A Modeling Tool for System of Systems

Communication-oriented modeling in the Papyrus framework

Emil Palmelund Voldby
Jonas Madsen
Sean Kristian Remond Harbo

Spring 2022

AALBORG UNIVERSITY
STUDENT REPORT

Title:

A Modeling Tool for System of Systems

Subtitle:

Communication-oriented modeling in the Papyrus framework

Group Name:

cs-22-ds-10-08

Supervisor:

Michele Albano

Group Members:

Jonas Madsen
Emil Palmelund Voldby
Sean Kristian Remond Harbo

Project Page Count: 136
Total Page Count: 146
Completion date: June 10, 2022

This report presents a Domain-Specific Modeling Language for System of Systems called "Abstract Communicating Systems". We give formal syntax and semantics for the modeling language. We provide a mapping to UML using a UML Profile diagram and create an Integrated Development Environment tool using Eclipse and Papyrus. We provide the source code freely.

We perform a qualitative usability test using the tool on three domain experts, they rate the tool an average of 8/10 even though the tool still has some issues. A Journal documents the process and we reflect on the process and product.

The main contribution is the realization of a new approach to System of Systems modeling.

Summary

In recent years the number of internet-connected devices has continued to rise[1], this paradigm is often referred to as the Internet of Things (IoT). All these interconnected devices raise serious problems related to security and manageability as the complexity of the overall system increases. Modeling technologies exist that address different parts of the problem from generalized systems modeling[2] to specialized robotics modeling[3]. We argue there is a need for a modeling tool that captures the complex interactions that can arise from System of systems (SoS) enabled by IoT.

This report outlines the design and realization of a language named “Abstract Communicating Systems” (ACS). ACS is designed to address SoS by capturing the communicative properties that arise in SoS. We define ACS rigorously in terms of both the syntactic components and their semantics. ACS features four diagrams that define the structure and behavior of systems, a textual syntax is used to denote communication properties like timing information and data type information. ACS is designed to support a wide range of communication paradigms like Peer-2-Peer, publish-subscribe, client-server, and more. ACS is an attempt at a protocol and paradigm agnostic solution to SoS communication modeling and verification. An integrated development environment (IDE) is constructed to support ACS modeling in the Papyrus Eclipse framework[4], it enables easy creation and modification of ACS models. The IDE also provides validation facilities and error messages that guide a user to a valid ACS model. The implementation is freely available from Github[5].

SysML[2] and UML[6] are generalized modeling technologies that can facilitate systems modeling and requirements specification but have no explicit communication verification support. UPPAAL[7] and other model checkers[8] can verify models but otherwise has little support for modeling SoS concepts. ACS attempts to bind these two approaches into a new approach, that to the best of our knowledge does not currently exist. Previous attempts (now defunct) were made to build modeling tools for SoS namely DANSE[9] and COMPASS[10]. A previous report[11] explored DANSE and COMPASS as well as their relation to ACS and concluded there were significant improvements to be made.

The ACS tool is used to present ACS to three industry experts and usability tests are conducted with the results summarized in the conclusion of this report.

Preface

This report is written by the university group cs-22-ds-10-08at Aalborg University during the Spring 2022 Master Thesis Semester at Aalborg University. This is a continuation from the Autumn 2021 semester that constituted a Pre-Specialisation semester. We give special thanks to our supervisor Michele Albano, for the great help throughout the process. We thank the three participants that helped us test the ACS IDE for their invaluable feedback and expertise within the field. The intellectual property rights to all original material brought forth in this report belong to the authors.

Aalborg University, June 10, 2022.

Emil Palmelund Voldby
<evoldb17@student.aau.dk>

Jonas Madsen
<jmad17@student.aau.dk>

Sean Kristian Remond Harbo
<sharbo17@student.aau.dk>

Contents

1 Introduction

This project is a continuation of the work on the “Abstract Communicating Systems” modeling framework (ACS), which was initially created as a semester project at AAU in the autumn 2020 [12]. An article [13] about ACS was subsequently published in the “16th International Conference of System of Systems Engineering (SoSE)” in June of 2021.

To our knowledge, ACS provides a novel approach to System of Systems (SoS) modeling and verification that abstracts away system-centric details such as internal system behavior for a thorough model of SoS-level communication. ACS is backed by formal semantics that verify the correctness of communication based on reachability and time safety.

ACS attempts to improve the state-of-the-art within SoS Engineering which previously consisted of general-purpose methods such as UML/SysML and a few SoS specific methodologies with a heavy focus on internal system behavior and details. Our 9th semester project [11] extended upon the initial analysis with a full survey of the state-of-the-art and motivated the discovery of new and old research problems and possible directions.

These research problems and directions, combined with existing SoSE tools, drove the project scope towards 1) improving the ACS formalism (see Chapters 2 and 3) and 2) the implementation of an ACS IDE (see Chapter 4). We base the IDE on Eclipse Papyrus, which has plugin and UML Profile support. We create a UML Profile to implement ACS modeling capabilities and a range of plugins to implement the necessary IDE features such as verification. Finally, we perform usability tests on the IDE (see Section 4.4).

1.1 UML and Metamodeling

UML [14] is a graphical, general-purpose language for modeling system architectures and behavior, which can be used for system documentation, artefact generation, fault analysis, and much more. One such additional use is “metamodelling” [15] which is the process of describing a modeling framework using another modeling framework.

A UML Profile [16] (also mentioned in the intro) is an extension to UML (the metamodel), where *stereotypes* that represent new concepts are made as extensions of components (called metaclasses) from the underlying metamodel. A Profile is applied onto a UML model to transform it into a model of whatever domain the profile is designed to model.

1.2 The Eclipse Ecosystem

The Eclipse Foundation [17] is home to a wide variety of open source software solutions, including the Eclipse IDE which is used to develop new IDE's through extensive plugin support [18].

Eclipse Modeling Tools [19] is part of the Eclipse ecosystem and allows a user to build and compile Eclipse plugins directly into a standalone application. Eclipse Papyrus [20] is itself an Eclipse plugin which provides UML modeling capabilities, and we extend papyrus with ACS specific features, using the extensive plugin support provided by both Modeling Tools and Papyrus. Thus, there is even support for plugins to plugins.

1.3 Existing Tools and Methodologies

The initial analysis [12] and the survey [11] identified two primary contenders to ACS, namely DANSE [21] and COMPASS [22], which have unfortunately been discontinued due to their respective research projects having ended.

ACS' primary difference to these methodologies is that it does not require any information about internal system behavior and only focuses on SoS-level communication, while still providing valid verification results. This allows SoS designers to only focus on SoS-level details, where the other methodologies would require SoS designers to provide system specific details that they might not always be aware of.

Furthermore, since ACS is backed by complete, formal semantics, automatic mappings from ACS to external verification tools and back is possible, which allows seamless integration of said tools into an ACS IDE. Conversely, COMPASS integrated up to a dozen of powerful verification tools, but without any automatic mapping the user had to learn all these formalisms, else they were useless.

2 Improving and Introducing ACS

This chapter doubles as an informal introduction to ACS and its features as well as a walkthrough of the changes and additions that have been made to ACS since its initial publication in the summer of 2021 [13].

At its core, ACS is a diagram-centric modeling and verification methodology for SoSE with a focus on formally verifying the (time-critical) communication between systems (rather than on simulation). As a consequence, ACS abstracts away all internal computations of systems as time constraints in favor of a thorough model of the communication between systems.

As such, SoS designers can focus on modeling communication properties, whereas system designers can focus on complying with the communicative behavior outlined by the model. Of course, not all systems in a SoS are under the SoS designers' influence, in which case the model must instead comply with these systems' communicative behavior.

An ACS model consists of the three layers "Structure", "Behavior", and "Data". The Structure layer models the structure of the SoS and the connections between the systems that communicate. Next, the Behavior layer models the communication systems perform with each other and the constraints on this communication. Finally, the Data layer models the structure of the data/information that is communicated between systems. With this, ACS can verify the structure of a SoS, the communication between systems, the type correctness of communication, and additional properties that utilize multiple layers.

It is important to note, however, that ACS cannot verify that the entirety of any concrete system or implementation will work correctly; only that structural rules are upheld and that the modeled (time-critical) communication is feasible and is type correct.

Section 2.1 presents an overview and introductory description of ACS as a modeling framework and the central constraints and semantics of the various syntactic components. Sections 2.2, 2.3, and 2.4 present more intricate details about the concepts and theories underlying the structure, behavior, and data layers, respectively. Finally, Section 2.5 describes the process of verifying an ACS model.

2.1 Model Components, Diagrams, and Documents

As mentioned in the introduction to this chapter, an ACS model consists of the “Structure”, “Behavior”, and “Data” layers. Whereas the Structure and Behavior layers are represented using diagrams, the Data layer is represented using text documents since this format fits better with ACS’ type system, as will be clarified later on in this section.

2.1.1 Structure Diagrams

Since one SoS can depend on other SoSs, an ACS model consist of possibly many SoS models which may reference each other in a non-cyclic manner. Some organization could, for example, have an ACS model with a SoS model of their own infrastructure alongside a SoS model of some distributed database system that the infrastructure makes use of. At the root of every SoS model in an ACS model lies a single “Structure Diagram”, an example of which is shown in Figure 2.1. As can be seen, the structure consists of different types of (nested) systems (the big boxes) that are interconnected by links (the black diamonds including the edges/arrows).

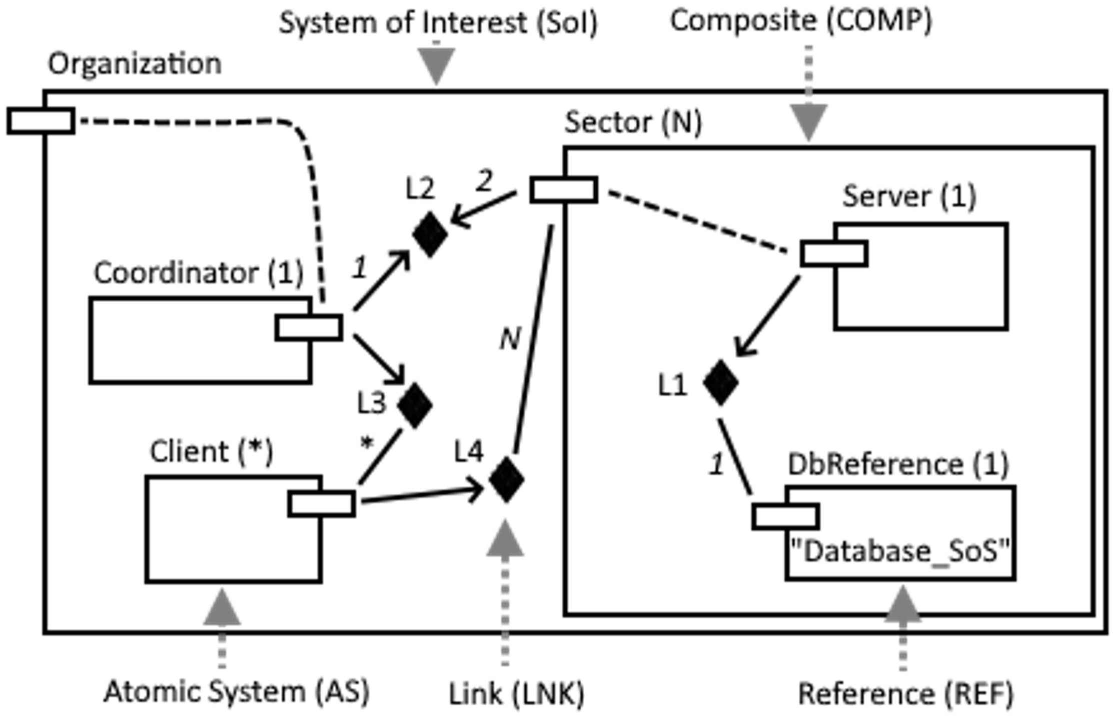


Figure 2.1: An example Structure Diagram for a SoS. The example is made specifically to demonstrate most of ACS’ structural features.

