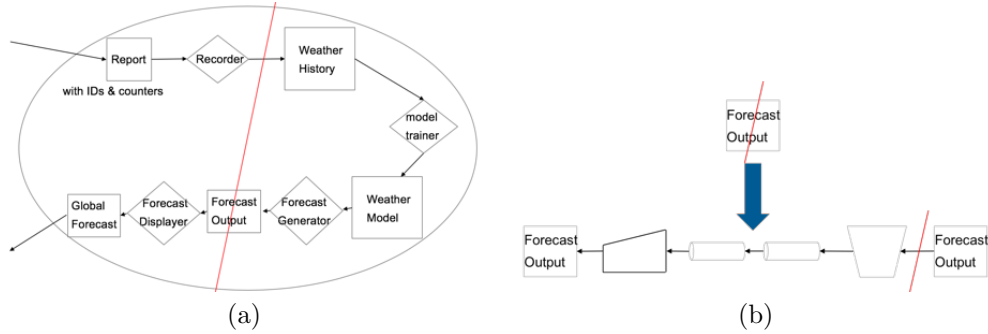


Figure 27: (a) Splitting the Forecaster Reasoner; (b) Generic Blackboard Divide

After performing a blackboard divide on Forecast Output, all the new additional structure shown in the right side of Figure 27 except for the rightmost blackboard is not needed to verify the correctness of the forecasting function, so it is removed from view. The resulting view is shown in Figure 28. In the tree of views shown in Figure 29, this figure would be a child of Figure 22 and it would be a sibling of Figure 23(b) and Figure 25(b).

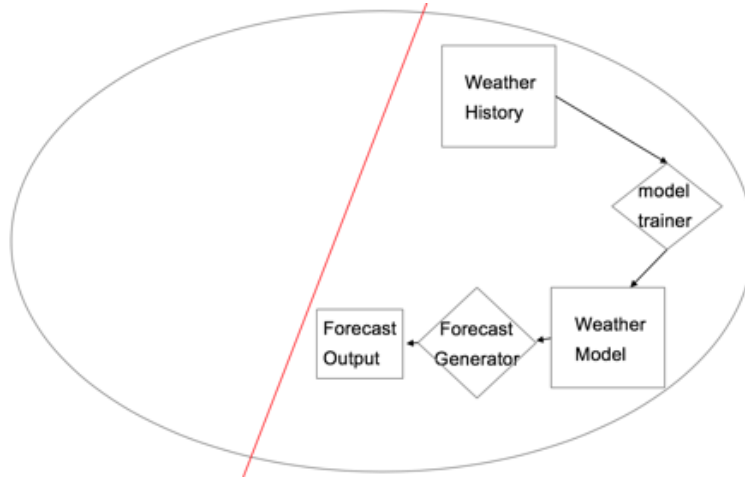


Figure 28: View when verifying the correctness of the Forecasting Function

Just as above, this view would lead to an argument in favor of the correctness of the forecasting function by examining properties of the model trainer for example. The models could be validated based on research, but there may be a question of the accuracy of the models for which formal methods may not be applicable. To evaluate the design of the model trainer, the analyst would need to examine the machine learning algorithm used to

create the Weather Model. However, doing so would not be sufficient to completely evaluate the model trainer, the analyst also needs to examine the correctness of the weather stations and the ingest function. If either of those is found lacking, the analyst would need to return to an analysis of the machine learning algorithm to verify that its correctness is robust to being trained on potentially corrupted data. Of course, the degree to which the data from the weather stations deviates from ideal must be within the bounds that the machine learning algorithm can tolerate. This is a good example of how the global argument is not simply a conjunction of the local arguments. They must work together in non-trivial ways.

The tree of views could expand further to support this argumentation, or it could stop at this level if the risk is deemed acceptable. For the purposes of exposition, it is sufficient to stop here. However, the tree of views shows where to revisit if the risk turns out to be unacceptable.

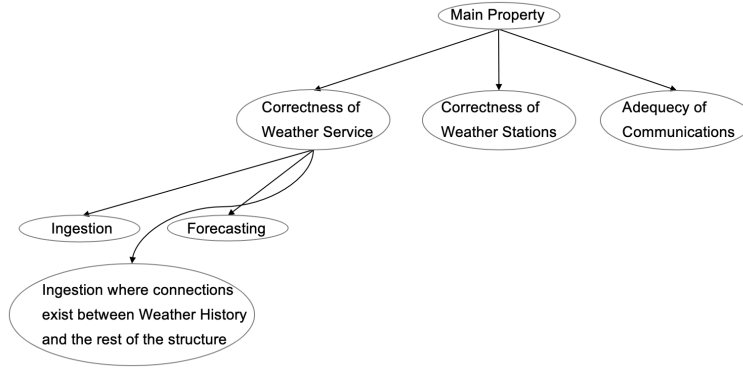


Figure 29: Forecasting is a child of the Weather Service

The analysis in this section highlights a main goal of our approach: Helping system designers and integrators identify who might be responsible for ensuring local properties are satisfied. Our structured view navigation provides a common model in which to identify dependencies and assumptions. It is not prescriptive of what those dependencies should be, but rather enables discussion/debate on common ground.

5.1.3 Reporting

The last function of the weather service that we consider is the reporting of the weather forecasts to each of the weather stations. Note our assumption is that the weather service broadcasts all weather forecasts to all weather stations and each station filters out the data needed to display the weather forecast for its location. The view of the internal structure of the weather service needed to analyze the reporting function is demonstrated in Figure 30 as the part of the structure below and to the left of the red line.

Note the red line for reporting also cuts through the Forecaster reasoner as the red line for forecasting does. We handle this situation the same way we did for forecasting, by splitting the Forecaster into Forecast Generator and Forecast Displayer with a Forecast Output blackboard in the middle. We then perform a blackboard divide on Forecast Output and keep only the final blackboard in the divide below the red line as shown in Figure 31(a). The right side, Figure 31(b), shows an alternative view that might result if verifying the

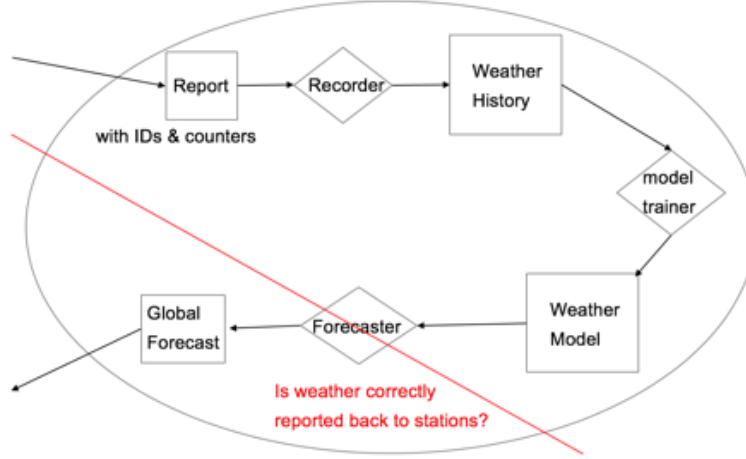


Figure 30: Internal Weather Service Details Needed to Analyze Reporting

correctness of the three elements on the left indicates some dependency on the rest of the graph that is invisible in previous views. In this case, we may need to expose some details above the red line to complete our analysis of the reporting function. For example, the correctness of the Forecast Output certainly depends on the correctness of the Forecast Generator which in turn depends on the Weather Model. Some details of one or both of these elements may need to be exposed in order to gain confidence in the correctness of the Reporting Function. The rest of the analysis would proceed as above.

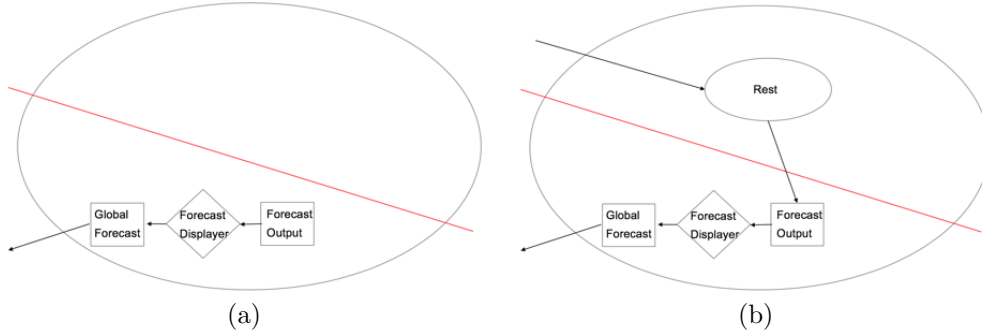


Figure 31: Views of Reporting Function (a) without and (b) with dependencies

In the tree of views shown in Figure 32, the views in Figure 31(a) and Figure 31(b) are children of the weather service.

5.2 Correctness of the Weather Stations

Recall our argument that the main property holds if the three local properties hold. In the previous section, we analyzed the first local property, the correctness of the weather service.

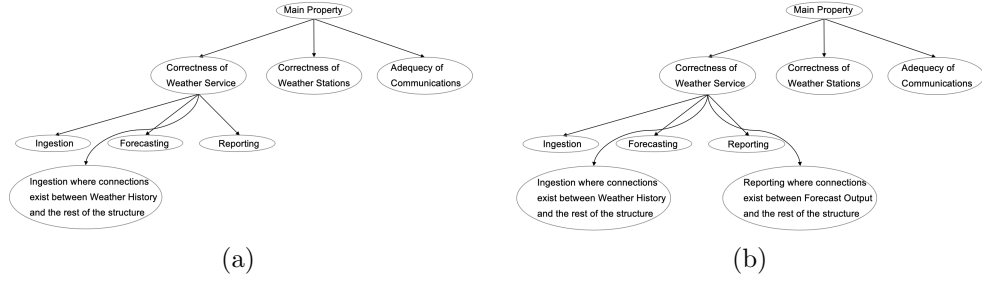


Figure 32: The Reporting Function (a) without dependencies and (b) with dependencies are children of Weather Service

We now move on to the second local property, the correctness of the weather stations.

To argue for the correctness of the weather service system, we consider a view that expands the details of a weather station and ignores details of the weather service. A view of the internal details of a weather station is depicted in Figure 33.

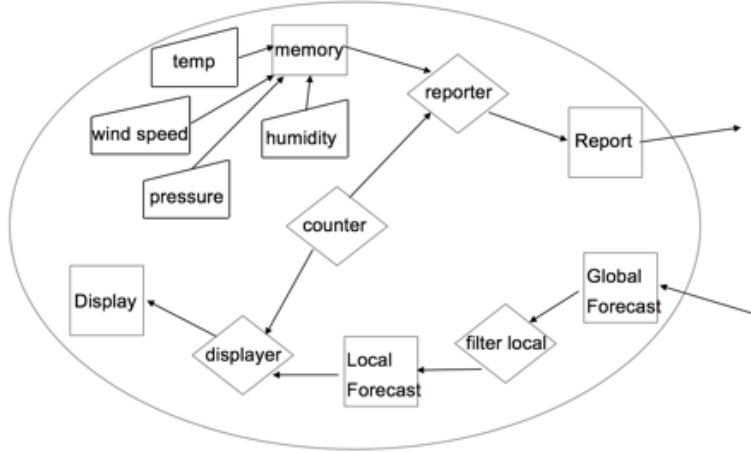


Figure 33: Internal Details of a Weather Station

The basic flow of information in a station is that the four sensors—temperature, wind speed, pressure, and humidity—collect and store environmental data in memory which is then processed by a reporter reasoner, to include a counter so the weather service has record of when the sensor data was collected. The reporter reasoner stores its output in a Report blackboard where it is then transmitted to the weather service. Once the weather service completes a forecast, it broadcasts to all stations which receive the forecasts into a Global Forecast blackboard. The station must then filter out the forecast for its location and store it in a Local Forecast blackboard. A displayer reasoner takes the Local Forecast data, combines it with a counter, and displays the forecast on a Display blackboard.

As with the weather service, this view suggests an argument for correctness that depends on the correctness of its various functions. The weather station functions identified in this

analysis are updating weather data from the sensors and reporting this information to the weather service, ingesting the weather forecast information from the weather service and extracting the forecast for the location of the weather station, and displaying the weather forecast. As the reader should now have a grasp of the methodology, we simply demonstrate how the tree of views would be modified as we navigate from the view in Figure 16 to views relevant to the component functions. We elide further discussion of how arguments get attached to these new views and guide the creation of new views to support those arguments.