All types of systems have the same general structure; a large box with some number (not necessarily one) of smaller rectangles called "*Ports*" on the side of the box. Ports represent the interfaces through which communication between systems is facilitated. Ports are connected by links, where the black diamond in a link is called the "*Link Hub*" and the edges/arrows between ports are called "*Link Connections*". The dashed lines in Figure 2.1 defines one port as an alias of another. The system "Server" is thus called an "*interface*" of "Sector".

2.1.1.1 Systems

A **`System`** is an abstraction of a behavioral unit with one or more interfaces used for communication. The label above each system displays the system's "*Name*" together with its "*Cardinality*" surrounded by parentheses (except for on the System of Interest). The cardinality can have three types of values: $\mathbb{Z}^+$ (i.e. a positive integer), the value '**`N`**', or the value '`*`' (star).

- $\mathbb{Z}^+$: The number of instances of this system type is known at *design-time*. It is possible to initiate communication with each individual instance of this system type.
- **`N`**: The number of instances of this system type is first known at *run-time*. It is possible to initiate communication with each individual instance of this system type.
- `*`: The number of instances of this system type is *unknown* at both design- and run-time. It is (almost, see Section 2.3) impossible to initiate communication with individual instances of this system type.

Thus, in the example in Figure 2.1, a system cannot initiate communication with a specific *Client* (as the number of instances is unknown), but a system can initiate communication with a specific *Sector* (as we know the number of instances at run-time). It is, however, possible to communicate with star-cardinality-systems in ways elaborated in Sections 2.1.2 and 2.3.

A **`System of Interest (SoI)`** is the single outermost system object in a SoS model and (indirectly) contains all other systems in the SoS-model. The SoI is a system that does not depend on the context of its surrounding environment to function, other than that the environment might initiate communication over the SoI's ports. The SoI-concept is borrowed from ISO-21839:2019 [23] where it refers to "the system whose life cycle is under consideration", however, it can also be perceived simply as a reusable system component, seeing as ACS allows references to other SoS models.

A **`Reference`** is simply a placeholder for the SoI of another SoS model and will provide the same interface and capabilities as the SoI it references. Every reference to the same

SoI creates different system instances that are distinct from each other, so two references cannot reference the same instance. The cardinality of a reference is used in place of the missing Cardinality on the SoI. The SoI cannot have a Cardinality of its own since it cannot know how many instances of itself will be referenced by other models.

A **Composite** is similar to the SoI, but unlike the SoI, multiple Composites can exist in a model and they can be indefinitely nested inside each other. Furthermore, a Composite is intended to require knowledge about the systems that surround it to have practical value. If this is not the case, and if it's non-trivial, then the Composite could likely be turned into a SoI/SoS-model and thus be made reusable.

Finally, an **Atomic System (AS)** is both the smallest structural component and the largest behavioral component in ACS. Internally, an AS is defined in terms of explicit communicative behavior rather than additional sub-systems (see Sections 2.1.2 and 2.3). At the conceptual level, an AS represents a process, machine, entity, group, or similar, that the SoS designer does not desire to model in any further structural detail. While this definition might seem vague, it is also one of the key concepts that allow ACS to model a SoS at any abstraction level, anywhere from low-level hardware components to high-level data-centers.

2.1.1.2 Links

A **Link** is an abstraction of a communication medium going between the systems connected by the link. The label adjacent to each link hub simply displays the link's "*Name*", whereas each link connection displays a label with that connection's "*Cardinality*". The cardinality on a link is similar to the cardinality on a system. It states the number of system instances that will be involved in communication over that link, as well as what types of communication are possible.

Furthermore, link connections have a "*Directionality*". This property is visualized by the number of arrows on a link connection. While only two types of directionalities are exemplified in Figure 2.1, there are three types in total with the following effects:

- **Non-directed (Line)**: Instances of the connected system are only allowed to respond to communication requests made over the link by other system instances.
- **Mono-directed (arrow)**: Instances of the connected system are allowed to both respond to and initiate communication with instances of other system over the link. It is, however, not allowed to communicate with other instances of its own type.
- **Bi-directed (double-arrow)**: Same as mono-directed, however, the connected system will also communicate with other instances of its own type (the number of which is determined by the link connection's own cardinality). This requires that the system's cardinality is greater than the Link's cardinality since there would not

be enough other instance to communicate with otherwise.

Combined, directionalities state which systems are allowed to initiate communication over a link (and whether communication with instances of the same system type is allowed), and cardinalities state how many instances of each type of receiver-system will be communicated with. Figure 2.2 shows a few example combinations of system-cardinalities, link-cardinalities, and directionalities.

Looking at Figure 2.1 again, a single Server instance can initiate communication with 1 DbReference instance. A single Coordinator instance can initiate communication with 2 Sector instances, whereas a single Sector instance can initiate communication with 1 Coordinator instance.

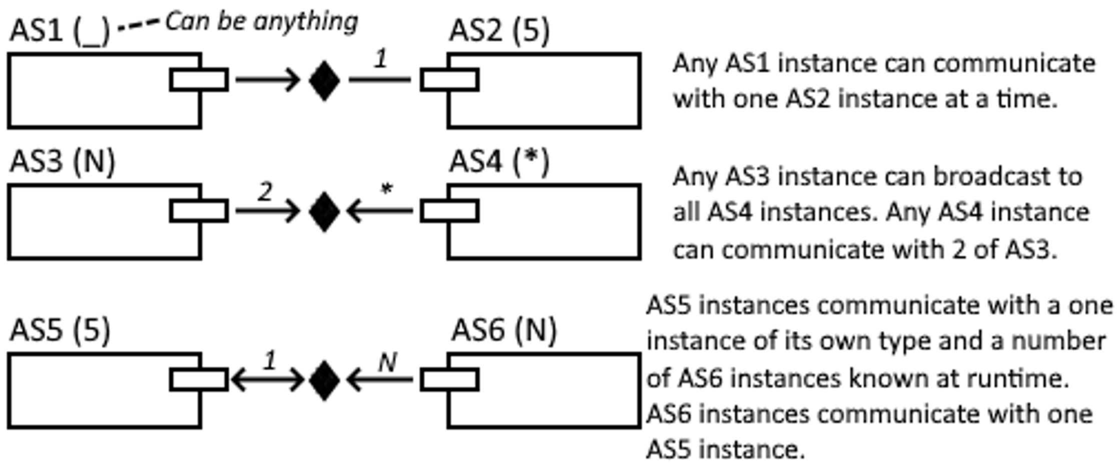

AS1 (_)
Can be anything
AS2 (5)
1
Any AS1 instance can communicate with one AS2 instance at a time.
AS3 (N)
AS4 (*)
2
*
Any AS3 instance can broadcast to all AS4 instances. Any AS4 instance can communicate with 2 of AS3.
AS5 (5)
AS6 (N)
1
N
AS5 instances communicate with a one instance of its own type and a number of AS6 instances known at runtime. AS6 instances communicate with one AS5 instance.

Figure 2.2: This figure shows some example configurations of systems and Links. Note that since AS2 cannot initiate communication with AS1, AS1's cardinalities (both on link connection and system) have no effect.

2.1.2 Controller- and Event-Diagrams

As explained above, only ASs have explicit behavior, meaning that the *behavior* of a Composite or SoI/Reference is actually a product of the interactions between the ASs within these. Behavior is event-driven and explicitly modeled using state machines. Each AS possesses a single “Controller Diagram” and one or more “Event diagrams” as illustrated in Figure 2.3. The Controller Diagram describes which Events on an AS are available in which states, whereas the Event Diagram describes how the AS communicates with other ASs’. Furthermore, when a controller transition is traversed, the state-machine/Event associated with that transition is “invoked”.

Communication is modeled by having one AS invoke Events on other ASs, which happens in the Event Diagram. An example of this can be seen in the bottom-left diagram of Figure 2.3, which describes that the "*NotifyClients*"-Event invokes the "*AcceptNotification*"-Event on all *Client*-instances. An invoked Event can then invoke even more Events on additional ASs, thus allowing ACS to model arbitrary chains of communication across any number of ASs.

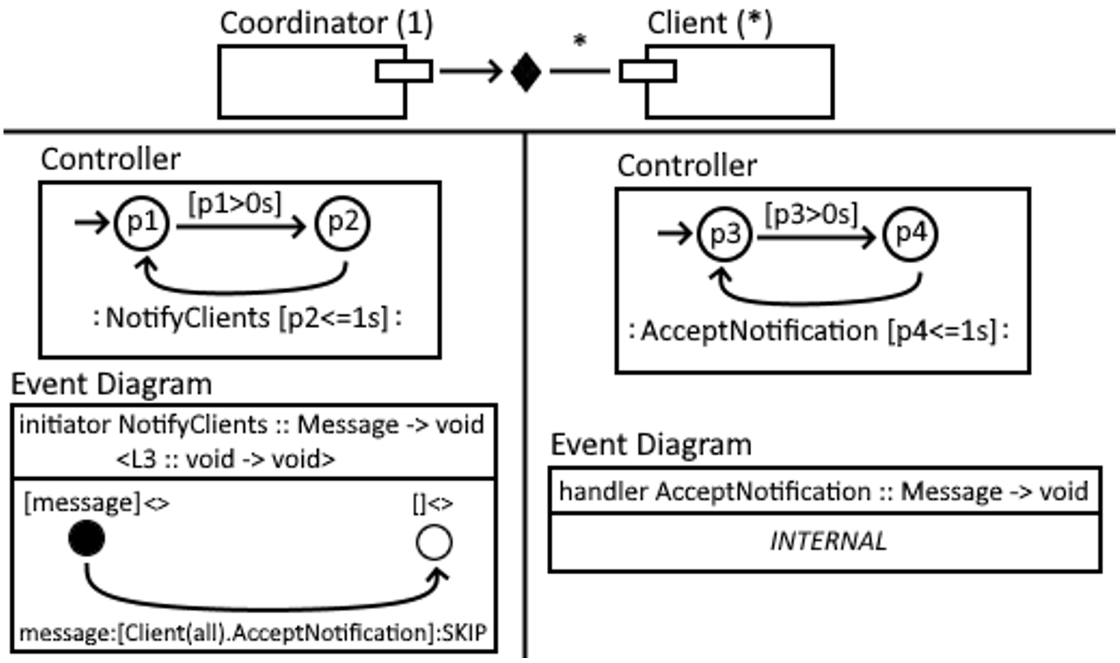

Coordinator (1)
Client (*)
*
Controller
[p1>0s]
p1
p2
: NotifyClients [p2<=1s] :
Controller
[p3>0s]
p3
p4
: AcceptNotification [p4<=1s] :
Event Diagram
initiator NotifyClients :: Message -> void
<L3 :: void -> void>
[message]<>
[]<>
message:[Client(all).AcceptNotification]:SKIP
Event Diagram
handler AcceptNotification :: Message -> void
INTERNAL

Figure 2.3: A simplified version of the behavior of and interaction between the Client and Coordinator systems from Figure 2.1, but assuming only these systems are present.