5.2.1 Weather Station Functions

In this section, we provide an outline of the analysis of the three weather station functions: Data Update and Reporting, Forecast Ingestion and Extraction, and Displaying the Local Forecast. The analysis is not complete but is provided to give an idea of what needs to be addressed in the weather stations to argue that the main property holds.

5.2.2 Data Update and Reporting

The view of the internal structure of a weather station needed to analyze the Data Update and Reporting function is demonstrated in Figure 34(a) as the part of the structure above the red line. It is not anticipated that there will be any dependency on the part of the view below the line, so we can focus our attention on verifying the correctness of the elements in the part of the structure above the red line.

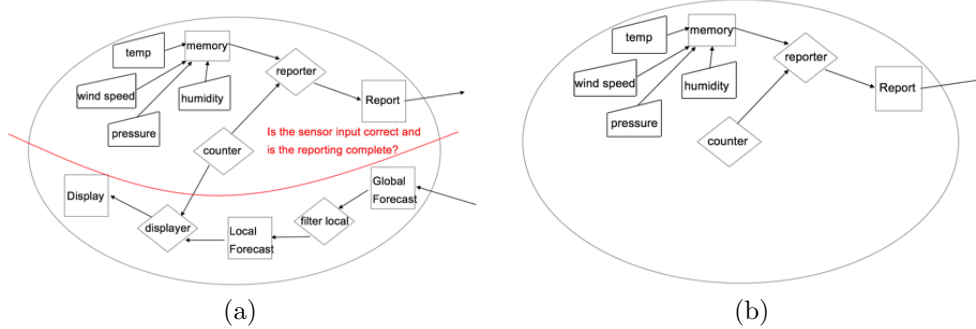


Figure 34: View for Data Update and Reporting. (a) In context and (b) in isolation

The view of a weather station provided in Figure 34(b) is a child of Figure 33 in the tree of views as shown in Figure 35.

5.2.3 Forecast Ingestion and Extraction

The next weather station function we will analyze is the ingestion of forecast data received from the weather service and the extraction of the local forecast. The view of the internal structure of a weather station needed to analyze the forecast ingestion and extraction function is demonstrated in Figure 36 as the part of the structure below the red line. Again, there is no anticipated dependency of the elements below the red line with the rest of the elements in the view, so we can simply occlude the rest of the view.

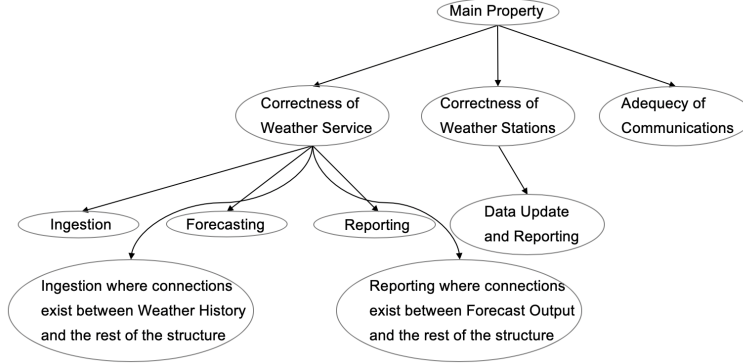


Figure 35: The Data Update and Reporting function is a child of Weather Station

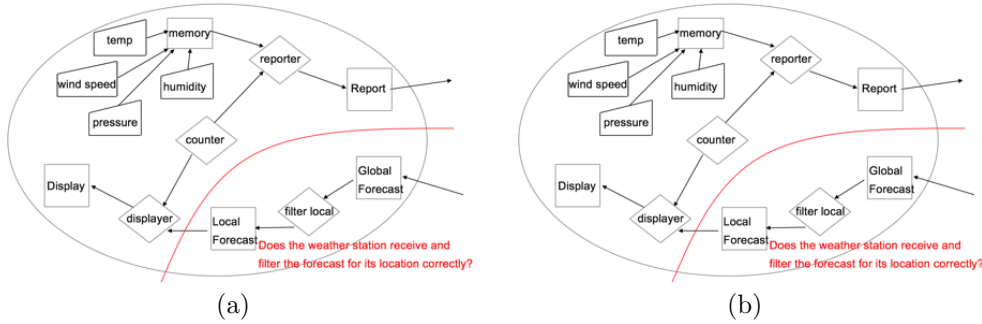


Figure 36: View for Forecast Ingestion and Extraction in (a) context and (b) isolation

The view of a weather station provided in Figure 36(b) is a child of Figure 33 and a sibling of Figure 34(b) in the tree of views as shown in Figure 37.

5.2.4 Local Forecast Display

The last weather station function we consider is displaying the local forecast. The view of the internal structure of the weather service needed to analyze the ingestion function is demonstrated by the part of the structure below the red line. There is the possibility that the structure below the red line will have some dependency on the Global Forecast and filter local reasoner in which case we would need to show the rest of the ingestion and extraction function as a single component that may need to be examined further, exposing details.

The view of a weather station provided in Figure 38(b) is child of Figure 33 and a sibling of both Figure 34(b) and Figure 36(b) in the tree of views as shown in Figure 39. This is the final tree created by the analysis process described in this document. The "Adequacy of Communications" node has no children because we chose to accept the risk in assuming that the communication channels are reliable, thus making it a leaf node in the tree of views.

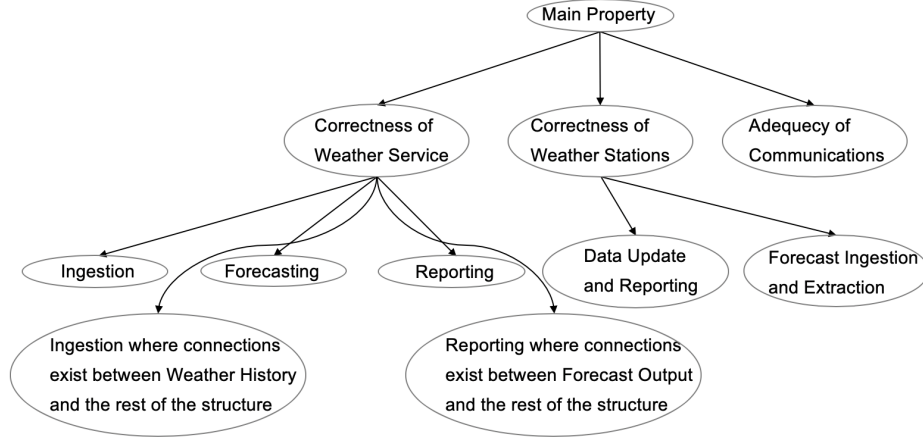


Figure 37: The Forecast Ingestion and Extraction function is a child of Weather Station

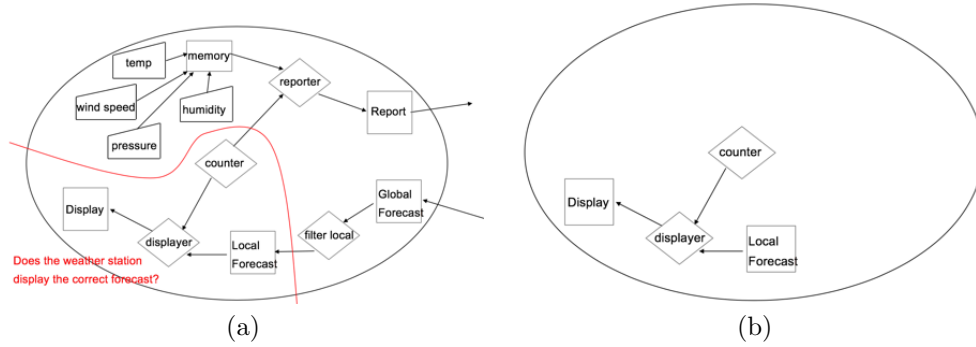


Figure 38: View Relevant to Forecast Display. (a) In context and (b) in isolation

5.3 A Global Assurance Argument

The primary value of the methodology exhibited in this section is to help system designers create global security arguments for their designs. A key benefit is to push system designers and integrators to confront and document the assumptions required to argue for the security of the system as a whole. We envision the tree of views that is built as a result of the methodology as an important artifact resulting from an application of the methodology. In addition to the graph-based system views at each node in the tree, every node will also contain a local security argument and some assumptions required for that argument to hold. Taken as a whole then, the tree of views represents a global assurance argument for the entire system, or it highlights deficiencies that should be remedied as part of an ongoing design process.

It is important to recognize that this global argument is not meant to be an ironclad proof of security. Rather, it can serve as a focal point for discussion and debate about the merits of the given design relative to alternatives. Our methodology does not offer an

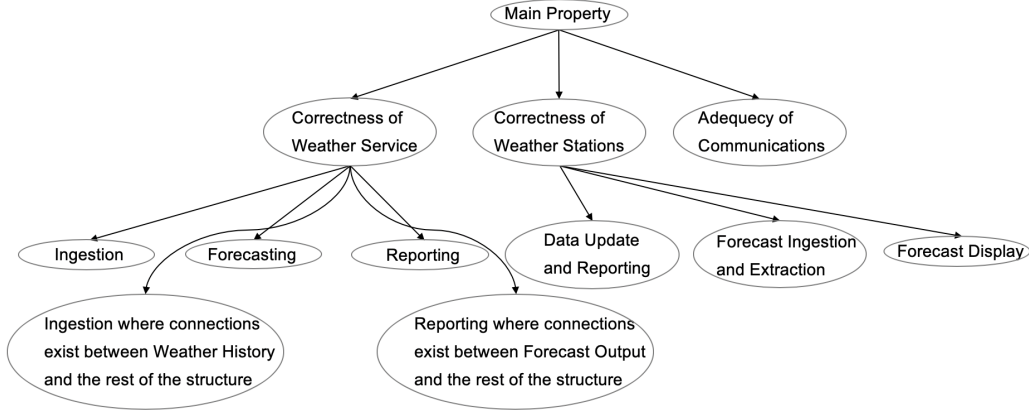


Figure 39: The Forecast Display function is a child of weather station in the tree of views

exhaustive list of properties for an analyst to check. It is the responsibility of the analyst to ensure that a complete set of critical properties is examined. We also do not prescribe a method for validating properties at any given point. At each stage in the process, the analyst has some notion of how the local design choices that were made contribute to the security of the overall system. Some reasons may be informal while others can be supported by rigorous experimentation or formal analysis. It is reasonable to expect differences of opinion in how a system view should be decomposed to support an argument, or in what constitutes sufficient evidence of security. Highlighting this diversity of opinion is an asset of our approach because it increases the likelihood that hidden assumptions will be brought into view.

Despite this diversity of opinion, however, we believe that the process is useful in identifying general reasoning principles that can be reused in diverse parts of the system. For example, in Section 2.1.1 we highlighted the potential use of SELinux as a technical mitigation to ensure proper access control for a shared data store. This mitigation would be useful in other areas where a shared data store must ensure controlled access regardless of the content of the data. Any argumentation about this mitigation attached to the tree of views can be reused as-is at another node where the issue of controlled access is the focal point of the system view. In this way, we can build a “tool box” of common sets of arguments, easing the analysis.

Thus, although the tree of views indicates a clean, modular decomposition, cross-references among the attached arguments may reveal a more realistic complex structure. Indeed, it would be quite revealing if these cross-references had cyclic dependencies. This would be a strong indication of hidden dependencies in the design that could undermine the overall security of the system.

Aside from questions of how the results of this process would contribute to system designs, there are also issues of making this approach accessible and usable to people. The mechanics of navigating system views, building the tree of views, and tracking assumptions and arguments is non-trivial. We expect there to be a relatively small set of common graph transformations that will be sufficient to achieve a wide variety of useful navigations. We have already seen above the strategies of expanding details of components, occluding irrel-

evant components, and logically dividing blackboards and reasoners. Identifying reusable strategies for composing such transformations would help guide analysts throughout this process. Furthermore, tool support for specifying and executing these transformation rules will likely be required if this approach is to gain traction.

5.4 Summary

We have indicated the tree of views as a useful artifact to serve as a focal point for discussions, but its concrete structure is still currently under-specified. Defining a fixed and common structure will be important for ensuring consistency in the composition of the local security arguments. We believe this structure can be an object of study in its own right. Understanding the semantic connection among the graph-based view at a given node, the attached argument, and the local assumptions is critical in properly evaluating whether or not the global security argument holds together. Although there is still work to be done, we believe the approach presented in this document can play an important role in taming the complexity that arises from systems designed with capabilities to monitor, respond, and adapt to their environments.