2.1.2.1 Event Declarations and Invocations

Of course, some Events must be able to self-invoke and not by request from another Event, else there would be no way to start communication in the first place. For this purpose, ACS supports 4 different types of Events. Figure 2.4 introduces these and shows: 1) whether an event type can be used on transitions in a Controller Diagram, Event Diagram, or both, and 2) which types of Events is used to invoke which other types of Events.

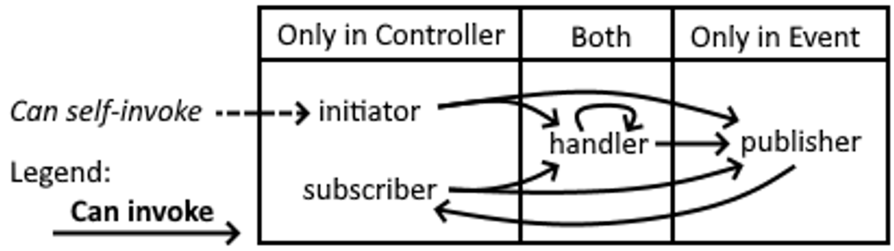


Figure 2.4: This table states: 1) which Events that can be used on the transitions of which Diagrams (see column titles), and 2) which Event types invoke each other.

An “**`Initiator`**” is the only Event type that can invoke without being requested to do so by another Event, and its purpose is thus to start chains of communication. Unlike all other Event types, which must be invoked by some other Event, the Initiator is invoked by an internal/environmental factor such as a human pressing a button or a timer outside the scope of the model. Such factors can be expressed as timing-constraints (see Section 2.1.2.2).

A “**`Handler`**” can either *continue* or *terminate* a chain of communication and is invoked when requested by another event. A Handler can be “*internal*”, in which case it has no associated state-machine, meaning it has almost no communicative behavior; only internal behavior which ACS does not model. Thus, an internal Handler terminates a chain of communication.

In Figure 2.3, the Controller Diagram of *Coordinator* states that after a non-zero duration of time has passed, the Initiator “*NotifyClients*” may be invoked, the Event Diagram of which then invokes the internal “AcceptNotification”-Handler from *Client*. The Controller Diagram for *Client* similarly states that “*AcceptNotification*” can be invoked at any time after a non-zero duration of time has passed.

The reader might notice that the two Event Diagrams in Figure 2.3 state the following in the top-box: “`Message -> void`”. This defines the input and output *data* types of the Event, meaning both Events take a `Message` object as input and produces nothing (`void`) as output.

The Event Diagram “NotifyClients” also specifies the following: “`<L3 :: void -> void>`”, which defines the Link as well as input and output **`Token`** types for the Event. This means that all communication happens over the Link `L3` and that the Event neither inputs nor outputs any tokens (indicated by (`void`)). The reader should refer to Section 2.3 for more information about tokens, but roughly stated, tokens are variables (with types) for remembering connections with other system instances (informally analogous to cookies in modern browsers).

The “**`Publisher`**” and “**`Subscriber`**” Event types (not included in the example above) are inspired by the publish-subscribe approach to communication. A Subscriber is defined to listen for data that is published on a specific Publisher, so when an Event invokes that Publisher, all Subscribers (from any number of ASs) that listen on that Publisher will react to the published data. The subscribing systems must be connected to the publishing system by a Link. Like Handlers and Initiators, Publishers and Subscribers have types, but unlike those, a Publisher only has input types and Subscribers only have output types (but take the input types of their Publisher). Additionally, a Subscriber can be internal like a Handler can.

2.1.2.2 Controller Diagram

Whereas Events represent the actions that an AS supports, the Controller further defines in which order and within which time intervals these Events are allowed. In any given Controller state, the available Events are determined by the available transitions which are annotated with Events. Thus, the set of all possible (non-simple) paths through the Controller's transitions (starting from the start state) also correspond to the set of all possible Event orders.

All transitions in a Controller are annotated with a time constraint surrounded by square brackets. For example, the time constraint "`[p2<=1s]`" (see Figure 2.3) states that the transition must finish within one second of last entering the "*reference state*", which is "`p2`". The time unit can range from nanoseconds to days. The reference state of a time constraint does not have to be the state that the transition leaves, but can also be an earlier state (which must have been visited beforehand). Conversely, the time constraint "`[p1>0s]`" requires that more than zero seconds have passed before the transition can be traversed.

While the examples does not make use of it, it is possible to put input and output "token variables" before and after the left and right colons ("`:`") on Controller transitions (e.g. "`tok_in : EV [p<=1s] : tok_out`"). See Section 2.3 for more information about token variables and how to use them.

2.1.2.3 Event Diagram

An Event Diagram is made of two parts: The "*Event declaration*" (at the top of the diagram) and the "*Communicator*" (the state-machine at the bottom of the diagram). Section 2.1.2.1 already explains the declaration part, so this section primarily focuses on the Communicator part. During the invocation of an Event, and during the traversal of a Controller transition, the Communicator of the invoked Event is executed fully. Thus, the time it takes to traverse a Controller transition is the same as the time it takes to fully execute the Event's Communicator.

A Communicator transition can invoke any one of Handler-Event, Publisher-Event, "**`Sub-Machine`**", or "**`Function`**". Generally, the invocation annotated on a Communicator transition consists of a list of inputs, what to invoke, and a list of outputs, however, in the case of invoking a Publisher, there is no output. A Sub-Machine is simply an additional state-machine in the Communicator and a Function is a placeholder for business/internal logic (to support code injection during code generation). Both have explicit input and output data types.

Since all "invokable components" have explicit input and output data types, it is possible to construct a complete model of the data flow within entire chains of communication and to check a model for type correctness as explained in Sections 2.1.3 and 2.4. However, data flow is "abstract" in the sense that no communication is ever made and no concrete data/values are ever created. ACS only does verification given the constraints.

The inputs to an invocation come from "*Data Variables*", an example of which is "`message`" that is defined as the name of the input of type "`Message`" to the "*NotifyClients*" Initiator from Figure 2.3. The name of this input is defined in the square brackets on the start state of the "*NotifyClients*" Event Diagram. The angle brackets are used for token variables, like before, see Section 2.3 for more information about these.

The invocation made within "*NotifyClients*" thus broadcasts the "`message`" to all *Client* instances (on the "*AcceptNotification*"-Handler) and does not wait for any response from these, which is indicated by the **`SKIP`** keyword given as the output. Due to the wide variety of valid invocation configurations – each of which represent different communication mechanisms – all further explanation of how invocations work is deferred to Section 2.3.

2.1.3 Type Documents

As explained in Section 2.1.2, all invocations are typed, which allows ACS to track data flow across all systems and to type-check all communication. Types are defined using a domain-specific language in a "Type Document". Every SoI and AS have one Type Document. The types defined on the SoI are globally visible (called *global types*) whereas those on ASs are locally visible (called *scoped types*), however, scoped types can "propagate" to other ASs through communication. Thus, only the ASs that receive a type through communication actually gets access to it – called the type's "scope" – which imposes additional structure on a model that can be used for verification.

The type system contains three "*fundamental types*": **`number`**, **`character`**, and **`boolean`**. Whereas booleans are restricted to true and false, numbers can have any numerical value with any decimal precision and characters can be any character/symbol. ACS further has four "*type constructs*": **`record`**, **`array`**, **`nullable`**, and **`enumeration`**. Whereas enumeration is used to make a new type with a finite value-space consisting of names, the remaining constructs are used to make new types from existing ones.

The running example from Figure 2.3 mentions the `Message` type, but did not provide a definition. To expand on the example, let us imagine that `Message` should represent some string containing a message, an priority (urgent, important, notice), and an optional deadline-timestamp. Futher, imagine also that the message string can only have English letters & spaces and be up to 100 symbols long, and that the timestamp must be a non-negative integer. These restrictions are expressed as "*constraints*". Listing 1 shows how this type would be modeled in ACS.

```
1      String = char[] { }                          // An array of characters
2      Priority = { URGENT, IMPORTANT, NOTICE } { } // An enumeration
3
4      Message = (            // A record
5          String message,    // A field with type "String"
6          Priority priority, // A field with type "Priority"
7          num? deadline      // A field with type "nullable number"
8      ) {
9          .message._elem."a-zA-Z ",  // Constraint array element values
10         .message.[[1;100]],        // Constraint array size
11         .deadline.[0;+inf],        // Constraint number-value
12         .deadline.(0)              // Decimal precision = '0' (integer)
13     }
```

Listing 1: The definition of the Message type from Figure 2.3 and related types.

An ACS implementation can additionally pre-define a range of global, common types such as different sizes of integers, positive integers, and strings (as opposed to declaring strings locally, as in Listing 1), so types don't need to be defined for each new model. Listing 2 shows some example pre-defined types that could be used.

```
uint   = num { .[0;+inf], .(0) }      int   = num { .(0) }
uint8  = num { .[0;2^8-1], .(0) }     int8  = num { .[-2^7;2^7-1], .(0) }
uint16 = num { .[0;2^16-1], .(0) }    int16 = num { .[-2^15;2^15-1], .(0) }

String = char[] { }
```

Listing 2: List of example predefined types.

There exists a third kind of type apart from global and scoped: "*anonymous types*", which are types without a declared name. An example of this is "`num?`" on line 7 in Listing 1. Another example would be to change the input type of "*NotifyClients*" in Figure 2.3 from `Message` to `Message[]`, in which case "array of messages" is a valid, existing type, but it is not referenced *purely* by name, thus making it anonymous.

ACS has the concept of "*type substitution*" which allows one type to be used in place of another when the first type's value-space is a subset of the second type's value-space. That is, the same principle from functions in programming where an *actual parameter* must simply be "compatible with" a *formal parameter*. In ACS, this is when the actual parameter has the same structure as, and is no-less constrained than, the formal parameter. For example, `uint8` on line 1 of Listing 2 can be used as a `uint16` since the value-space of `uint8` is a subset of that of `uint16`. Conversely, `uint8` cannot be used as `int8`, `int16`, nor `int`.

An important difference between scoped types and the other two kinds of types (global & anonymous) is that whereas scoped types can only be used in place of types that they "extend" or "specialize", global & anonymous types must simply have the same structure (the comparison of constraints remain the same). For example, imagine two types "`Message1`" and "`Message2`", which are both identical to "`Message`" from Listing 1. If these two types were global (or even anonymous), then they could be used in place of each other, whereas if one or both of "`Message1`" and "`Message2`" were scoped, then they would be considered different for the purposes of enforcing the type structure.

2.2 The Structural Model

The structural model does not represent one concrete SoS, but rather a template that can get instantiated one or more concrete SoSs (called "instances of the structural model" or "SoS instances") that adhere to the template. The only structural distinction between instances of the same structural model is the number of system instances in the SoS instance, which is determined by system Cardinalities. Cardinalities are also used to vary the number of system instances that will communicate with each other.