6 Building an Assurance Case

The graph and grammar work of the previous sections showed us (1) how one might describe individual parts of a system model; (2) how those parts relate to each other; (3) how one might keep track of those system components in a human friendly way. This previous work however did not show us how we could apply statements of assurance to individual components of the system or a system as a whole and measure or evaluate those statements in some deterministic way. We needed a method that would allow us to apply some set of constraints on a system and enable us to track the validity of those statements throughout the system. Specifically, we needed a method that helped us track and structure assurance statements and help us evaluate the validity of these statements using empirical evidence.

6.1 Background

Upon reviewing existing methods to perform such analysis we discovered the field of assurance cases and believed that a reduced version of assurance cases could be applied to MRS for our purposes. Ideally, such a method could be coupled with our graph-based tree of views work to evaluate and measure arguments on components of a system and track how invalidated arguments affect other parts of the system. This was ultimately the goal of this work but we were unable to achieve this at the time of this writing. This technique coupling is presented as future work in Section 7.

We explored assurance cases as a means to keep track of the relationships between the assurance arguments (and analysis questions we ask) and the evidence (system model and views of that model represented by a tree of views). Assurance cases in this context continue the story of bookkeeping described in Section 2.2. The large community around assurance cases from the safety critical engineering community and the growing interest by the security community inspired us to use this technique for such bookkeeping needs. In this context, we found that assurance cases provided a structure upon which we were able to organize our thoughts concerning security and functionality claims of an MRS, connect those thoughts to evidence, and act as a cohesive artifact that ties the work presented in this report together.

Somewhat unintentionally, we found our use of assurance cases was motivated as much by organizational and bookkeeping needs as well as a need for a formal method.

An assurance case can be defined as *A reasoned and compelling argument, supported by a body of evidence, that a system, service or organization will operate as intended for a defined application in a defined environment.*[14]. In other words, an assurance case is not a mathematical proof of soundness or correctness or some other proof of security. Instead, an assurance case is a Baconian argument that attests a system, service, or organization is *probably* sound and correct for some goal under some set of assumptions.

One might now consider the case for forgoing an assurance case and building a mathematical proof of security for MRS. While this could prove fruitful for small systems, we have found that such a proof may be impractical for sufficiently complex systems such as MRS which have wide implementation diversity and loose foundational definitions. Beyond the considering of just security, an assurance case can be used for reasoning about the safety of a system or the presence of unintended behavior such as incident response visualization inconsistencies in MRS [2]. In this section, we will show that (1) generating and analyzing an assurance case of an MRS can reveal the possibility of unexpected behavior of the MRS; (2) how one might address that unexpected behavior; (3) even a brief effort spent on generating an assurance case can lead to the discovery of surprising design improvements.

As defined above, an assurance case is a methodology for exploring and making a reasonable attestation of soundness and correctness for a system. An assurance case does not generate a model for a system nor does an assurance case provide a standard language for communicating a resulting argument. To solve this problem, one might use graph transformations to generate a system model or a system specification document. Furthermore, we will use language inspired from the Goal Structuring Notation [14] to document our assurance case example. The visual elements used by the GSN are presented in Figure 40 and used in the next section.

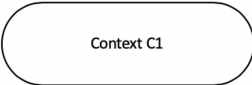
	A security claim.		Evidence to support a goal.
	Documentation the reasoning step as to how a goal is supported by sub-goals.		
	The context in which the claim or reasoning step should be interpreted.		Statement that must be true for the associated goal or strategy to be true.
	A goal that needs further strategies and/or sub-goals to support the truth of the goal.		Explanation as to why an associated goal or strategy is acceptable.

Figure 40: Visual descriptions of the Goal Structuring Notation

Importantly, an assurance case does not provide a security model or system requirement upon which an assurance case is build. For this, we have elected to use system specifications and reasonable expectations about what a system should provide. We document this model and expectations in the example in the next section.

6.2 Assurance Cases In Practice

In this section, we will explore an assurance case with a focus on understanding and analyzing the security of OSSEC and Wazuh which is briefly described in Section 2.4.3. The goal of this section is to demonstrate how one might build an assurance case for a system like an MRS using the goal structuring notation. An assurance case can enable one to make relatively bold claims about the security of an MRS and reasonably support those claims with a structured argument in the form of an example assurance case.

6.2.1 Practical Consideration

In our exploration of assurance cases, we found several practical concerns that potentially present a barrier for using assurance cases. We address the following concerns here.

1. The value of a reasoned argument vs. a mathematical proof
2. The complexity of an assurance case with complex systems such as MRS
3. The general usefulness of an assurance case
4. Choosing the correct security model

Concern 1: Assurance cases do not provide a mathematical proof of the claims or goals that one may set out to validate. Instead, assurance cases provide a structure and supporting evidence to reason about a claim or goal of a system. While we will show that assurance cases are useful even when they are limited, it may be difficult to determine when an assurance case has a threshold of evidence or structure to sufficiently validate a claim or goal. It is up to the author of such an assurance case to determine when the collection of support is sufficient. In any case, halting the construction of an assurance case will strongly rely on the relationship of the case goals with a predetermined security model. Nevertheless, we found assurance cases to be useful in analyzing an MRS where proofs of security may be impractical or otherwise infeasible.

Concern 2: By using a graph-based model of an MRS, one can determine that, given a set of a dozen goals, the number of edges and nodes in the model graph can quickly expand into the hundreds; more complex systems can result in thousands of leaves. In such a system, it may be impractical to build an exhaustive assurance case in a small research or design environment. However, even when complexity prevents one from building an exhaustive assurance case, building a brief assurance case can provide value by revealing vulnerabilities and unexpected behavior in the underlying system design.

Concern 3: While a brief assurance case may not sufficiently assure security for certain strict definitions, it may nevertheless be useful in providing a practical level of software assurance, an automated testing framework, or enable a designer or developer to discover unexpected system design behaviors. Importantly, assurance cases can be relatively straight forward and can also be concise as we will show in a later section.

Concern 4: In any security analysis, a security model should be presented that outlines the abilities of the systems and the capabilities of an adversary. In practice, this has proved to be a particularly difficult challenge. This is in part because MRS touch many systems, contain many protocols, and there is no standard or consistent set of components upon which an MRS security model applies. In other words, there doesn't appear to be an accepted security model standard for MRS that applies to application security as well as the protocol security of an MRS. However, there has been some effort to do so [1] [32] [7]. For this work, we have taken the approach of using a basic set of security assumptions for a security model and modified that model as we reasoned about how to apply assurance cases to MRS.

In particular, we initially we made the following security assumptions.

1. an adversary will not have root access to a client or server manager
2. an adversary may read all traffic sent over the wire
3. standard cryptographic assumptions

We have used three approaches for generating assurance cases that may help reduce the complexity of generating assurance cases by narrowing the focus of the assurance support to a subset of the system. The first approach is a breadth-first search or top-down approach in which one attempts to exhaustively provide evidence to an assurance goal for only the first few layers of the system model. In the context of a graph-based tree of views, this would correspond to limiting the depth of edges from the root. Such a model may appear similar to the system design specification and may be useful for analyzing high level design flaws. The second approach is a depth-first search or bottom-up approach where one attempts to focus on a single goal and only analyze edges from the system model that are strictly necessary or can not satisfied by making an assumption. In a tree of views, this would correspond to the minimum number of paths needed to support the root. The third approach is a hybrid search where one prioritizes assurance goals that most closely affect the security of the system and include the least evidence needed to support the assurance goal without considering dependencies. After this initial assurance case has been generated, one then iterates through each goal and associated support and addresses dependencies and assumptions. We have found that this approach allows us to better map a model description of a system to an existing implementation in source code. In practice, we have used system documentation and source code to help us guide the generation of our assurance cases. As we will later discuss, our results have shown a hybrid search approach to be the most practical as it helps identify and address assumptions shared by components of the MRS. Furthermore, it provides a framework on how to verify assurance arguments through supporting evidence (source code and documentation).

In any case, the structure of an assurance argument allows for it to be expanded upon or otherwise improved when new security concerns, product requirements, or system changes affect the underlying system model. We will use assurance cases to explore security and functionality goals that correspond to real implementations of an MRS and provide a semblance of mathematical soundness that will assert that the underlying system is reasonably secure under some security model.

6.2.2 Approaches and Challenges

Our initial intuition was to perform a top-down / breadth-first approach. This approach tracks closely with the standard approaches used for assurance case modeling in the Goal

Structuring Notation (GSN). We believed that by using this standard approach, our research would be better set up to help designers and analysts understand the security properties about a modeled system. Furthermore, we believed that this approach would help better convey to analysts the reasons why a system achieves or does not achieve a target set of security properties. More specifically, we explored assurance cases and the GSN as a way to construct an assurance argument tree to organize how we might model MRS design security.

Figure 41 is a high-level example of this approach when applied to OSSEC scoped to an assurance case for OSSEC processes that access the key files and processes that maintain confidentiality of the key files.

OSSEC operates securely.

1. Show confidentiality and integrity of files used by OSSEC.
 - a) The OSSEC processes that access the key files maintain confidentiality of the key files.
 - (i) Using `manage_agents` to provision an agent key maintains confidentiality of the key files.
 - (ii) Using `agent-auth` to provision an agent key maintains confidentiality of the key files.
 - (iii) The server and agent using their keys to communicate maintains confidentiality of the key files.
 - b) The OSSEC process that access files other than the key files maintain confidentiality of those files.
 - c) OSSEC maintains integrity of the files.
2. Show the OSSEC processes operate as expected.

Figure 41: Using the Goal Structure Notation to show OSSEC operates securely

The main goal is “OSSEC operates securely”. This is very broad, so we use the GSN concept of strategies to break the goal into strategies (that lead to sub-goals). The first strategy is to show confidentiality and integrity of files used by OSSEC; the second is to show the OSSEC processes operate as expected. As we are focusing on OSSEC processes that access the key files and processes that maintain confidentiality of the key files, we explore only the first strategy. Given the assumptions and context for this strategy, we define three sub-goals of the main goal such that the main goal is achieved if the three sub-goals are achieved. The sub-goals are (1) the OSSEC processes that access the key files maintain confidentiality of the key files, (2) the OSSEC processes that access files other than the key files maintain confidentiality of those files, and (3) OSSEC maintains integrity of the files. Again, since we are focusing on OSSEC processes that access the key files and processes that maintain confidentiality of the key files, we only explore sub-goal (1). To show OSSEC achieves this first sub-goal, we define three sub-goals of sub-goal (1) for the same reasons as before and proceed in the same way. The GSN example in Figure 42 demonstrates a partial assurance argument tree for secure operations of OSSEC. Recall the definitions of the visual elements in the GSN are shown in Figure 40.

We also looked at the tree of views constructed to identify high-level security goals of a system and then arrive at more concrete claims or goals that can be verified in some specific

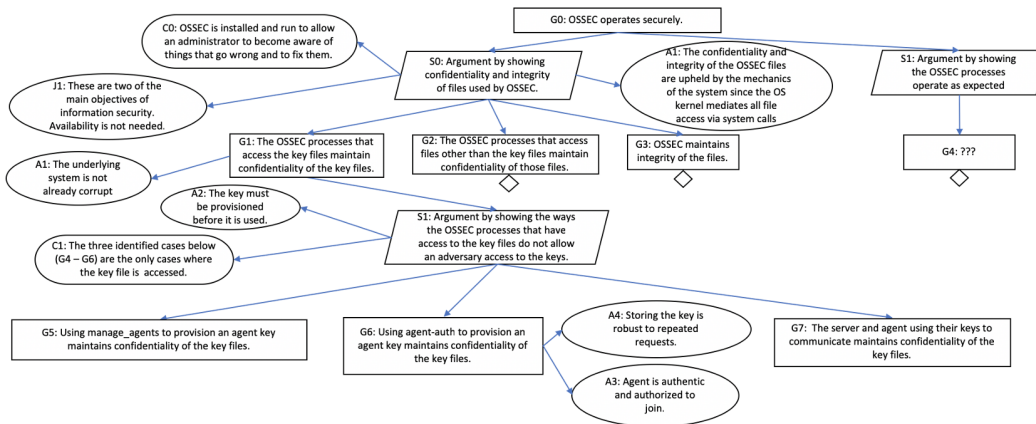


Figure 42: A partial assurance argument tree for OSSEC security by examining the security of the `client.keys` file

manner through a process of decomposition and refinement. Figure 43 shows a tree of views for the security of OSSEC.

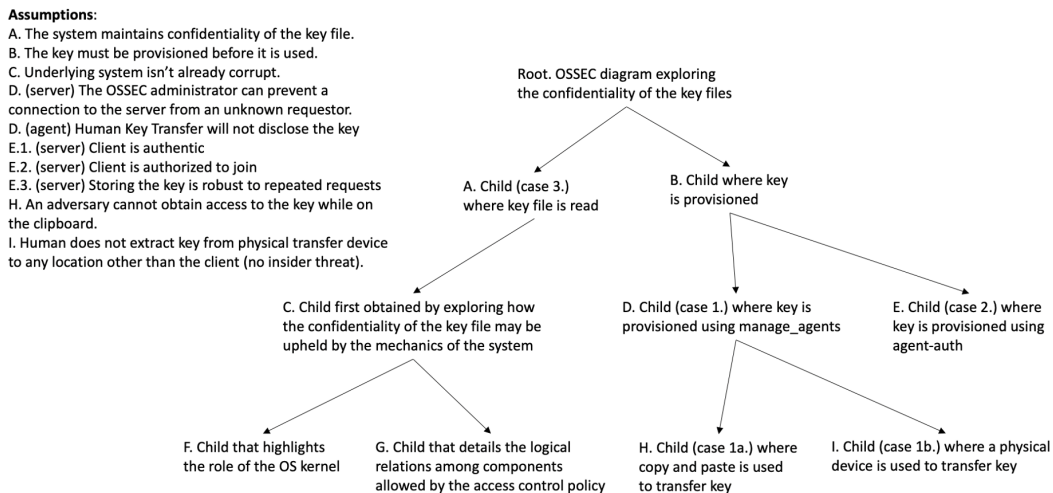


Figure 43: Tree of views for OSSEC by examining the security of the `client.keys` file

Unfortunately, these approaches were deemed impractical to study MRS in part because they tended to grow rapidly with the number of paths we traversed and in part because generating an assurance case for a MRS without a clear definition of what a MRS should be provided no practical foundation upon which to verify assurance arguments.

To address these shortcomings, we decided to adopt an interactive approach. With respect to a MRS such as OSSEC or Wazuh, documentation and source code are easily obtained. Using this information we (1) use the design specification to generate an initial

set of assurance claims and goals; (2) generate the highest level of assurance case that allows us to test these assurance arguments against an implementation; (3) verify these claims against source code and/or an implementation. We then iterate between claim generation and verification until we are satisfied with the scope of the assurance case. Using this interactive approach between theory and implementation has allowed us to rapidly create and test the validity of assurance cases for Wazuh.

Within the context of graph transformations, we focused on some particular leaves of a (notional) assurance argument. There is value in identifying higher-level places in the trees where we can apply the interactive approach. To this end, we have been focusing on the functionality of Wazuh rather than the security.

6.3 Summary

As seen in Section 4, graph transformations provide a mechanism with which we can generate a model of a complex system, specifically an MRS. In this section we showed how we were able to use assurance cases to incorporate a reasoning about the correctness and soundness of an MRS. With the goal of understanding and reasoning about the security and guaranteed functionality of an MRS, assurance cases provide a proof-like structure that enables us to assert that an MRS is indeed secure, provides functionality that we expect, and limits functionality that we don't expect. In short, assurance cases allowed us to document, track, and validate claims that we may want to impose on an MRS. In future work, we would like to explore how one might combine graph rewriting and assurance cases through a tree of views. Such a blend of techniques might be able to help us reason about how a violation of an assurance argument within one view of an MRS affects components in other views. Furthermore, it would provide a bookkeeping strategy that might be partially automatable.

7 Open Problems

This section is intended for those readers interested in possible directions for continuing the work described in this paper. We start by identifying some general areas of our research that are incomplete and could use further development.

We did not drive this work to completion, however it may be useful to readers interested in demonstrating that a given MRS meets its security goals. We have demonstrated some capabilities to find software vulnerabilities using assurance cases, but we have no general tool support for the entire process of building an assurance case; without tool support, we have no way to determine whether our methodology is more or less efficient than existing methodologies and/or tools.

A couple questions about our work, that could be asked about any methodology for building assurance cases, are how far/deep a designer or analyst needs to go in the process of building an assurance case before it is considered complete and when and how does a designer or analyst know that all the desired security properties have been covered.

The rest of this section addresses specific tasks that require further development and should be usable as a self-contained document to enable others to take the work described in this paper and build upon it.

7.1 Extending our Work

Omitted from this paper is proving that our methodologies and techniques are valuable as compared to other methodologies, techniques, and tools via experimentation. In part, this will require determining what challenges there are when using these methodologies in practice. For example, the ideas in each section, grammar-based model (Section 3), graph transformations (Section 4.2), tree of views (Section 4.3), and assurance cases (Section 6) need to be fleshed out further.

When representing a system with a grammar-based model, there is no definition of a formal consistency condition nor has it been proven that the (yet-to-be-discovered) principles of MRS abstraction and refinement yield consistent results. Second, there is still work to be done in graph transformations to define rules for exposing the hidden details of a sub-graph that has been occluded (e.g., inverting a contraction) so that an analyst may move both up and down in a tree of views. Third, although we have determined many properties of a tree of views and identified some basic transformations that allow an analyst to move throughout a tree of views, determining the base properties of a tree of views (e.g., is there a system view for every system that consists of only atomic elements?) is a difficult problem that remains open. Furthermore, the analyst-driven process of changing system views in a tree of graphs and building assurance cases needs to be refined so that the process can be standardized. Last, there is still significant work to be done in determining next steps for building assurance cases which will likely necessitate some level of tool support.

7.2 Arguments and Modeling

As described in Section 6, assurance cases can be used as a bookkeeping method for linking analyst-driven assurance arguments and the supporting empirical evidence derived from the model. However, we were not able to couple assurance cases with our graph-based tree of views work to evaluate and measure arguments on components of a system to track how invalidated arguments affect other parts of the system. We would also like to explore how to combine graph transformations and assurance cases through trees of views. Such a blend of techniques might be able to help us reason about how a violation of an assurance argument within one view of an MRS affects components in other views. Furthermore, it would provide a bookkeeping strategy that might be able to partially automate.

7.3 Tooling Support

During the course of this work, we explored many different tools to see if any could support the analysis and documentation that we were looking to perform. Originally we used simple diagramming tools like Microsoft Visio and Graphviz to create the kinds of diagrams we needed to document our thoughts. A simple whiteboard was also used as we tried to solidify the concepts for how we could build and manipulate the various models we had proposed. However, as we pushed further on the models and tried to document larger systems in higher detail, we found these tools to be quite lacking.

Appendix A goes into detail about each of the tools we explored and includes examples of how to use these tools to partially perform the techniques described in previous chapters. However, in the end, we found that none of the tools met all of the functionality we had envisioned. The exploration of tools provided two main benefits to this work. First, it demonstrated that the individual methods we present have merit and similar techniques show up in pieces across other tooling. Second, the exploration of some of the tools actively

informed our ideas and helped shape their direction. In the end, we believe that useful new tooling could be created by combining features and functionality from three main tools described in the appendix.

8 Conclusion

This section concludes the paper with some final take-aways from the material presented herein.¹ The main contributions of this work include a novel methodology for constructing MRS models that provides the ability for a designer or analyst to navigate between different views of those models in order to assess the security posture of an MRS. Additionally, this paper presented a process for using assurance cases to keep track of both the sequence of views of and security arguments made in support of an MRS. In general, the methodology introduced in this paper can aid designers and analysts in analyzing an MRS to determine whether it meets the intended security goals.

We present concluding remarks specific to each of the concepts we studied, pursued, and improved throughout this paper in the following three sections.

8.1 Conclusions on Grammar-based MRS Models

In Section 3, we presented our work in using a context-free grammar, derived from five base MRS components, to model an MRS that facilitates representing the MRS at different scales. Grammar-based MRS representations are useful in general to describe general paths of information through an MRS. However, they do not provide the desired capability to navigate between views of an MRS in order to explore security properties of the MRS. To be able to focus in, pan across, and occlude parts of the MRS in a particular view and to make use of existing tools for creating MRS models, we moved to graph-based MRS models.

8.2 Conclusions on Graph-based MRS Models

The path to graph-based MRS tree of views models is long and winds through a lexicon (presented in Section 4.1) we developed to standardize modeling notation, graph transformations (also called graph-rewriting; presented in Section 4.2) to change views of an MRS to focus on a local security property that provides evidence towards a global security argument, and the development of the novel method for constructing a tree of views (presented Section 4.3) to better understand how different parts of an MRS relate and to keep track of the sequence of graph transformations applied by an analyst.

A graph-based MRS model enables an analyst to construct global security arguments for the MRS by combining local security arguments derived from particular views of the MRS. We therefore envision a means for an analyst to guide the transformation of one view into another based on the need to highlight new dependencies visible in the second view. A key benefit of using our methodology is that an analyst-driven approach to building a tree of views allows for discussion between designers and analysts about whether the views capture necessary assumptions and whether the arguments in favor of an MRS satisfying a particular security property are convincing. Section 5 is dedicated to a demonstration of the methodology described in this section on a notional example system.

¹The conclusions and open problems do not imply government solicitation.

8.3 Conclusions on Security Assurance Cases for MRS

In Section 6 we demonstrated using assurance cases to incorporate a reasoning about the correctness and soundness of an MRS and to document, track, and validate claims that we may want to impose on an MRS. With the goal of understanding and reasoning about the security and guaranteed functionality of an MRS, assurance cases provide a proof-like structure that enabled us to assert that an MRS is indeed secure and provides functionality that we expect and limits functionality that we do not expect.

References

- [1] Rebecca Bace and Peter Mell. Nist special publication on intrusion detection systems. Technical report, BOOZ-ALLEN AND HAMILTON INC MCLEAN VA, 2001.
- [2] R. Calinescu, D. Weyns, S. Gerasimou, M. U. Iftikhar, I. Habli, and T. Kelly. Engineering trustworthy self-adaptive software with dynamic assurance cases. *IEEE Transactions on Software Engineering*, 44(11):1039–1069, 2018.
- [3] Alexander Christoph. Graph rewrite systems for software design transformations. In Mehmet Aksit, Mira Mezini, and Rainer Unland, editors, *Objects, Components, Architectures, Services, and Applications for a Networked World*, pages 76–86, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.
- [4] Paul Clements, Felix Bachmann, Len Bass, David Garlan, James Ivers, Reed Little, Robert Nord, and Judith Stafford. *Documenting Software Architectures: Views and Beyond*. Addison-Wesley, 2003.
- [5] Darren Cofer, Isaac Amundson, Ramachandra Sattigeri, Arjun Passi, Christopher Boggs, Eric Smith, Limei Gilham, Taejoon Byun, and Sanjai Rayadurgam. Run-time assurance for learning-enabled systems. In *NASA Formal Methods: 12th International Symposium, NFM 2020, Moffett Field, CA, USA, May 11-15, 2020, Proceedings*. Springer.
- [6] Julien Delange, Peter Feiler, and Ernst Neil. Incremental life cycle assurance of safety-critical systems. 2016. hal-01289468f.
- [7] Kelley L Dempsey, L A Johnson, Matthew A Scholl, Kevin M Stine, Alicia Clay Jones, Angela Orebaugh, Nirali S Chawla, and Ronald Johnston. Information security continuous monitoring (iscm) for federal information systems and organizations. Technical report, 2011.
- [8] D. Dori and E.F. Crawley. *Object-Process Methodology: A Holistic Systems Paradigm*. Springer Berlin Heidelberg, 2013.
- [9] Dov Dori. Object-process methodology applied to modeling credit card transactions. *Journal of Database Management (JDM)*, 12(1):4–14, 2001.
- [10] Elastic.co. Elastic products. Available at <https://www.elastic.co/products/>. Accessed on 11-May-2020.