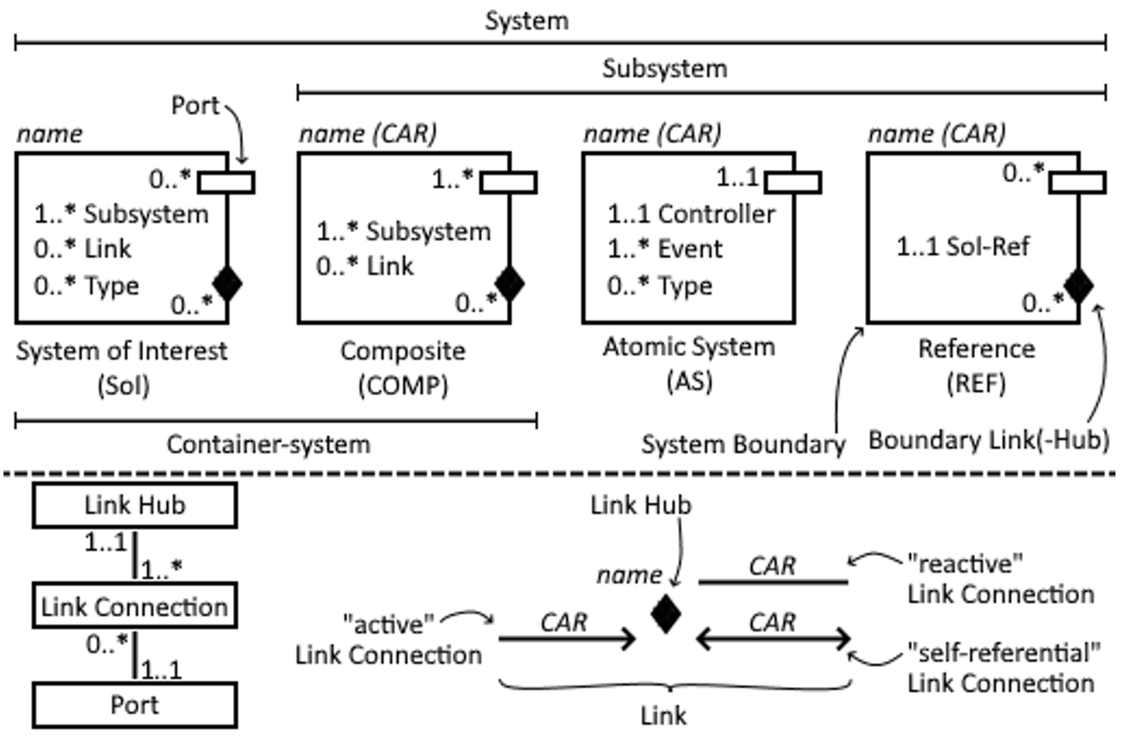


Figure 2.5: The diagrammatic syntax for ACS' structural components. "CAR" means "Cardinality". Figure 2.9 form Section 2.2.2.2 shows one additional structural component which is too intricate to present here.

Figure 2.5 describes the syntax/structure of all but one (see image text) of ACS' structural components, as well as the different categories of Systems, Links, and Link Connections. Ranges (e.g. `0..*` and `1..*`) denote the required/allowed number of each component within systems and in relation to the structure of Link Connections. The components "Controller" and "Event" are behavioral (see Section 2.3) and the "Type" component relates to the data layer (see Section 2.4).

This section explains the design choices related to the structural model and the rules at its foundation. It is recommended to read Section 2.1.1 first to get a general understanding of the structural components.

2.2.1 Cardinalities

As mentioned in Section 2.1.1.1, a Cardinality in ACS is a measure that expresses "an amount of system instances" and "whether these instances can be uniquely addressed". Table 2.1 shows the key properties of each of the three classes of Cardinalities, namely, positive integers (from the set $\mathbb{Z}^+$), the value ‘N’, and the value ‘*’ (i.e. $CAR = \mathbb{Z}^+ \cup \{\texttt{N}, *\}$).

Property	$\mathbb{Z}^+$	‘N’	‘*’
(1) Exact number known *before* SoS instantiation (design-time)	✓		
(2) Exact number known *after* SoS instantiation (run-time)	✓	✓	
(3) Uniquely addressable instances	✓	✓	

Table 2.1: An overview of the key properties of the three classes of Cardinalities.

The reason for using this exact set of Cardinality classes stems from the desire to 1) allow the modeling of as many communication mechanisms as possible with as few model constructs as possible, and 2) to visually express the structure of communication. The Cardinalities are also inspired by various types of communication that modern computer networks use, such as client-server communication, peer-to-peer communication, and various casting mechanisms (e.g., unicast, multicast, broadcast, anycast).

Section 2.3 describes how Cardinalities help model (and restrict the usage of) various communication mechanisms. Section 2.2.2 explains how the duality between “*System Cardinalities*” and “*Link Cardinalities*” allow visually expressing the “structure of communication”. This separation lets the designer separately reason about the number/amount of system instances in a SoS, and the number/amount of system instances involved in communication.

Another reason for the three Cardinalities and their properties is that the number of system instances (in a SoS or during communication) can be known: 1) at design-time (thus, a Z-Cardinality), 2) at run-time (N-Cardinality), or 3) never at all (*-Cardinality), and that whenever the exact number is known, there must be a way to uniquely identify each instance.

Whereas Z-Cardinalities could be used for central systems such as servers, the *-Cardinality (Star-Cardinality) could be used for ad-hoc systems such as clients for which you cannot have or do not need individual identities. The N-Cardinality could be used in a Peer-to-Peer network, where the number of peers may vary, or any system where the number of receivers of some message varies but each peer may need an individual identity.

2.2.2 The Structural Components

This section explains Ports and Links on their own before explaining the four kinds of system components, even though Links and Ports make up parts of these. This order aims at explaining the most fundamental elements first.

Finally, the section explains why Types are declared specifically on SoIs and ASs, and also explains the top-most "ACS Model"-component that contains the main SoS as well as all SoSs that can be referenced.

2.2.2.1 Links, Link-Cardinalities, and Ports

Whereas **`Ports`** represent interfaces that perform communication, **`Links`** represent mediums that facilitate communication. As seen in Figure 2.5, a Link consists of a **`Link Hub`** which must have one or more **`Link Connections`**. Each Link Connection connects to one Port and states a "*Link Cardinality*". Whereas Section 2.1.1.2 explains how to read and understand Links, this section explains why we chose this Link design among alternatives.

Links are split into the Link Hub and Link Connections to better visualize the "structure" of communication. Communication is more than just independent requests and responses. One system may request data from two other systems and send some computed result to a fourth system. The Hub & Connections design allows one Link to visualize that all four systems are part of a larger of communication and data flow. An alternative design would have been to allow Ports directly connect with a single line/edge, but this design does not properly convey this "communication structure". Figure 2.6 demonstrates this.

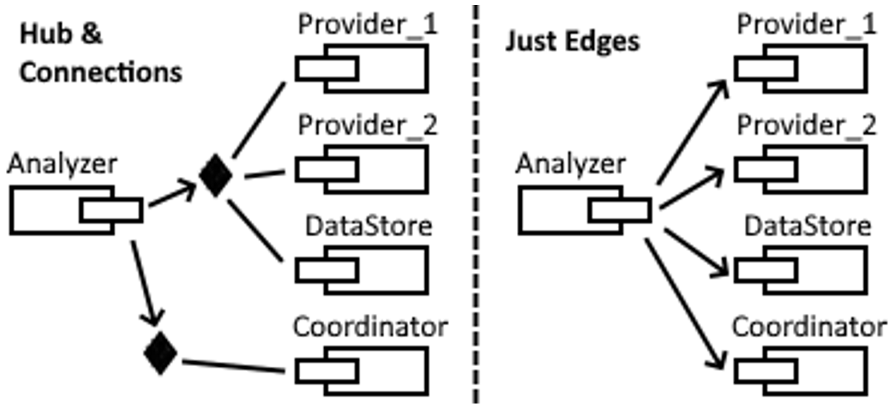


Figure 2.6: Comparison of two Link designs. The Hub & Connections design better conveys the kind of communication the Analyzer makes with the other systems.

Figure 2.7 presents the three Link Connection types. The "**`reactive`**" type only allows a system to react/respond to communication requests made over the Link. Next, the "**`active`**" type allows a system to both initiate and respond to communication over the Link. Finally, the "**`reflexive`**" type allows a system to initiate communication with other instances of its own type, as well as all other systems connected to the Link (note that `active` does not allow this self-communication). Section 2.1.1 used different names for ease of understanding.

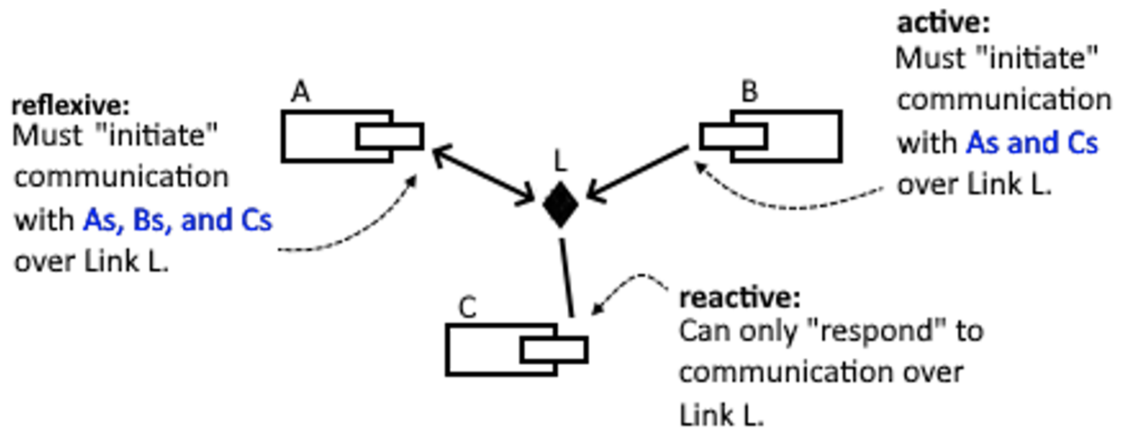

active:
Must "initiate" communication with As and Cs over Link L.
reflexive:
Must "initiate" communication with As, Bs, and Cs over Link L.
A
B
L
C
reactive:
Can only "respond" to communication over Link L.

Figure 2.7: The effects of the three types of Link Connections. There can be multiple `A`, `B`, and `C` instances, which is why they are plural in the blue text.

Link Connection types also serve as constraints on the behavior of systems, which can be used to validate whether systems "behave correctly" according to the structure. The semantic rules below are used for verifying Links. Additional Link rules follow in this section and Section 2.3.

Rule 1 *Only one Link Connection can exist between any pair of Link Hub and Port/System.*

Rule 2 *A Link must allow at least two system instances to communicate with each other, possibly of the same system type (e.g.* `A` *from Figure 2.7). This requires "at least one* `reflexive` *Link Connection" or "two or more Link Connections where at least one is* `active`*".*

Rule 3 `active` *and* `reflexive` *Link Connections requires a system (e.g.* `A` *and* `B` *in Figure 2.7) to initiate communication over a given Link. Communication must happen with all systems that the initiating system is allowed to communicate with.*
Exception: *Boundary Links on a SoI only require at least one* `active` *or* `reflexive` *Link Connection, since the remaining system(s) would be connected to a Reference to that SoI.*

ACS assumes there is only ever one system instance that initiates communication over a Link, and with that in mind, Figure 2.8 demonstrates how to read Link Cardinalities, which depends on the perspective of the initiating system and its associated Link Connection type. Section 2.3 explains the actual types of communication allowed by each Cardinality. Note that the **`N`** cardinality allows the number of involved system instances to change every time any system initiates communication over the Link.

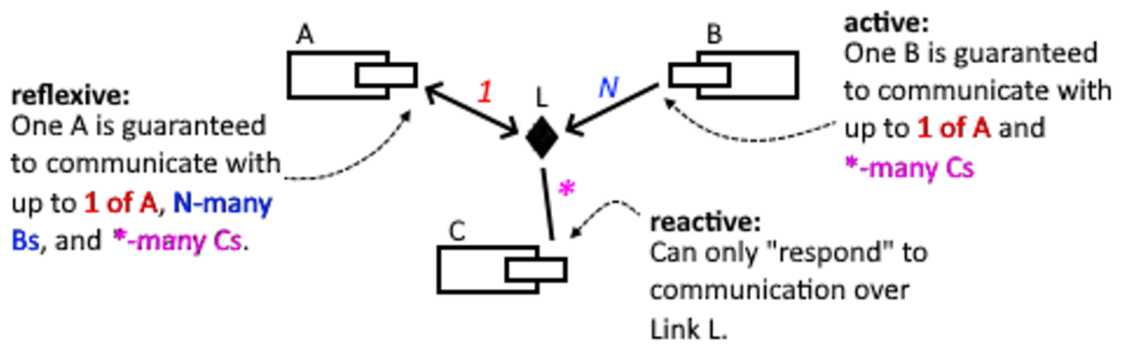

A
B
C
L
1
N
*
reflexive:
One A is guaranteed to communicate with up to 1 of A, N-many Bs, and *-many Cs.
active:
One B is guaranteed to communicate with up to 1 of A and *-many Cs
reactive:
Can only "respond" to communication over Link L.

Figure 2.8: Explains how Link Cardinalities express how many instances of each system an initiating system should communicate with. Color coding is used to show how Cardinalities are read from the perspective of each connected system.

When Figure 2.8 says "guaranteed to communicate with up to X instances", this means that any protocol related to any communication initiated over the link `L` must have at least one "execution path" that makes use of all available instances (across all connected systems). Rule 4 below describes this in more accurate terminology.

Rule 4 *All Events with communicative behavior must have at least one path in its Event Diagram that makes use of all $\mathbb{Z}^+$, **`N`**-many, or `*`-many system instances allowed by all Link Connections on the Link that the Event communicate over.*

Thus, with the above rule in mind, the combination of the systems that are connected, the types of connections, and the Cardinalities on Link Connections both constrains and give insight into the kind of communication that happen over a given Link.

2.2.2.2 Systems, System-Cardinalities, and Interfaces

Systems are abstractions that give structure to a SoS and behavior to Ports. All systems are surrounded by a box, called a "**`System Boundary`**" (see Figure 2.5), on which "**`Ports`**" and "**`Boundary Links`**" reside. **`Links`** can also be placed freely inside Container Systems (called "Free Links"). The System Boundary serves to isolate a system's insides from the outside world (and vice versa), whereas the Port and Boundary Link serves to allow communication across said System Boundary.

Ports and Interfaces

Section 2.1.1.1 already describes the general use and structure of the four kinds of systems, including how Container Systems support arbitrarily deep nesting. These structural and nesting properties are expressed diagrammatically in Figure 2.5 above. Furthermore, since only Atomic Systems have explicit behavior, all Container systems (and indirectly References) must have an AS inside to actually provide behavior for them. Figure 2.9 shows how this is modeled using “aliasing” where a Container System Port is set as an alias for an AS Port.

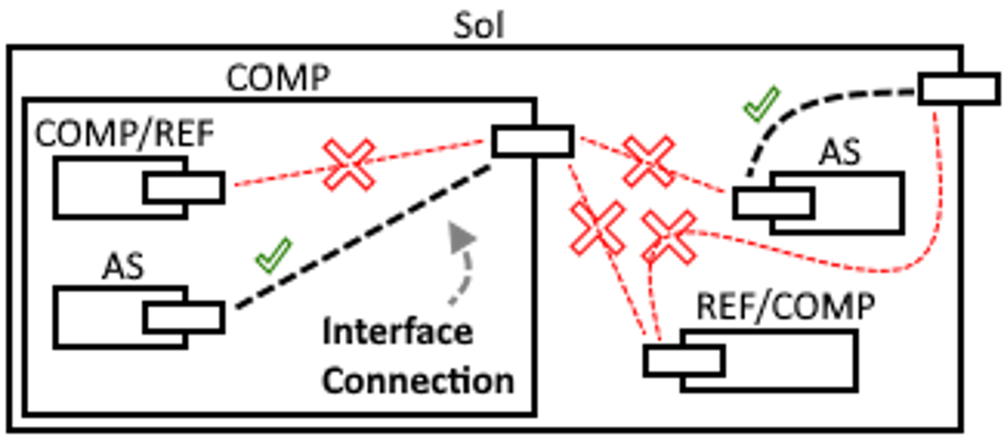


Figure 2.9: The figure demonstrates how Interface Connections can be used to mark an AS as an interface of a Container system. The crossed connections are not valid.

The dashed line is called an “`Interface Connection`”, and the ASs that have such a connection to one of their parent Container’s Ports are called “the interfaces of the container”. The rules for interface connections depicted in the figure above are described textually below.

Rule 5 *All Ports on all Container Systems must have an Interface Connection to an AS Port.*

Rule 6 *An Interface Connection can only exist between an AS Port and a Container System Port, where the Container System is the direct parent of the AS.*

Boundary Links

The nested structure of systems, as well as System Boundaries and Container System interfaces add some additional constraints to Free Links (and Boundary Links). These constraints are expressed textually in the rules below and most are visualized in Figure 2.10.

Rule 7 *A Link Connection cannot cross a System Boundary.*

Rule 8 *Boundary Links can have Link Connections on both sides of one System Boundary.*

Rule 9 *A Link Connection cannot be connected to a Port on its parent Container System, nor on its Link Hub's parent Container System.*

Rule 10 *Only an AS with an Interface Connection can have an* `active` *or* `reflexive` *Link Connection to a Boundary Link.*

Rule 11 *The interface system and the Boundary Link Hub must have the same parent Container System.*

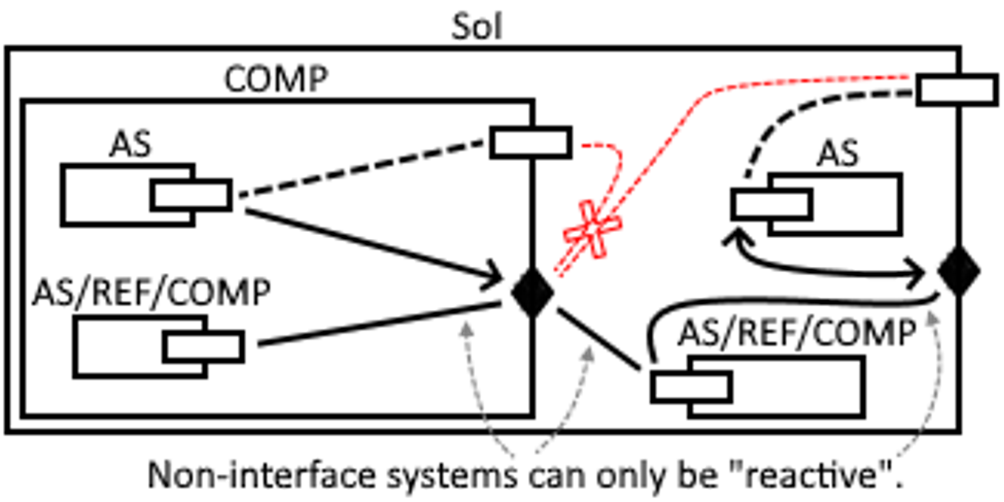


Figure 2.10: The figure demonstrates how only interfaces of a Container System can initiate communication over Boundary Links on said Container. It also shows how link connections cannot connect certain Link Hubs and Ports.

Since System Boundaries are meant to isolate the inside and outside of a system, Link Connections cannot cross System Boundaries. The reason for including the Boundary Link, however, is two-fold.

Firstly, since a Reference knows nothing about its surrounding environment, it cannot initiate communication directly with any outside systems. Therefore, the only way for a Reference to initiate communication "outwards" is by using a publish-subscribe mechanism, where systems outside the Reference subscribe to the messages published from inside it. The subscribers must thus have a Link Connection to a Boundary Link on the Reference, over which the messages will be published. See Section 2.3 on communication mechanisms.

Secondly, when an interface system initiates communication with an inside and outside system at the same time, only a Boundary Link can connect all systems in a single Link. The only alternative would be to have a Free Link connect the interface and inside system, and another Free Link outside to connect the interface and outside system. However, separating into two links gives the visual impression of separate communication, which goes against the core principles of Links. Figure 2.11 shows these two alternatives.

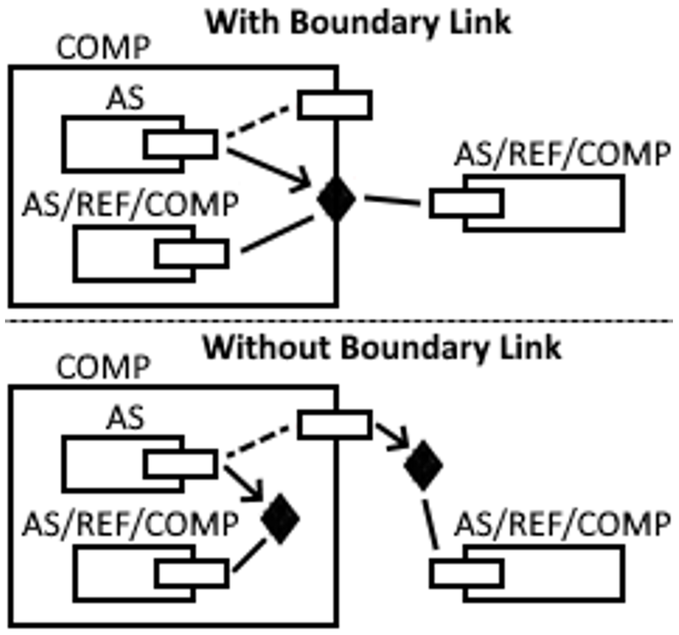


Figure 2.11: Only Boundary Links can properly model communication that spans two nesting layers. The alternative with Free Links gives the wrong impression that communication with the inside and outside systems are separate.

System Cardinalities

All systems have a "*System Cardinality*" apart from the SoI, which is assumed to have only one instance since it never interacts directly with its surroundings (nor with other instances of itself). Of course, References to a SoI can make multiple instances of it, but this does not affect how the SoI can interact with the outside. The Cardinality of Container Systems also affects the total number of instances of their child systems. For example, for each of the `N`-many instances of `COMP` in Figure 2.12, there will be 5 instances of `AS1`.

As seen in Figure 2.12, when ACS verifies communication with a Port on a Composite (and also a Reference), one instance of that Port's interface AS is selected non-deterministically for communication. This is because 1) outside systems are unaware of the insides of Composites and References, and 2) allowing an outside system to freely choose the interface instance would cause ASs and non-ASs to have different communication syntax (see Section 2.3).

This non-deterministic approach works well since ACS uses formal verification to determine whether communication works, rather than simulating it or other methods. In a real implementation (or simulation), however, this selection would be a question of network routing or similar.

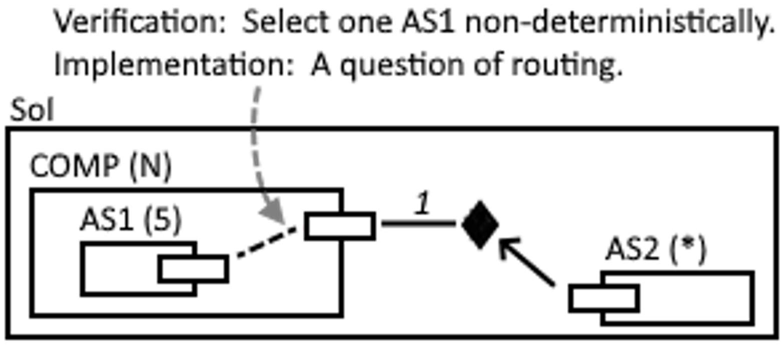

Verification: Select one AS1 non-deterministically.
Implementation: A question of routing.
SoI
COMP (N)
AS1 (5)
1
AS2 (*)

Figure 2.12: How to select which interface AS instance to use for communication (in a Composite or Reference) when multiple instances are available.