- [11] Peter Feiler, Julien Delange, and Lutz Wrage. A requirement specification language for aadl. Technical Report CMU/SEI-2016-TR-008, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2016.
- [12] Peter H Feiler and David P Gluch. *Model-based engineering with AADL: an introduction to the SAE architecture analysis & design language*. Addison-Wesley, 2012.
- [13] Andrew Gacek, John Backes, Darren Cofer, Konrad Slind, and Mike Whalen. Resolute: An assurance case language for architecture models. *Ada Lett.*, 34(3):19–28, October 2014.
- [14] Assurance Case Working Group et al. Goal structuring notation community standard (version 2), 2018.
- [15] John Hudak and Peter Feiler. Developing aadl models for control systems: A practitioner’s guide. Technical Report CMU/SEI-2007-TR-014, Software Engineering Institute, Carnegie Mellon University, Pittsburgh, PA, 2007.
- [16] Gartner Inc. Security orchestration, automation and response (soar). Available at <https://www.gartner.com/en/information-technology/glossary/security-orchestration-automation-response-soar>. Accessed on 08-May-2020.
- [17] ISO Central Secretary. Automation systems and integration – Object-Process Methodology. Standard ISO/PAS 19450:2015, International Organization for Standardization, Geneva, CH, 2015.
- [18] Jeffrey O Kephart and David M Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.
- [19] Anneke G Kleppe, Jos Warmer, Jos B Warmer, and Wim Bast. *MDA explained: the model driven architecture: practice and promise*. Addison-Wesley Professional, 2003.
- [20] Philippe B Kruchten. The 4+ 1 view model of architecture. *IEEE software*, 12(6):42–50, 1995.
- [21] A. Martini, H. Ehrig, and D. Nunes. *Graph Grammars: An Introduction to the Double Pushout Approach*. Bericht (Technische Universität Berlin. Fachbereich 20, Informatik). Leiter der Fachbibliothek Informatik, Sekretariat FR 5-4, 1996.
- [22] Enterprise Systems Modeling Laboratory Technion Israel Institute of Technology. Model-based system engineering: Opcloud: Opm. <https://www.opcloud.tech/>. Accessed: 2020-02-03.
- [23] OSSEC. OSSEC documentation. Available at <https://www.ossec.net/docs/>. Accessed on 11-May-2020.
- [24] Patrick Edwin Moran. Diagram of the OODA Loop, 2008. Url: https://en.wikipedia.org/wiki/OODA_loop#/media/File:OODA.Boyd.svg. File: OODA.Boyd.svg. License: CC BY 3.0.
- [25] Mor Peleg and Dov Dori. Extending the object-process methodology to handle real-time systems. *JOOP*, 11(8):53--58, 1999.

- [26] A. Piovesan and E. Griffor. Reasoning about safety and security: The logic of assurance. In Edward Griffor, editor, *Handbook of System Safety and Security*, pages 113 -- 129. Syngress, Boston, 2017.
- [27] Iris Reinhartz-Berger and Dov Dori. Object-process methodology (opm) vs. uml-a code generation perspective. In *CAiSE Workshops (1)*, pages 275--286, 2004.
- [28] Arend Rensink. Groove. Available at <https://groove.ewi.utwente.nl/>. Accessed on 11-May-2020.
- [29] Arend Rensink. The groove simulator: A tool for state space generation. In John L. Pfaltz, Manfred Nagl, and Boris Böhlen, editors, *Applications of Graph Transformations with Industrial Relevance*, pages 479--485, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.
- [30] Raymond Richards. Cyber assured systems engineering (case). Available at <https://www.darpa.mil/program/cyber-assured-systems-engineering>. Accessed on 10-Jun-2020.
- [31] Raymond Richards. High-assurance cyber military systems (hacms). Available at <https://www.darpa.mil/program/high-assurance-cyber-military-systems>. Accessed on 10-Jun-2020.
- [32] Karen Scarfone and Peter Mell. Guide to intrusion detection and prevention systems (idps). Technical report, National Institute of Standards and Technology, 2012.
- [33] Arnon Sturm, Dov Dori, and Onn Shehory. Single-model method for specifying multi-agent systems. In *Proceedings of the second international joint conference on Autonomous agents and multiagent systems (AAMAS '03)*, 2003.
- [34] Carnegie Mellon University. Open source aadl tool environment. Available at <https://osate.org/>. Accessed on 10-Jun-2020.
- [35] Wazuh. Architecture. Available at <https://documentation.wazuh.com/3.12/getting-started/architecture.html>. Accessed on 11-May-2020.
- [36] Gerhard Weiss, editor. *Multiagent Systems: A Modern Approach to Distributed Artificial Intelligence*. MIT Press, Cambridge, MA, USA, 1999.
- [37] Michael Wooldridge. *An Introduction to Multiagent Systems*. Wiley, Chichester, UK, 2 edition, 2009.

Appendices

A Existing Tools / Techniques

During the course of this work, we explored many different tools to see if any could support the analysis and documentation that we were looking to perform. Originally we used simple diagramming tools like Microsoft Visio and Graphviz to create the kinds of diagrams we needed to document our thoughts. A simple whiteboard was also used as we tried to solidify the concepts for how we could build and manipulate the various models we had proposed. However, as we pushed further on the models and tried to document larger systems in higher detail, we found these tools to be quite lacking. If we had a diagram we wanted to show off, we could make such a diagram in these tools but building models, checking for consistency, and manipulating these models into many different views was beyond their scope.

There are many tools and categories of tools when it comes to system architecture. There are a lot of tools and methods aimed at how to document architecture. For example, [20] and [4] focus on ways to document software through multiple views of a system, each with their own purpose. Further, there are a lot of tools for doing modeling and model based engineering of systems which fall on a spectrum from generating documentation to in-depth analysis of systems.

The rest of this section sets out to document the three tools that most closely aligned with our efforts and in some cases influenced our direction.

A.1 OPCAT: Object Process Methodology

The Object Process Methodology (OPM) is a modeling language and technique for designing systems [8]. As the name implies the method focuses on two types of things within a system: (1) stateful objects and (2) processes that transform these objects either by creating, changing, or consuming the object. OPM models include both a textual language for describing systems (Object-Process Language or OPL) and a graphical diagram (Object-Process Diagrams or OPD) that visualizes the model of the system.

The methodology was developed at Enterprise System Modeling Laboratory which is part of the Technion Israel Institute of Technology and has been in development for over 20 years. In 2015 it was also formalized as an ISO standard [17]. Over the years, researchers have published work in which OPM is applied to domains as varied as real-time systems [25], web applications [8], credit card transactions [9] and multi-agent systems [33].

OPM models systems as graph like structures with nodes being either objects or processes and many types of relationships that connect them. These two things lets the modeler represent things at very different levels of detail from abstract top-level system diagrams that may include stakeholder requirements and other pieces of context for the system all the way down to very specific subsystem implementation details. An example top-level graph can be seen in Figure 44a.

The main reason that OPM was appealing to our work was the inclusion of three methods for dealing with model complexity. These methods are: “(1) unfolding/folding, which is used for refining/abstracting the structural hierarchy of a thing and is applied by default to objects; (2) in-zooming/out-zooming, which exposes/hides the inner details of a thing within its enclosing frame and is applied primarily to processes; and (3) state expressing/suppressing, which exposes/hides the states of an object” [27]. These methods of manipulating a model were very similar to concepts we had explored and are discussed in Section 4.

Opcat In order to work with OPM models, the creators of the methodology have also released a tool called Opcat. The tool is a Java-based tool that provides a GUI for creating OPD models of the systems and auto-generating the associated OPL models that complement the diagrams. The tool is similar in use to Microsoft Visio in that you can create nodes (Processes, Objects, or States) and then create arrows of different types for the relationships between these nodes. This makes the tool very easy to use and get started modeling. Figure 44 shows an example of a top-level diagram (System Diagram in OPM terms) that was created to represent the OSSEC case study. As the graphical diagram in (a) was created, the tool produced the textual output in (b).

Once a System Diagram is created, the abstract objects and processes can be further refined until the model has reached a sufficient level of detail. This refinement is done through the unfolding and in-zooming concepts which create new views / diagrams that can then contain further amounts of detail. A key concept in OPM is that each diagram only contains the level of detail needed for a given analysis / description task. In addition to just refinement, other “views” can be created which show partial sets of abstract and/or refined components of a system.

Opcat also lets you “test” models in which they are instantiated with objects being set to their default states. Then, as the processes are activated, state changes are simulated and updated in a step-wise function. This allows for users to explore the state transitions as they occur within the system model.

One thing to note, Opcat was originally a freely available (closed-source) software package that could be downloaded and used offline. However, development appears to have stopped on Opcat and a new web-based product, OpCloud [22] has taken its place. Further, access to OpCloud seems to require contacting the Enterprise Systems Modeling Laboratory at Technion and paying for a subscription. The information in this section is based on our use of the offline Opcat tool and not the OpCloud tool.

Case Study: OSSEC Figure 44 is an example of a top-level System Diagram in OPM terms. This diagram provides the surrounding context for the system. In our case, this shows the OSSEC system, the Enterprise IT system that the system would be a part of, and the goal of the system to detect a threat actor that is trying to compromise the Enterprise IT system. Further, both the “System Compromise Probability” and “Attacker Detection and Response Ability” objects show examples of object state values. These two objects have two states each and switch their states when OSSEC is used / activated. This demonstrates the goals of OSSEC to reduce the likelihood of compromise and increase the ability for detection and response.

Objects can have states as depicted in Figure 44, although these states can also be hidden or “suppressed”. OPM advocates for only including states of objects in diagrams when it is important to the specific design or analysis being performed with that view. This is the third mechanism included to address complexity of system modeling.

Figure 45 shows the initial refinement of our system diagram. This shows that a system is made up of both a server and an agent. The agent has a sensing functionality which creates “Sensor Data” which is then sent to the Server. The Server receives the “Sensor Data”, analyzes it, and sends the analysis result (i.e. active response commands) to the Agent. The Agent then uses these results to drive its actuation. If you look closely, you will also see that both Agent and Server are objects themselves. Objects do not have the concept of sub-objects nor can they include processes within them. Instead, these objects “Exhibit” both other objects and processes, allowing us to relate them together. This is

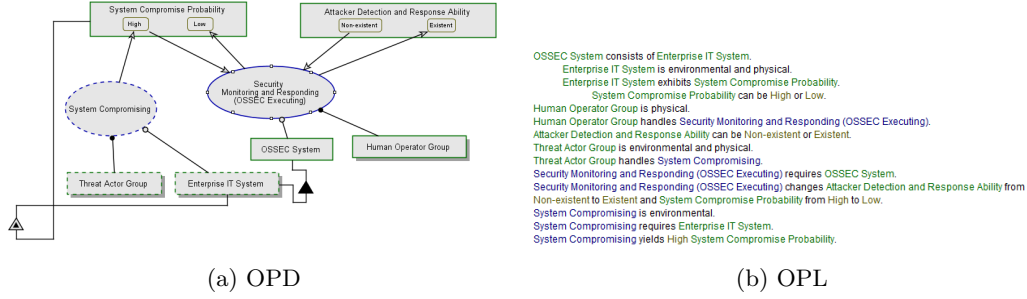


Figure 44: OSSEC System Diagram

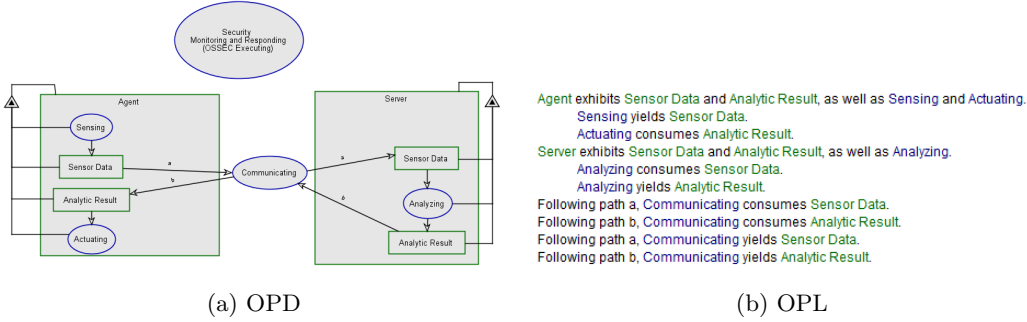


Figure 45: OSSEC Top Down View OPM Models in Opcat

an example of an “Unfolded” view of both Agent and Server. To abstract this notion, we could have folded both Agent and Server and had just a single process that linked these two objects.

Discussion Overall, Opcat and OPM in general is an interesting approach to system modeling that has much potential when it comes to modeling MRS systems. One major piece that is lacking is any built-in support for assurance cases or tracking of assurance arguments. Currently this would require a secondary tool and it does not appear easy to modify Opcat to add such functionality due to its closed-source and un-maintained status. Further, due to this closed-source nature, there did not appear to be any sort of large community or projects using Opcat, though the OpCloud website claims that it is used by many Fortune 500 companies including the “Automotive industry, Aviation industry, White appliances industry, Energy companies, Space agencies, Insurance companies.” Either way, the techniques demonstrated in OPM could be further explored as MRS modeling and analysis techniques are further developed. At the very least, future work should seek to model the ease of use of the tooling and the methods for dealing with model complexity.