Note that even if both a Link Connection and its Connected System have an **N**-Cardinality, this does *not* mean that all **N** system instances in the SoS are involved in communication. Instead, it means that N_{Link} out of N_{System} instances are involved, where "$0 \leq N_{Link} \leq N_{System}$".

The System Cardinality does not affect communication/links apart from restricting the possible Link Cardinalities. Table 2.2 shows which Link Cardinalities are allowed with which System Cardinalities (and the `reflexive` Link Connection type in one special case).

Rule 12 *The Link Cardinality must be compatible with the System Cardinality (see Table 2.2). The Link Connection type affects the Z-Z case.*

	Z_{SYS}	$\mathtt{N}_{SYS}$	$*_{SYS}$
Z_{LNK}	`reflexive`: $Z_{LNK} < Z_{SYS}$ otherwise: $Z_{LNK} \leq Z_{SYS}$	✓	
$\mathtt{N}_{LNK}$	✓	✓	
$*_{LNK}$	✓	✓	✓

Table 2.2: The conditions under which a Link (LNK) Cardinality can be used with a System (SYS) Cardinality.

The `reflexive` Link Connection type requires there to be at least one more system instance than there are instances involved in communication since one extra instance is needed to initiate communication with the other instances of the same system type.

The `*`-Link-Cardinality can always be used since this simply allows broadcasting messages to any system instances that are listening for said broadcast (see Section 2.3). Conversely, since the Z- and **N**-Link-Cardinalities require that all instances involved in communication are uniquely addressable, these become incompatible with the `*`-System-Cardinality, which does not allow unique addressing.

Different System Boundary Layouts

The AS represents a single behavioral unit – where this behavior is modeled by the Controller and Events – and thus only has one Port. The AS further has no Boundary Links since these must connect to systems on both sides of the Boundary and there are no systems nested inside an AS.

Container Systems can have any number of systems inside, and as such, they can also have any number of interfaces (see Figure 2.9) and thus any number of Ports. However, whereas the SoI can have zero Ports, a Composite must have at least one Port.

A Composite is by definition a system that requires knowledge about its surrounding environment to have practical value, and as such, it must communicate (both ways) with its environment, which requires at least one Port. Conversely, the SoI is by definition a system that does not know anything about its surrounding environment and is thus free to decide whether it will even allow communication in/out of it. Therefore, the SoI does not require having any Ports or Boundary Links, even though at least one of either is necessary for any reference to that SoI to be usable.

Data Types on the SoI and ASs

Figure 2.5 shows how Data Types are defined specifically on the SoI and on ASs. As Section 2.1.3 also mentions, types defined on the SoI are "*global types*", whereas those on ASs are "*scoped types*". Whereas global types can always be used anywhere, scoped types can only be used by ASs that (indirectly) receive the type through communication with the AS that defined the scoped type. Section 2.4 explains these principles in more detail.

Since every SoS (and thus SoI) can define its own set of global types, we simply define global types in a document alongside the SoI, whereas scoped types are defined in a document alongside the AS on which they are defined.

2.2.2.3 The ACS Model Component

Since an ACS model may contain not only the main SoS but also all the SoSs that the main SoS (indirectly) depends on, there is a hidden "**`ACS Model`**" or just "Model" component on top of all the SoS models. The ACS Model component is simply a list of pairs between a SoI component and a list of dependencies in the form of SoI names. The ACS_Model component is described mathematically below and graphically in Figure 2.13. Rule 13 poses a constraint on References in a SoI.

$$\mathtt{ACS\_Model} = \overrightarrow{\left(\mathtt{SoI}\,,\ \overrightarrow{name_{SoI}}\right)}$$

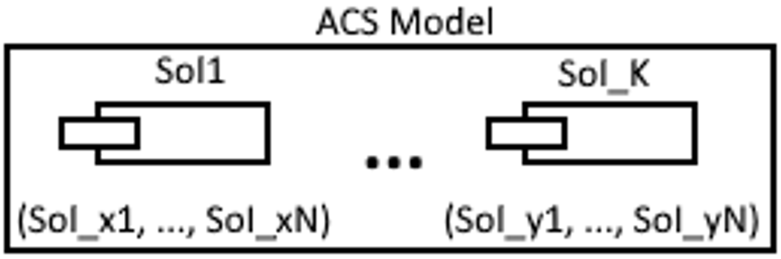


Figure 2.13: A depiction of the ACS Model component, which is a list of pairs between SoI components

Rule 13 *References in a SoI are only allowed to reference the SoIs whose name is in the SoI's dependency list.*

2.2.3 High-Level Features and Limitations

ACS' System and Link concepts are designed to allow the modeling – at any scale – of groups of units that communicate over some medium. The Systems/units could be anything from people to server computers, and Links/mediums could be anything from a speech to intercontinental networks. One weakness, however, is that SoSs where physical properties (such as location) influence communication are hard to model/approximate with ACS' "template structure" approach; especially if these properties are subject to frequent change.

Problem 8 from our 9th-semester project report [11, p.40] describes how there is a need for better methods for modeling interfaces to better manage and integrate possibly highly heterogeneous systems in large numbers. ACS tackles this using Ports and Links, since these can represent very heterogeneous concepts, and since all Ports/interfaces are readily available in the Structure Diagram and Links further describe how Ports relate to each other.

In extension of generalized representations of interfaces and mediums, Problem 2 [11, p.36] identifies a need for generalized communication mechanisms that are independent of the concrete properties of the interfaces and mediums that Ports and Links represent. Sections 2.3.4 and 2.4.4 describe how the Behavioral and Data models contribute to solving this problem.

2.2.4 Names and Qualified Names

All systems and links have a "*name*" for identification purposes. Names must be of the form: "`("_" | letter) ("_" | letter | digit)*`" and cannot be a keyword (see Table 2.3). The range of supported letters is left as an implementation detail (but should at least support English letters), since ACS will function correctly with any range of letters. Name length restrictions are left for implementation as well. No named object in an ACS model can share a name with its direct parent. This prevents certain bad naming patterns that make resolving ambiguous names (see below) very cumbersome.

`reactive`	`active`	`reflexive`	`initiator`	`handler`	`subscriber`
`publisher`	`internal`	`lockable`	`function`	`machine`	`lock`
`unlock`	`all`	`WAIT`	`SKIP`	`char`	`num`
`bool`	`void`	`_base`	`_elem`		

Table 2.3: The keywords in ACS. Taken from the abstract syntax presented in Chapter 3.

A "*qualified name*" is a sequence of DOT-separated (`'.'`) *names* consisting of the parent component name(s) and the name of some component itself. Qualified names have the form "`"."? name ("." name)*`", where *partially qualified names* start with a name, and *fully qualified names* start with a DOT ('`.`'). Qualified names are used for the purpose of disambiguation, and whereas fully qualified names are absolute, partially qualified names are compared to the end of the full name of all names of the required kind (Data Type, System, Link, etc.).

Table 2.4 shows an example where two types with ambiguous names can only be disambiguated by some of their qualified names. It also shows how a partial name and the full name can be almost identical (see the "Full" and "Almost full" columns), the only difference being the DOT at the start.

Full	**"Almost full" (Partial)**	**Partial**	**Name**
`.SoI1.AS1.TypeName`	`SoI1.AS1.TypeName`	`AS1.TypeName`	`TypeName`
`.SoI2.AS1.TypeName`	`SoI2.AS1.TypeName`	`AS1.TypeName`	`TypeName`

Table 2.4: Two SoIs have an AS called `AS1` that defines a type called `TypeName`. The full name or the almost full name would be needed to disambiguate these.

2.2.5 Changes since the old version

ACS initially discerned between "*System Counts*" and "*Link Multiplicities*" [13], which were subsequently unified into the singular "*Cardinality*" during the current development efforts. Counts and Multiplicities worked the same way as Cardinalities, but Link Multiplicities did not support the **`N`**-value. The unification not only simplifies the structural model since there are less concepts and exceptions to remember, but also allows ACS to model communication with a varying number of uniquely addressable system instances.

The `reflexive` Link Connection type is a new addition, and "directed" and "non-directed" Link Connections have been renamed to `active` and `reactive`. The only way to previously model "self communication" was to have both an `active` and a `reactive` Link Connection to the same system. However, when two Link Conenctions were connected to the same system over one Link, there was no good way to decide which Link Cardinality should be used if other systems could initiate communication over the Link as well.

Boundary Links are a new addition and were added for the reasons outlined in Section 2.2.2.2. The initial release of ACS can therefore not model communication that spans two Container systems. Boundary Links were added together with the publish-subscribe communication mechanism (see Section 2.3).

Renamed the **`SoS`** component into **`SoI`** to avoid confusion between the SoS concept and component. The SoI would previously contain all the SoI's it depended on inside itself.

Since this recursive structure of dependencies complicated both syntax and semantics, we added the ACS Model component to contain all SoIs in a list for simplicity and without the risk of duplication when multiple SoIs depend on the same SoI.

There were previously no clear semantics for how names and qualified names should be related to model objects, nor a clear distinction between names, partially qualified names, and fully qualified names.

Global Types is newly added to define general types that all systems should have access to. See Section 2.4 about the new type system and Section 2.4.5 about the old type system.

"Global References" have been removed, but we plan to re-introduce them again in the future. Global References reference SoS instances that exist independently of the currently modeled SoS, however, no semantics exist for Global References. Upon discussing how they should work, we realized that our current ideas would break system isolation in undesirable ways, and figuring out proper semantics could end up as its own project.

2.3 The Behavioral Model

The behavioral model is a template for all possible system behaviors akin to how the structural model is a template for all possible SoS structures. However, the behavioral components are not "instantiated" in the same sense as systems, but are instead properties of the ASs (and thus AS instances) on which they are defined. Figure 2.14 describes the structure and syntax of ACS' behavioral components (Figure 2.5 describes how these relate to Atomic Systems).