A.2 GROOVE - Graph Rewriting

The area of graph rewriting, also known as graph grammars or graph transformations, is an area of computer science focused on creating a new graph based on a starting graph. More specifically, graph rewriting systems take as input a graph and set of rules which define ways

to generate new graphs. These rules and the ability to transform the graphs are governed by formal algebraic methods. For example, [21] sets out to explain the algebraic theory of graph grammars and the associated double-pushout (DPO) approach to rewriting graphs. These formal methods allow for the rewrite systems to make specific guarantees about the resulting graphs as well as the invert ability of rules.

Graph rewriting has been used in many areas to include, mechanical engineering, biology, and software engineering. It is this last domain that was of the most relevance to the goals of modeling MRSs. [3] explores how graph rewriting can be used for software design transformations. More specifically, it explores how graph rewriting can support the Model Driven Architecture (MDA) methodology [19] proposed by the Object Management Group (OMG) to take abstract system designs and transform them into multiple different implementation designs based on platform-specific details. This technique of manipulating system designs and converting between abstract and implementation level details was part of the inspiration for exploring graph rewriting and graph transformations as a core piece of our work.

GROOVE Our exploration of graph rewriting focused on if we could use the grammar defined in Section 3 and create rewriting rules to implement some of the transformations discussed in Section 4. In order to do this, we searched for a tool that could support an interactive exploration of building and applying rules to a graph. There are many different tools that use graph rewriting and allow for the exploration, but we settled on the GROOVE (GRaphs for Object-Oriented VERification)[29] tool.

GROOVE is an open-source tool developed by researchers at the University of Twente and has been in development since at least 2003. As the name implies, it has the specific focus of applying graph rewriting techniques to analysis and verification of object-oriented systems. Given this focus, it was closely aligned with our goals of modeling MRSs as graphs. GROOVE is a package that includes many subcomponents including an editor for creating graphs and rules, a simulator for conducting graph transformations based a set of rules, and a generator which automatically explores the state spaces created by these transformations. [28]

Some of the nice features of GROOVE was the GUI simulator and editor which allowed for interactive exploration of graph rewriting. A user could define a base/start graph, a type graph that defined various type checking details for nodes in the graph and associated relationships, and a system of rules that defined what operations could happen to transform the graph. The interactive exploration would then show the different possible match conditions of the rules, and allow a user to selectively apply rules and see the resulting graph structure. These rules could then be rolled back and other paths through the possible state tree could be explored. That is, you could choose a different ordering of rules and see the new resultant state of the graph.

Case Study: Weather Service Recall the Weather Service case study as explored in Section 5 and the details of the grammar that were presented in Section 3. In the rest of this section, we will walk through some examples of defining the grammar, graphs, and transformations in graph-rewriting terms within the GROOVE environment.

The type system within GROOVE aligned well with the grammar we specified for our models, so this was implemented as a type graph shown in Figure 46. Each node has a specific type that is defined in the type graph as well as a name attribute. Type-checking functionality was then used as we built graphs of the weather service system itself.

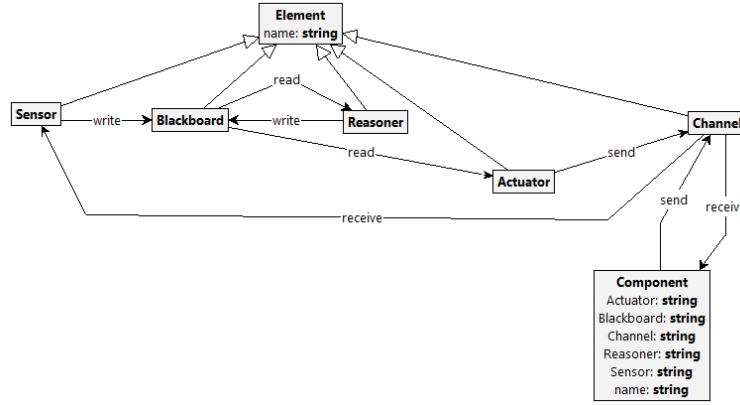


Figure 46: GROOVE Type Graph Example

Figure 47 shows three rules that have been defined based on the functions presented in Section 4.2. These are combination diagrams that represent both the left-hand-side (LHS) and right-hand-side (RHS) portions of a graph transformation rule. While typical representations of LHS and RHS portions of a rule use two separate diagrams, GROOVE includes them in the same window to make creating the rules easier. Objects in blue will be removed while objects in green will be created. GROOVE also adds in extra functionality that allows for referencing nodes without specifying them explicitly. For example, see the $@ - - > \forall$ symbols in the rules which is used to represent all nodes that may be connected to a given node. Additionally, all of the rules have been parameterized, as denoted by the ?0 and ?1 decorators on string objects, which allows for data values to be passed in as arguments to the rule. This allows us to create a few rules and apply them in many different situations by changing the parameters provided to the rules.

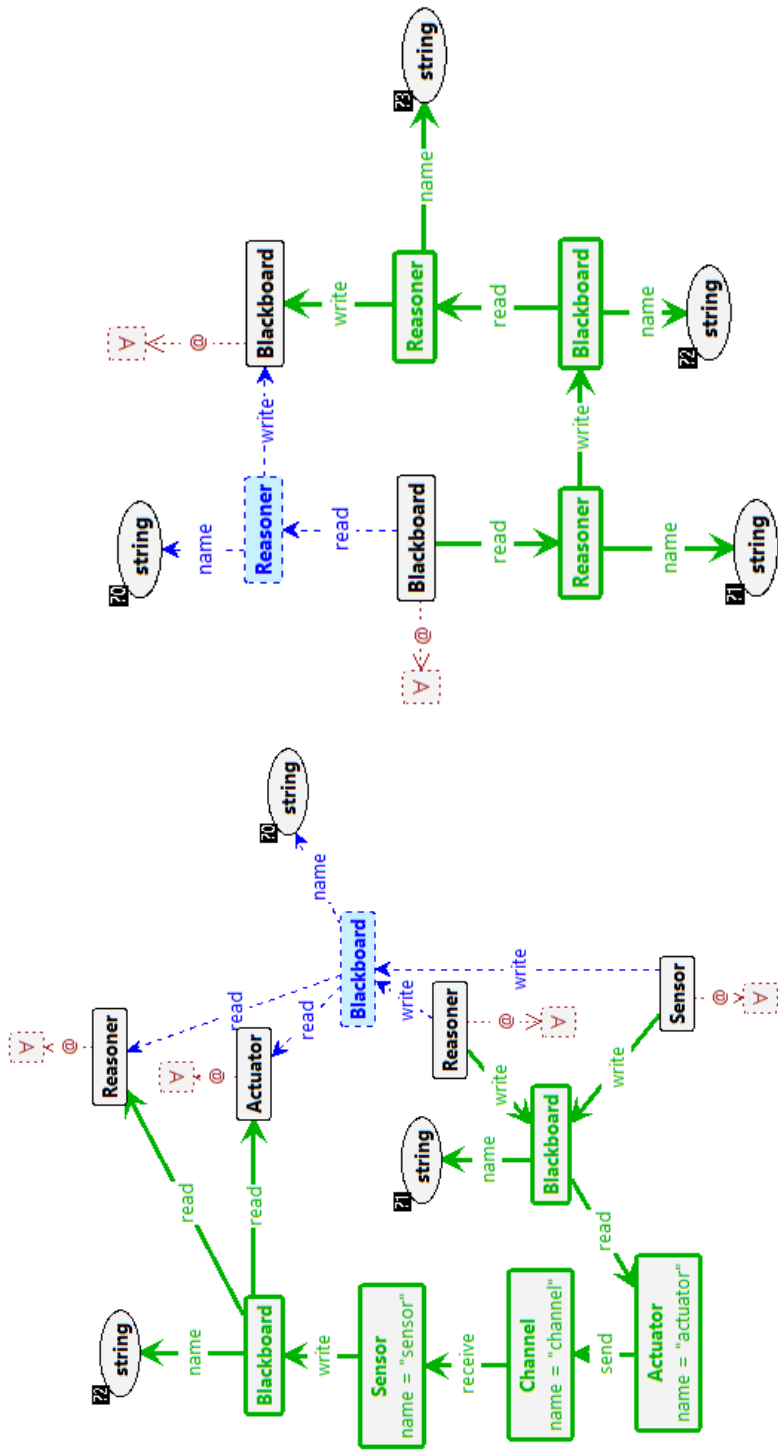


Figure 47: Examples of Graph Transformation Rules in GROOVE
In these rules, blue objects represent nodes and edges to be deleted and green objects represent nodes and edges to be created.

```

1 // We want to verify the correctness of the forecasting
2 // function.
3
4 // First we need to split the forecaster
5 split_reasoner("Forecaster",
6               "Forecast Generator",
7               "Forecast Output",
8               "Forecast Displayer");
9
10 // As we're only interested in the Forecast Output,
11 // we'll need to divide the Forecast output blackboard
12 blackboard_divide("Forecast Output", "Forecast Output 1", "Forecast Output 2");
13
14 // Now we can obscure those pieces we are not interested in
15 obscure("Report");
16 obscure("Recorder");
17 obscure("actuator");
18 obscure("channel");
19 obscure("sensor");
20 obscure("Forecast Output 2");
21 obscure("Forecast Displayer");
22 obscure("Global Forecast");

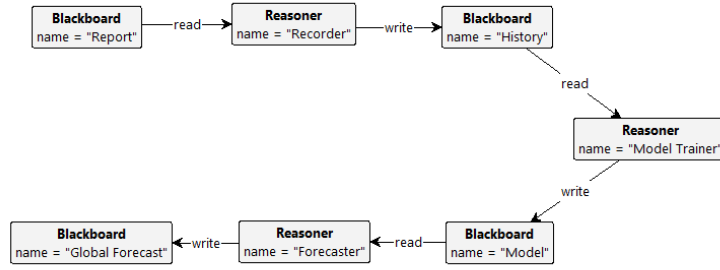
```

Figure 48: GROOVE Control Program

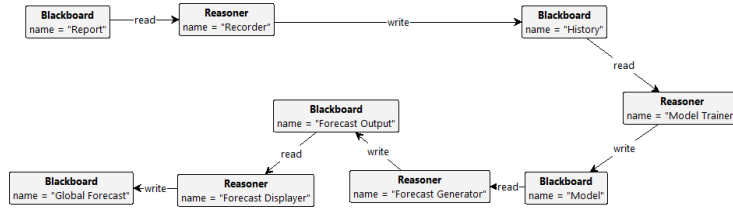
GROOVE provides a graphical interface through the Simulator program that allows for the creation and exploration of graph grammars. Part of this function allows for a user to either manually select rules to be applied to the starting graph or let GROOVE perform an exploration of the grammar. The default behavior for this exploration is an exhaustive search of all possible orderings of valid rule applications. This can be performed in a breadth-first or depth-first search manner. However, GROOVE also allows for more complex search methods to include assigning priorities to rules in order to constrain their application as well as the creation of control programs. Figure 48 shows a control program that implements the manipulations performed in Section 5 for investigating the correctness of the weather service. First the Forecaster reasoner is split, then the Forecast Output blackboard is split and finally unneeded portions of the system view are obscured.

Now suppose we performed one more Blackboard Divide operation after those performed above but this time on the “History” node. This would result in the graph shown in Figure 50. While we took one path to get to this final state, we could have also split the “History” node first before applying the control program.

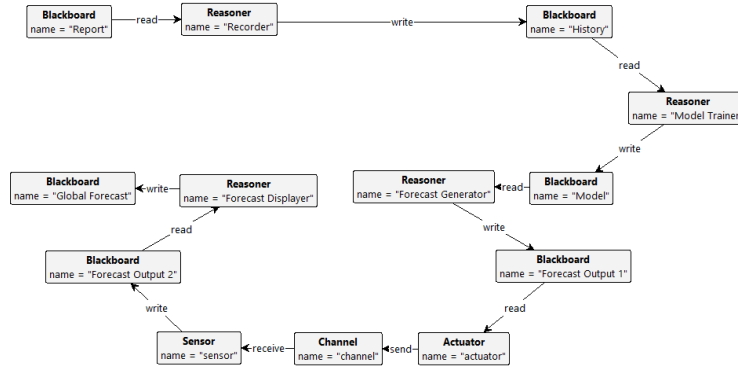
These multiple different paths through the state space is what the GROOVE simulator is designed to explore and track via the labeled transition system (LTS) panel. Figure 51 shows a partial view of this state showing the two different paths we could have explored. The state represented by node *s2* on the left-hand side of the diagram represents the path in which “Forecast Output” is divided first and then “History” is divided second. State *s3* on the right-hand side of the diagram is the other order. At the bottom of the diagram we see state *s4* which shows that both paths result in the same graph, or view of the system. This tree of states is very similar to the idea of the Tree of Views we propose in Section 4.3 and our explorations with GROOVE were one of the motivations for this idea.