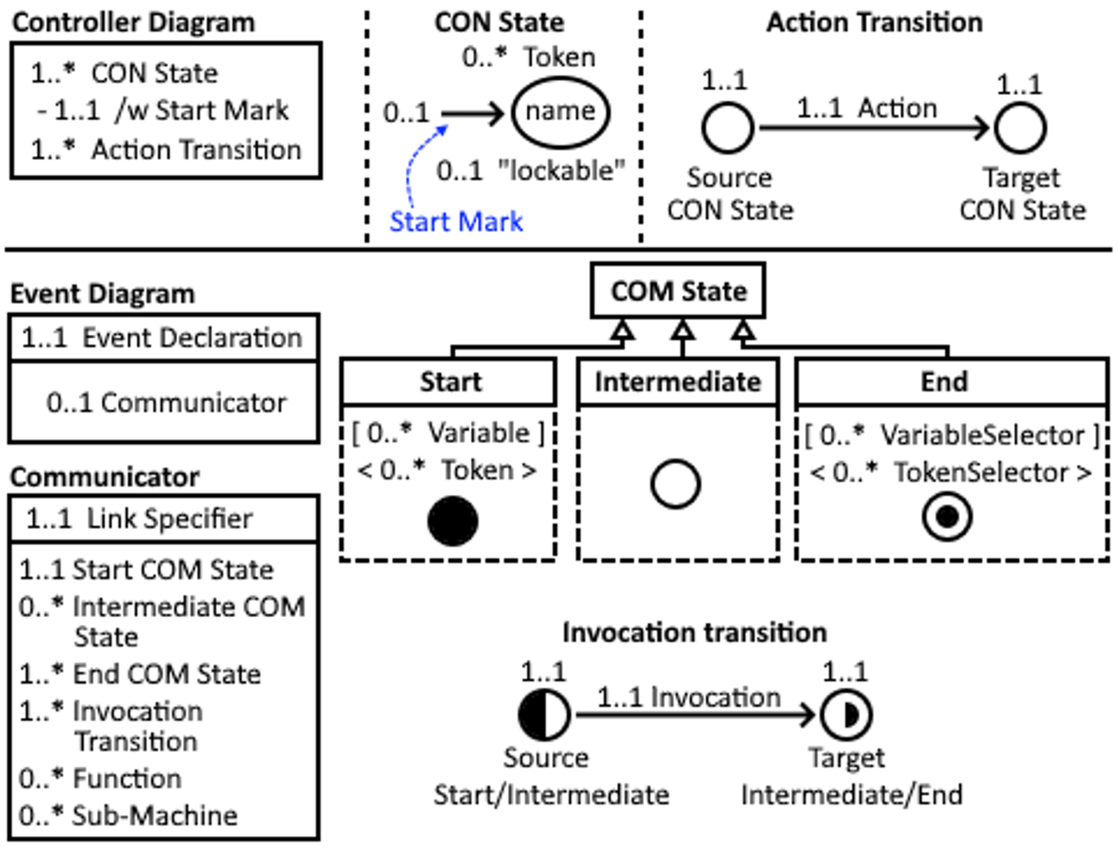


Figure 2.14: The structure and syntax of the behavioral components. Event Declaration, Link Specifier, Function, Sub-Machine, Invocation, Action, and the Selectors are (complex) textual elements and thus not included in the image.

Figure 2.14 shows the **`Controller Diagram`** and **`Event Diagram`**, as well as the components related to these. Most components are composed of other components, and the ranges (e.g. `0..1`) denote the required/allowed number of sub-components (e.g., Figure 2.14 shows how "`CON State`" may or may not have the `"lockable"` attribute). Whereas the behavioral diagrams are structurally very uniform (i.e. primarily state machines), the textual parts (such as Actions and Invocations) are very diverse. The latter are described throughout this section.

This section explains the design choices related to the behavioral model and its rules, as well as a lot of details that could not be covered in Section 2.1.2. It is, however, recommended to read Section 2.1.2 first to get a general understanding of the behavioral components.

2.3.1 Events and Data Flow

Events represent (possibly empty) collections of communicative behavior with some input and output, and which are given a human readable name. The **`Controller`** of an AS describes which Events may invoke when and in which order, and the **`Communicator`** describes which Events (if any) an Event invokes on other systems/ASs. The reason for making behavior event-driven is reminiscent to the reason for including functions in programming languages: to abstract away low level behavior under a high-level and human-readable (and self-explanatory) name.

The four Event types differ in structure and effects in ways that arise from their individual purposes and how they can be used. Table 2.5 below shows the structure of an Event Diagram depending on the Event type. Figure 2.4 on page 9 shows which diagrams can use which Events and which Events can invoke each other.

Event Diagram		
Event type	**Event Declaration**	**Communicator**
`Initiator`	**`initiator`** $name_{INI}$ `::` $\overrightarrow{DataType_{IN}} \rightarrow \overrightarrow{DataType_{OUT}}$	Required
`Handler`	**`handler`** $name_{HAN}$ `::` $\overrightarrow{DataType_{IN}} \rightarrow \overrightarrow{DataType_{OUT}}$	Optional
`Publisher`	**`publisher`** $name_{PUB}$ `::` $\overrightarrow{DataType_{IN}}$	None
`Subscriber`	**`subscriber`** $name_{SUB}$ `::` $qname_{PUB} \rightarrow \overrightarrow{DataType_{OUT}}$	Optional

Table 2.5: The structure of an Event Diagram depending on the Event type. The `qname` is a qualified name (see Section 2.2.4).

The inputs and outputs, as well as the requirement for a Communicator, vary depending on the type of Event. Events with no Communicator terminate a communication chain; including Publishers since they do not know their Subscribers and thus cannot get responses from them. An invoked Subscriber can then start its own communication chain, or can have no Communicator and immediately terminate the chain. An Event can spawn multiple communication chains by invoking multiple events, thus forming a "communication tree".

Rule 14 *An Event's Communicator's presence or absence must be in accordance with Table 2.5.*

ACS models data flow throughout the entire communication tree, and each Event type has its own mechanism for passing data into and out of itself, which causes differences in how the Events' inputs and/or outputs are declared. However, since internal behavior is abstracted away, including the behavior that creates concrete data, no assumptions can be made about concrete values, which are thus not present during verification. Instead, ACS assumes that all non-deterministic choices can be made with the available information at any point.

An Initiator's inputs come from the system on which it is defined and its outputs are returned to said system. The initiating system thus feeds the data needed to start a communication chain or tree and must thus have a Communicator, since without one there would be no communication, which is considered internal behavior. Conversely, a Handler's inputs and outputs come from, and are returned to, the Event/Communicator that invokes it, so the Handler's Communicator is optional.

A Publisher has a single input that comes from the Event/Communicator that invokes it, but no output since it cannot get responses from its Subscribers which it does not know. As such, it cannot have a Communicator. A Subscriber's inputs come from the Publisher in its declaration and its output is returned to the system on which the Subscriber is defined. As such, its Communicator is optional.

We decided on this set of Events and their semantics since the Initiator and Handler allow for direct communication where the sender knows the receiver, whereas the Publisher and Subscriber allow for indirect communication where the receiver knows the sender. By controlling which party knows which, the number of instances wrt. the known party, and the kind of data that is communicated, this combined allows for modeling a wide variety of communication (see Section 2.3.3.4).

2.3.2 Tokens

Figure 2.14 mentions the "`Token`" which is a typed variable that remembers a connection with a system instance or group of instances for later use. Tokens exist because there are cases where communication with the same system instance(s) at multiple points in time must be guaranteed. Thus, similar to how "data flow" is modeled using Variables in Communicators, "connection flow" is modeled using Tokens in both Controllers and Communicators as described in Section 2.3.3.3.

Whereas Single Tokens allow addressing a single system instance, Group Tokens only allow addressing a group of instances as a whole. This is because system instances are selected non-deterministically (see Section 2.3.3.2) when connections are established, which makes the number of instances in a Group Token unknowable at design time.

Tokens have the same structure as any other name in ACS, and Token Types are based on system Ports, since each Port on a Composite or Reference represents a different AS/interface. Table 2.6 shows the Token Type syntax. An AS Port is specified using the name of the AS. A Composite or Reference Port is specified using said system's name and the name of the AS that has an interface connection to the desired Port on the Reference or Composite. Ports are specified the same way for Single and Group Tokens.

System type(s)	Single Token	Group Token
AS	`SINGLE`($qname_{AS}$)	`GROUP`($qname_{AS}$)
REF & COMP	`SINGLE`($qname_{REF/COMP}$[$name_{AS}$])	`GROUP`($qname_{REF/COMP}$[$name_{AS}$])

Table 2.6: The syntax for single/group tokens for ASs, References, and Composites.

The AS names inside square brackets are not qualified since the AS can only be in the scope of $qname_{REF/COMP}$. The values of Tokens are determined non-deterministically by "**`TokenSelectors`**" that are located on the Communicator End States (see Section 2.3.3.2). Tokens can also be used to determine the instance(s) a Subscriber should listen to (see Section 2.3.3.4).

2.3.3 The Behavioral Components

The behavior of an AS is split into one Controller and multiple Communicators (see Section 2.1.2) to 1) separate what an AS is allowed to do at a given time, from how it actually communicates with other systems, and 2) make the behavioral layer more manageable to read and write. Events are thus the agents that bind the Controller and its Communicators together, which is part of the reason why behavior is event-driven.

The Controller and Communicator have a very tight interplay – both within and across ASs – which can be expressed in terms of three kinds of "flows", namely, "Control Flow", "Connection Flow", and "Data Flow" (also mentioned in Section 2.3.1). These flows emerged as a product of the constructs and mechanisms built into ACS. The rest of this section explains the Controller, Communicator, and the three flows.

2.3.3.1 Controllers

As described in Section 2.1.2.2, the Controller describes the orderings and timings of when Events on an AS may invoke. The key component for this is the **`Action`** that is located on **`Action Transitions`** (see Figure 2.14). Table 2.7 outlines all variations of the Action syntax. The non-Delay Actions are collectively referred to as **`Event Actions`**.

Action Type	Syntax
Initiator & Handler	$\overrightarrow{Token_{IN}}$ **:** $name_{INI/HAN}$ **[**$TimeConstraint$**]** **:** $\overrightarrow{Token_{OUT}}$
Subscriber	$\overrightarrow{Token_{IN}}$ **:** $name_{SUB}$**(**$Target$**)** **[**$TimeConstraint$**]** **:** $\overrightarrow{Token_{OUT}}$
Delay	**[**$TimeConstraint$**]**

Nested Components	Syntax
$TimeConstraint$	$name_{Reference\_State}$ $Comparator$ $\overrightarrow{\mathbb{N}\ Unit}$

Leaf Components	Syntax
$Comparator$	**<** | **<=** | **=** | **>=** | **>**
$Unit$	**d** | **h** | **m** | **s** | **ms** | **us**
$Target$	**all** | $Token$

Table 2.7: All syntax variations of the Action component.

Action Transitions

Actions are placeholders for behavior that is invoked "in response to something". For Initiators, that is the AS' internal environment. For Handlers and Subscribers, that is by request from another system. Publishers are invoked to produce such a request and thus cannot be used in an Action. Lastly, the Delay Action exists to provide additional timing for Event Actions or to represent internal operations that take time.

Rule 15 *The Events and Tokens used in Action transitions must be defined on the parent AS.*

The input and output Tokens on Event Actions relate to connection flow, where Controllers define input and output Token Types (akin to Events' input/output Data Types), which is described further in Sections 2.3.3.2 and 2.3.3.3. The **`Target`**-part of a Subscriber Action allows it to listen for published data, either from all instances of the AS that defines the Subscriber's Publisher or only from the instance(s) referred to by a Token.

If an Action Transition is traversed, it must finish the traversal within the time range denoted by the Time Constraint. This range is defined using an arithmetic **`Comparator`** to compare "the latest time of entering a given reference state" with a constant time duration (e.g., `1m30s`, where `1` and `30` are natural numbers and `m` and `s` are time **`Units`**).

Alternatively, the reference time could be compared with a lower and upper duration to give finer timing control, however, double bounded time constraints take up more space, might not always be needed/desired, and can be simulated by two single bounded time constraints. Figure 2.15 shows how the two alternatives can represent the same time constraints (in this case “10 seconds to 1 minute”).

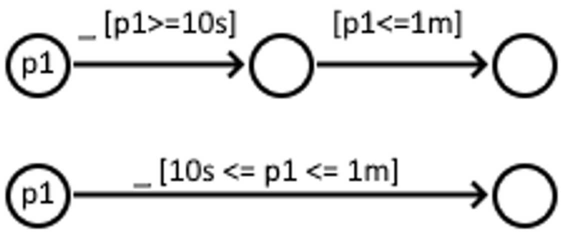


Figure 2.15: Single and double bounded Time Constraints are equal. The interval could be any other valid interval. The underscore can be an Event or nothing.

Delay Actions are traversed instantaneously. If an Event Action has no associated Communicator, it is assumed to finish within its Time Constraint’s interval. In the other case, the Event finishes in an interval decided by the Time Constraints of the Events that the Communicator invokes. If the Time Constraints of the invoked Events make it impossible to satisfy the Action’s time constraint, this is a verification error (see Section 2.5). The local Time Constraints in the Controller can also be unsatisfiable as seen in Figure 2.16.