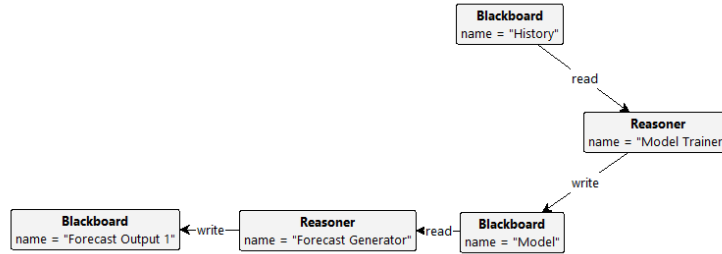
(a) Starting Graph of High-level Weather Service System View



(b) Split "Forecaster" Reasoner



(c) Split "Forecast Output" Blackboard



(d) Final View for Analyzing Correctness of Forecasting Function

Figure 49: Resulting State of the Graph during application of Control Program from Figure 48.

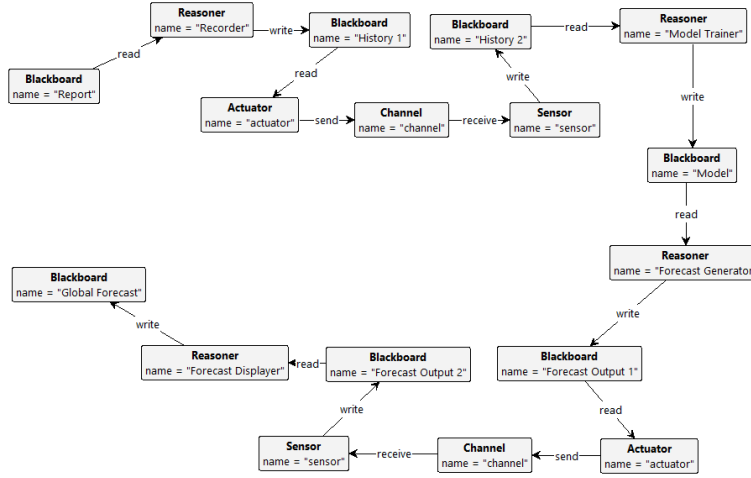


Figure 50: GROOVE Result 2

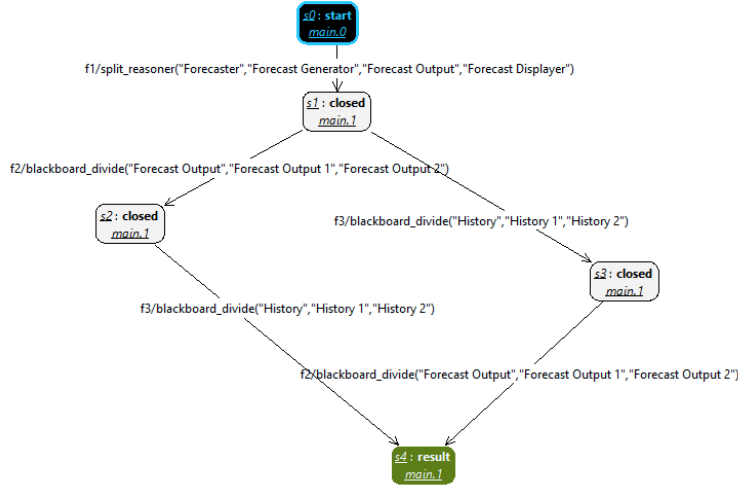


Figure 51: Tree of States in GROOVE

Discussion Overall, graph rewriting techniques and the GROOVE tool in particular showed a lot of close alignment with the kinds of ideas described in the previous sections. First, the type-checking functionality allowed for the tool to build a representation of the MRS grammar that we defined. While this grammar may need to be tweaked, having a representation of it within the modeling tool allowed for explorations on both the grammar itself as well as the rules used for transformations of the graphs.

Second, the LTS view was an inspiration for our tree-of-views concept that tracked the different views of a system and how they related to all of the other views that an analyst

may create. Unlike other architecture documentation tools that allow a user to create many different independent views of a system, such a tree-of-views concept would link all of these views together through the use of graph rewriting rules. This allows for changes made to the model to be reflected in all of the different views. More importantly, if there are any problems introduced by making a change in one part of the model, it could be highlighted by problems introduced in the application of the transformation rules in another part of the system.

Finally, the feature of control programs allowed us to simulate the kind of interactive application of rules that we were looking for. This differed from other tools where explicit matches needed to be defined. Instead, generic rules could be defined and the application of these rules could be controlled through these programs. Ideally, a tool would allow a user to interactively select pieces of the current view and then choose from a set of available rules to apply (e.g. split, collapse, etc) all while ensuring the transformation aligned with formal rules.

The one major feature that was lacking in GROOVE was any sort of assurance case type functionality. In future explorations of GROOVE, it would be interesting to explore how goals or evidence within an assurance case could be tied to or at least reference specific views, or states in the LTS, within GROOVE. This would allow documentation of why a certain view was created and the analysis that was being performed.

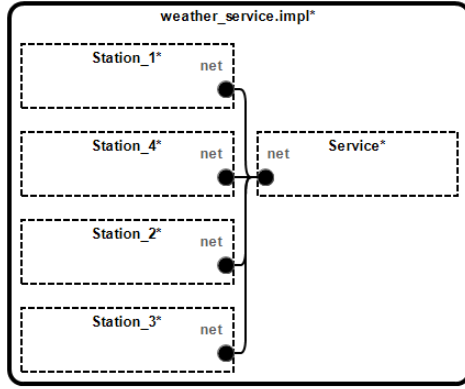
A.3 OSATE - Architecture Analysis and Design Language (AADL)

The Architecture Analysis and Design Language (AADL), formerly the Avionics Architecture and Design Language, is an SAE standard (AS5506A) that was released in 2006 [12, 15]. At its core it is an architecture description language (ADL) with a strong focus on modeling embedded systems, to include both software and hardware architecture. These models can then be used as the basis of design documents, analysis such as cost, information flow, and scheduling, or code generation of the underlying components.

AADL also provides multiple ways in which to extend the core language which has allowed a large community to adopt AADL as an underlying tool to support many types of formal modeling and analysis. This community includes organizations like SAE, Collins Aerospace, and the Software Engineering Institute (SEI), and many other performers under the DARPA HACMS [31] and CASE [30] projects. The CASE project has even investigated how to use AADL to provide assurance of learning systems [5] which is closely aligned with the type of decision control systems present in MRSs. This flexibility and large community of contributors was an appealing feature as we looked for tools and researchers related to the kinds of analysis we were investigating.

AADL has been incorporated in multiple tools and workflows, with the Open Source AADL Tool Environment (OSATE) from CMU/SEI being one of the most popular [34]. This tool is built on top of the Eclipse Modeling Framework (EMF) and provides a development environment for building and analyzing AADL models. Within this environment, users can build the models as code and then generate diagrams of different views of these models. The interface for exploring the diagrams was similar to the GUI portions of the GROOVE and Opcat tools discussed earlier. However unlike those tools, the primary way of developing the models was to code them up using the AADL language itself as well as the various language annexes.

Many of the extensions to AADL have been built as plugins that incorporate with OSATE and enable new types of analysis. Two extensions that were of the most interest to



(a) AADL Diagram

```

package mrs::weather_service
@public

abstract Station
  features
    net: feature;
  end Station;

abstract Service
  features
    net: feature;
  end Service;

system weather_service

end weather_service;

system implementation weather_service.impl
  subcomponents
    Station_1: abstract mrs::weather_service::Station;
    Station_2: abstract mrs::weather_service::Station;
    Station_3: abstract mrs::weather_service::Station;
    Station_4: abstract mrs::weather_service::Station;
    Service: abstract mrs::weather_service::Service;

  connections
    co0: feature Station_1.net <-> Service.net;
    co1: feature Station_2.net <-> Service.net;
    co2: feature Station_3.net <-> Service.net;
    co3: feature Station_4.net <-> Service.net;

  end weather_service.impl;
end mrs::weather_service;

```

(b) AADL Model Code

Figure 52: Weather Service Overview Diagram in OSATE

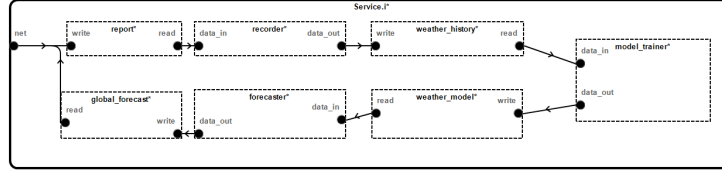
us were the Architecture-Led Incremental System Assurance (ALISA) [6] and Resolute [13] packages which provide support for building assurance cases that are tied to the architectural model. ALISA defines a syntax for specifying system requirements (ReqSpec [11]) which are the basis for the Assurance Case itself. These requirements can be nested in nature, similar to the goals defined in Section 6. Resolute provides a language for defining the assurance case as well as analysis functions to perform against an instantiation of the AADL model to check if each of the assurance arguments holds.

In developing our ideas, we have explored breaking case studies up into three sections. First, Components would define pieces of the actual system that existed. Second, Functions would define what core bits of logic that were being performed by the system. Third, Properties would focus on how those functions needed to be performed, for example integrity and confidentiality requirements.

Similarly, ALISA focuses on defining 1.) the AADL model itself, 2.) functional requirements, and 3.) non-functional requirements. What we called components aligned nicely with the AADL models themselves. That is, specifying what actually exists within the system. What we called Functions aligned nicely with Functional Requirements which could be specified via the ReqSpec notation within OSATE. Finally, what we were calling properties seemed to nicely align with nonfunctional attributes, usually including things like performance, timeliness, and fault tolerance.

The rest of this section demonstrates how AADL and OSATE can be used to model and build an assurance case for the Weather Service and OSSEC case studies.

Case Study: Weather Service One of the first steps in working with AADL is to start creating the model of your system. The language allows for modeling many well-known component abstractions including software ones like threads, data, processes, subprograms and hardware ones like devices, and memory, and buses. When starting out, using the



(a) Diagram

```

system implementation Service.i
subcomponents
  report: abstract mrs::weather_service::Blackboard;
  recorder: abstract Reasoner;
  weather_history: abstract Blackboard;
  model_trainer: abstract Reasoner;
  weather_model: abstract Blackboard;
  forecaster: abstract Reasoner;
  global_forecast: abstract Blackboard;
connections
  c0: feature report.read -> recorder.data_in;
  c1: feature recorder.data_out -> weather_history.write;
  c2: feature weather_history.read -> model_trainer.data_in;
  c3: feature model_trainer.data_out -> weather_model.write;
  c4: feature weather_model.read -> forecaster.data_in;
  c5: feature forecaster.data_out -> global_forecast.write;
  c6: feature global_forecast.read -> net;
  c7: feature net -> report.write;
end Service.i;

```

(b) AADL Model

Figure 53: Weather Service Detailed Implementation

“abstract” type allows you to mock things out before filling them in with more specific details. In addition to components, there is a concept of a system which is made up of components and connections between these components. Finally, components can have many different properties associated with them.

Figure 52 shows the AADL model code and related diagram of a high-level view of the Weather Service and stations. The model code includes an example of how abstract components are defined, along with “features” which act as ports and used as endpoints for connections. The language allows many different qualifiers to be associated with these features such as directionality (i.e. in, out, both), type (e.g. data, events, subprogram access), and even requirements such as bus access. For this example we have defined both Stations and Services to be abstract components with a simple unqualified “net” feature each. These are then combined in a system implementation called *weather_service.impl*.

To expand part of the model and add more details requires creating a system implementation for one of the components. In Figure 53 we show the system implementation for the *Service* component of model. We can then go back and edit the model in Figure 52b such that the *Service* component was of the *Service.i* type, representing an implementation instead of just an abstract component. This would allow for all of the details exposed in Figure 53b to be exposed in the the top level diagram from Figure 52a. OSATE allows a user to select whichever details from a component to include so that only the pertinent details are shown.

In addition to creating the AADL model, a user can start to create requirements for the system as well as a verification plan for how to to verify those requirements. Figures 54 and

```

system requirements weather_service_reqs for mrs::weather_service::weather_service.impl [
  description "These are requirements for Weather Service System"

  requirement R1 : "Correctness of Weather Service" [
    description "Weather Service Operates Correctly"
    rationale "It should work correctly"
  ]

  requirement R1_1 : "Ingestion" [
    description "Weather Service Correctly Ingests Information"
    rationale "It should ingest correctly"
  ]

  requirement R1_1_1 : "Ingestion De-multiplexing" [
    description "De-multiplexing: Correctly handle reports from multiple locations"
  ]

  requirement R1_1_2 : "Ingestion De-duplication" [
    description "De-duplication: Ensure no duplicate entries"
  ]
]

```

Figure 54: Weather Service Requirements (Partial)

```

verification plan weather_service_vplan for weather_service_reqs [
  claim R1 [
    claim R1_1 [
      claim R1_1_1 [
        activities
        codeReview: VerificationMethods.CodeReview("Recorder")
        codeTesting: VerificationMethods.CodeTesting("Recorder")
      ]
      claim R1_1_2 [
        activities
        codeReview: VerificationMethods.CodeReview("Recorder")
        codeTesting: VerificationMethods.CodeTesting("Recorder")
      ]
      claim R1_1_3 [
        activities
        mandatoryAccessControl: VerificationMethods.hasMandatoryAccessControl("Weather History")
        noBufferOverflows: VerificationMethods.ReviewForBufferOverflows("Recorder")
      ]
    ]
  ]
]

```

Figure 55: Weather Service Verification Plan (Partial)

55 provide partial examples of both of these for the Weather Service example. Requirements allow for the specification of a lot of information such as links to Stakeholder goals, references to other related requirements, descriptions, and rationale. Requirements are specified for a system implementation. While we have specified a single plan here for the top level weather service, multiple plans can be created for each implementation. This way the individual requirements documents are easier to read and manage.

Verification plans are defined against a requirements document and multiple plans can be registered much in the same way as requirements documents. Within the plan “Claims” are defined which align with a specific requirement. These claims can then either contain sub-claims or a list of activities that make up the verification procedures for a given requirement. By using nested sub-claims, we can create similar assurance cases as those presented in

Evidence	Pass	Fail	Error	Todo	Description
Case WeatherServiceCase				12	
Plan WeatherServicePlan(weather_service.impl)				12	
Claim R1				8	Weather Service Operates Correctly
Claim R1_1				6	Weather Service Correctly Ingests Information
Claim R1_1_1				2	De-multiplexing: Correctly handle reports from multiple locations
Evidence codeReview				1	Typical code review to search for any glaring problems
Evidence codeTesting				1	Typical code testing including unit and integration tests
Claim R1_1_2				2	De-duplication: Ensure no duplicate entries
Evidence codeReview				1	Typical code review to search for any glaring problems
Evidence codeTesting				1	Typical code testing including unit and integration tests
Claim R1_1_3				2	Ingestion is not impacted by the rest of the system
Evidence mandatoryAccessControl				1	A code review is conducted to verify if SELinux or another MAC is used.
Evidence noBufferOverflows				1	Testing to search for buffer overflows is conducted
Claim R1_2				1	Weather Service Correctly Ingests Information
Claim R1_3				1	Weather Service Correctly Reports Forecasts
Claim R1_3_1				1	Reporting is not impacted negatively by the rest of the system
Claim R2				3	Weather Stations Operate Correctly
Claim R2_1				1	Data Update and Reporting Functions work correctly
Claim R2_2				1	Forecast Ingestion and Extract works correctly
Claim R2_3				1	Forecast is Displayed Correctly
Claim R3				1	Comms between Stations and Service should work correctly
Evidence placeholder				1	Placeholder for future activities

Figure 56: Weather Service Assurance Case

Section 6. Finally, verification plans can be linked to other systems such that the verification of one plan will trigger the verification of dependent components.

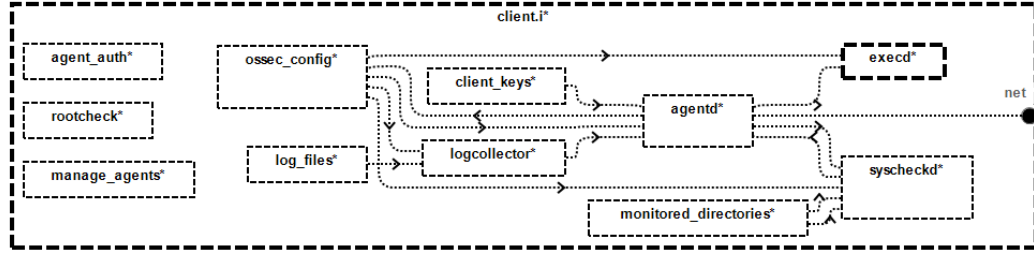
The activities that are used to carry out the verification can be manual steps to be completed by humans, java code which may run unit tests or other software testing activities, or Resolute methods which can analyze the AADL model itself. For this example we have just defined some manual steps, but an example of Resolute code is demonstrated in the next section.

The final step in creating an assurance case is to run the verification plan. This steps through all of the claims and executes the various methods, or marks them as needing to be completed in the case of manual steps. At the end of this process, an assurance case is generated and can be explored through an interface as presented in Figure 56. The nested structure can be used to explore the various components of the assurance case. This example mimics the tree of views structure from Section 5.

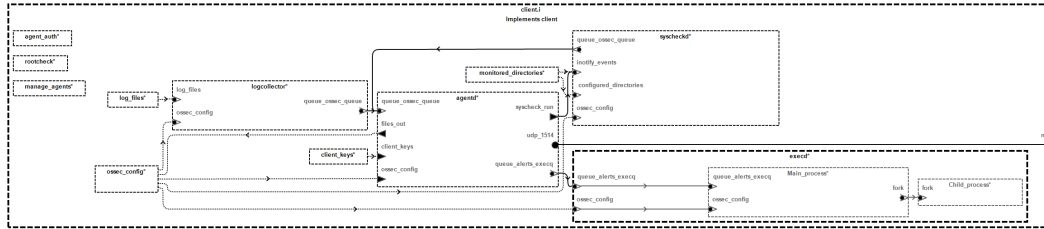
Case Study: OSSEC For OSSEC, we followed the same process as described above to create a detailed model for many of the components of the system. An abstracted and detailed view of the Client can be seen in Figure 57. Both of these views were created from the same *client.i* system implementation with the only difference being how many details were exposed in the diagram view. While transforming the models like is possible in Opcat or GROOVE is not possible, this does allow for some level of abstraction mechanisms to deal with model complexity.

Section 6 went into a lot of detail when developing an assurance case for OSSEC and the case was modeled in OSATE and can be seen in Figure 58. For the Weather service assurance case, we used only manual methods which resulted in only TODO items. Here, we show an example of other types of methods we could define. For example, we could define an assumption that we make and say it should show as passing. We could also define methods in Resolute which will analyze the model itself.

We have defined two sets of methods in Resolute. The first two methods represent



(a) Abstract View



(b) Detailed View

Figure 57: OSSEC Client Diagrams

simple assumption markers. That is, we can designate some claim as an assumption that we either believe holds or fails to hold. The second set deals with connections and the use of encryption. As our model does not include any code or details about communication stacks, we assign properties to *features* that we have verified are using encryption. The only *feature* in the part of the assurance case shown that uses encryption is the network communication the Client uses over UDP port 1514. This communication is encrypted using the key files that were discussed in Section 6. Some of the other features may not make sense to use encryption, for example the IPC mechanisms. To address these failing checks, we either need to further refine our verification plan, resolute code, or assurance case to better represent the argument.

By leveraging the Resolute code in this manner, we are able to directly link the assurance case to the system model. Then, verification can be rerun each time the model or assurance case changes, providing an up-to-date understanding of the current assurance of the system. In addition to Resolute methods, ALISA can also call out to other types of code. For example, source code can be attached to software components and associated unit tests from this software can be called as part of the verification plan. This way, a requirement for tests passing can also be included in an assurance case and allows for added flexibility.

Additionally, OSATE provides many examples in their Github repository² of both models and how to use many of the plugins. The security example projects (ALISASecurityExamples and E_EnabledAircraft_Security) have many useful Resolute methods defined which include activities to assess if all connections were using encryption, if that encryption met key length requirements, if components are using authentication, reason about multilevel security flows, and more. These examples are a rich starting point for anyone getting started with building models and assurance cases with OSATE.

²<https://github.com/osate/examples>

▼ [F] Claim G1_1	1	1	1	All commands executed by an agent were sent by the authorized manager
> ✓ Claim G1_1_partial_1	1			Only the legitimate manager has access to the client key
> ? Claim G1_1_partial_2			1	All parties are authenticated prior to transferring data
▼ [F] Claim G1_1_partial_3		1		All communications use encryption
▼ [F] Evidence allEncrypted		1		All component ports have encryption
▼ [F] Failure: allComponentsEncrypted				All components have encrypted connections
▼ [F] Failure: isEncryptedLeafClaim				Component agentd : mrs::ossec::client::agentd::agentd has all features encrypted
> [F] Failure: isFeatureEncrypted				Feature Client_Linux.agentd.client_keys is encrypted
> [F] Failure: isFeatureEncrypted				Feature Client_Linux.agentd.files_out is encrypted
> [F] Failure: isFeatureEncrypted				Feature Client_Linux.agentd.ossec_config is encrypted
> [F] Failure: isFeatureEncrypted				Feature Client_Linux.agentd.queue_alerts_execq is encrypted
> [F] Failure: isFeatureEncrypted				Feature Client_Linux.agentd.queue_ossec_queue is encrypted
> [F] Failure: isFeatureEncrypted				Feature Client_Linux.agentd.syscheck_run is encrypted
> ✓ Success: isFeatureEncrypted				Feature Client_Linux.agentd.udp_1514 is encrypted

Figure 58: Partial OSSEC Assurance Case

```

assumption(e: aadl) <=
  ** "This is assumed true" **
  true

invalid_assumption(e: aadl) <=
  ** "This assumption does not hold" **
  fail ** "This assumption does not hold" **

allComponentsEncrypted(root : component) <=
  ** "All components have encrypted connections" **
  forall(c: component). subcomponents(c) = {} => isEncryptedLeafClaim(c)

isEncryptedLeafClaim(c: component) <=
  ** "Component " c " has all features encrypted" **
  forall (f: features(c)). isFeatureEncrypted(f)

isFeatureEncrypted(f: feature) <=
  ** "Feature " f " is encrypted" **
  property(f,prop.encrypted)
  or else fail ** "Feature " f " is not encrypted" **

```

Figure 59: Example of Resolute Methods

Discussion Overall, OSATE and AADL provided an interesting environment to test out some of our ideas. It shows real promise with being able to create assurance cases and attach the left nodes to the architectural model itself through the Resolute extension. Further, the approach that the ALISA method takes with defining requirements, functional and non-functional, against the architectural model aligns with our ideas of breaking a system up by components, functions, and properties. Also with the large community that is using and contributing to the OSATE ecosystem a lot of useful extensions have been added which could be leveraged for further model analysis.

However, we did not find a feature in OSATE in which models could be transformed. While models can be extended over time, there is not built in tracking of these changes. Version control software could be used to track the model files, but this is not the kind of “Tree of Views” support we saw in GROOVE. If we could augment OSATE to incorporate some of the graph rewriting ideas and features from GROOVE we would be much closer to

the kind of tooling we were ultimately searching for.

A.4 Summary

Throughout our explorations of these tools we have identified many useful features within each of the tools as well as missing features to fully execute our vision of MRS modeling and analysis. For OPM and Opcat the mechanisms for dealing with model complexity were very similar to our ideas of transformations and the approach was mostly straightforward and easy to get started with. However it lacked the ability to integrate with a framework for requirements and assurance cases.

For graph rewriting and GROOVE, the ability to define and transform typed graphs was directly in line with our definition of a grammar and transformations. Further, the ability to follow the series of transformations through a state space with the LTS view was almost exactly what we were looking for with regards to the tree of views concept and even influenced our thoughts around this concept. However, the selection and application of rules from the GUI left a lot to be desired with rules having to be parameterized and called from a control program instead. It also did not have any support for assurance cases.

Finally, AADL and OSATE were very close to the modeling and analysis concepts we had though typically used for real-time systems. Therefore there was great support for building models and linking them to assurance cases. The methods employed to decompose a system even mirrored our approach with the use of functional and non-functional requirements. However, the major piece missing was the ability to transform the model itself without just editing the model code. Assurance cases could be built to point to parts of the model, but it was difficult to tell how one diagram may relate to another as there was no tree of views concept.

Overall, these three tools represent a great starting point for future work and give hope that tooling to support our proposed modeling and analysis methodology may not be far away.