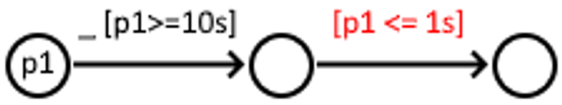


Figure 2.16: It is impossible to satisfy the red time constraint.

Rule 16 *Time Constraints may not make other Constraints in the same Controller impossible to satisfy.*

Rule 17 *The reference state of a Time Constraint must be guaranteed to have been visited before getting to the Time Constraint. Otherwise, the relative time will be undefined.*

Controller states

A Controller State (i.e, **`CON State`** from Figure 2.14) has three possible attributes. Firstly, the start state of a Controllers is marked by the **`Start Mark`**. The `"lockable"` attribute allows other systems/instances to temporarily reserve the right to communicate with the system (instance) that contains the lockable state (see Section 2.3.3.2). Lastly, the list of Tokens on top of a state denotes the scope of a Token.

Since Controllers are cyclic (see the next paragraph), and to make the destruction of a Token/Connection explicit, we annotate the scope of Tokens onto Controller States to avoid them being used in places that were not intended. All states following the transition from which a Token was output are annotated with the Token, and once a state without the Token is reached, this token cannot be used as input anymore.

Overall structure

A Controller must be a single state machine where all states and transitions are reachable, and since the verification process assumes that the number of system instances remain constant, systems cannot terminate which means Controllers cannot have end states. This leads to Rule 18. Furthermore, to avoid unused (but defined) Events that could give a designer the wrong idea about a system's capabilities, Rule 19 requires that all Events are used in an Action and are invoked from a Controller.

Rule 18 *All CON States must be reachable from their parent Controller's start state and have at least one outbound Action Transition.*

Rule 19 *All Initiators, Handlers, and Subscribers defined on an AS must be used on one or more Action Transitions. All Handlers and Subscribers/Publishers must be invoked by one or more Communicators (from any AS).*

2.3.3.2 Communicators

As described in Section 2.1.2.1, Events may or may not initiate new communication with other systems when invoked. Events that do not do this are called "Internal" and "have no associated state machine". This state machine is contained in the Communicator which further describes the Link (see Section 2.2.2.1) over which communication happens. The key components here are the **`Link Specifier`**, **`Start COM State`**, **`End COM State`**, and **`Invocation`** on **`Invocation Transitions`** (see Figure 2.14).

Link Specifier and Input/Output Tokens

Communicators have a Link Specifier to denote which Link all communication initiated from that Communicator must happen over. Note that Handler and Subscriber Events can be invoked over any Link that is connected to the system that defines said Event. The Link Specifier is only ever needed when an Event initiates new communication, and is thus part of the Communicator. The syntax for the Link Specifier is shown just below.

$$< name_{Link} :: \overrightarrow{TokenType_{IN}} \rightarrow \overrightarrow{TokenType_{OUT}} >$$

When invoked, a Communicator/Event non-deterministically selects a required/allowed number of system instances to communicate with as denoted by the presence, Directionality, and Cardinality of Link Connections in the specified Link. Since Links represent a promise about the types and numbers of system instances an Event must communicate with, Rule 20 and Rule 4 (on page 18) ensure that the behavior of an AS is faithful to all the Links it connects to.

Rule 20 *A system that can initiate communication over a Link must have at least one Event (with a Link Specifier) that uses said Link. Furthermore, the Link Specifier can only specify a Link that is connected to the parent AS' Port (or an alias thereof).*

Links further restrict the Link Specifier's input/output Token Types as stated in Rule 21, Rule 22, and Rule 23. As an example of Rule 22, if a Link allows an Event to communicate with two instances of `ExampleSystem`, then no more than two Single Tokens for `ExampleSystem` may be given as input to that Event. If less than two tokens are given, the remaining instances are selected non-deterministically from the total set of instances.

Rule 23 disallows using Group Tokens for systems with a Z-Link-Cardinality since such systems must be individually addressable, which Group Tokens do not support. Conversely, Single Tokens can be used with **`N`**- and *-Link-Cardinalities to be able to individually address a subset of all corresponding system instances involved in communication.

Rule 21 *The input and output Token Types can only refer to systems that are connected to the specified Link.*

Rule 22 *The input and output Token Types cannot allow inputting/outputting more system instances than involved in communication. When the Link and System Cardinalities are* **`N`** *and/or* **, there is no upper limit.*

Rule 23 *Only Single Tokens (see Section 2.3.2) can be used as input for systems with a Z-Link-Cardinality. Both Single and Group Tokens can be used for systems with* **`N`**- *and* **-Link-Cardinalities.*

Finally, output Tokens are given the type(s) of the output Token Types of the Link Specifier. Rule 24 states how to correctly annotate input and output Tokens on Action transitions. The "values" given to output Tokens are defined on the end states of Communicators as described in the following.

Rule 24 *The number of input and output Tokens on an Event Action Transition must correspond with the number of input and output Token Types in the given Event's Link Specifier.*
The types of input Tokens must be the same as the Token Types given in the Link Specifier.

Communicator States

Figure 2.14 shows the three types of Communicator States (**`COM States`**). The Start COM State gives names to the input Data Types and Token Types (from the Event Declaration and Link Specifier), where data names are called Variables and Token names are still called Tokens. Variables and Tokens are used in Invocations, which are described shortly. The actual data and Tokens that are output by an Event are specified on End COM States using **`Variable Selectors`** and **`Token Selectors`**. Table 2.8 shows the syntax of these.

Selector Type	**Syntax**
Variable Selector	$name_{Variable}\overrightarrow{\texttt{.}name_{RecordField}}$
Token Selector	$qname_{AS}\texttt{(}InstanceSelector\texttt{)}$ $\mid qname_{SSM}\texttt{(}InstanceSelector\texttt{)[}name_{Port}\texttt{]}$

Sub-Components	**Syntax**
$InstanceSelector$	$\mathbb{Z}^{+} \mid$ **`all`**

Table 2.8: The syntax of Variable Selectors and Token Selectors.

The Variable Selector outputs the value of a Variable or a field on the Variable if the Variable is a **`Record`** (see Section 2.4.2.3). The Record Data Type contains named, typed fields, which can also be records. Therefore, zero or more fields can be appended after a Variable to select the value of either the Variable or a (however deeply nested) field.

The Token Selector uses Ports to identify the type of Token to create (similar to defining Token Types), but also adds the **`Instance Selector`** to either select a single instance (creating a Single Token) or all instances involved in communication (creating a Group Token). Single Tokens can only be created for systems with a Z-Link-Cardinality, in which case each of the $x_{total} \in \mathbb{Z}^{+}$ involved instances can be uniquely addressed by a positive integer $x_{instance} \in \mathbb{Z}^{+}$ where $1 \leq x_{instance} \leq x_{total}$.

Tokens are created inside Communicators since they can only be created when connections are made to other systems. The Variable and Token Selectors are placed on End COM States, since 1) it mirrors the declarations on the Start COM State, and 2) since there is no gain in creating them earlier in the Communicator. Finally, as with Action Transitions, the inputs/outputs on Start and End COM States must correspond with the Link Specifier.

Rule 25 *The number of Variables and Tokens defined on the Start COM State must correspond exactly to the number of input Data Types from the Event Declaration and input Token Types from the Link Specifier.*
The number of and types produced by Variable and Token Selectors defined on the End COM State must correspond exactly to the number and types of output Data Types from the Event Declaration and output Token Types from the Link Specifier.

Sub-Machines and Functions

Figure 2.14 shows that Communicators can have any number of **`Sub-Machine`** and **`Function`** components, where both take inputs and produces outputs. These must be declared in the Communicator using the syntax below, where a separate State Machine Diagram associated with each Sub-Machine declaration (see Figure 2.17) is created.

$$\texttt{function}\ name :: \overrightarrow{DataType_{IN}} \longrightarrow \overrightarrow{DataType_{OUT}}$$
$$\texttt{machine}\ name :: \overrightarrow{DataType_{IN}} \longrightarrow \overrightarrow{DataType_{OUT}}$$

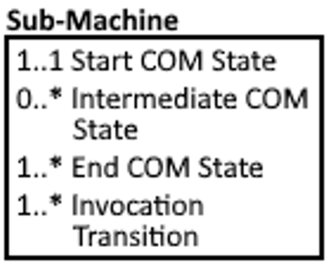


Figure 2.17: The structure of a Sub-Machine.

A Sub-Machine is like a Communicator, but cannot declare Functions or Sub-Machines, and does not have a Link Specifier. Thus, Sub-Machines reuse Tokens, Functions, and Sub-Machines defined on the Communicator. A Function is a placeholder for internal logic or behavior that is assumed to exist in any implementation of the AS that defines the Communicator with said Function. ACS models only the inputs and outputs to represent that an operation or data transformation happens, but not *how* the input is transformed into the output.

Invocation Transitions

Invocations generally represent “requests to make something happen”, but their effects can vary greatly as explained in the following. Table 2.9 outlines all variations of the Invocation syntax.

Invocation Type	**Syntax**
Publisher	$\overrightarrow{VariableSelector_{IN}}\ \mathtt{:}\ name_{Publisher}$
Handler	$InputSelector\ \mathtt{:}\ TargetSelector\ \mathtt{:}\ HandlerOutput$
Lock/Unlock	$\mathtt{lock}\ Instance \mid \mathtt{unlock}\ Instance$
Function/Sub-Machine	$\overrightarrow{VariableSelector_{IN}}\ \mathtt{:}\ name_{Func/SubMac}\ \mathtt{:}\ FuncMacOutput$
Variable Declaration	$name_{Variable}\ \mathtt{=}\ Value$

Nested Components	**Syntax**
$InputSelector$	$\overrightarrow{VariableSelector} \mid \mathtt{[}\overrightarrow{VariableSelector}\mathtt{]}$
$TargetSelector$	$Instance\mathtt{.}name_{Handler} \mid \mathtt{[}Instance\mathtt{.}name_{Handler}\mathtt{]}$
$HandlerOutput$	$\overrightarrow{Variable} \mid \mathtt{WAIT} \mid \mathtt{SKIP}$
$FuncMacOutput$	$\overrightarrow{Variable} \mid \mathtt{WAIT}$
$Instance$	$qname_{AS}\mathtt{(}ID\mathtt{)} \mid qname_{Subsystem}\mathtt{(}ID\mathtt{)}\mathtt{[}name_{Port}\mathtt{]}$
$Value$	$\mathtt{(}\overrightarrow{name_{Field}\ \mathtt{=}\ VSL}\mathtt{)} \mid name_{Record\_Type}\mathtt{(}\overrightarrow{name_{Field}\ \mathtt{=}\ VSL}\mathtt{)} \mid \mathtt{[}\overrightarrow{VSL}\mathtt{]} \mid name_{Array\_Type}\mathtt{[}\overrightarrow{VSL}\mathtt{]}$

Leaf Components	**Syntax**
$VariableSelector\ (VSL)$	$name_{Variable}\overrightarrow{\mathtt{.}name_{RecordField}}$
ID	$\mathbb{Z}^{+} \mid Token \mid \mathtt{all}$

Table 2.9: All syntax variations of the Invocation component.

Invocations can be separated into two groups: “those that synchronize with other system instances” (**`Publisher`**, **`Handler`**, and **`Lock/Unlock`**) and “those that affect the local control and data flow” (**`Function/Sub-Machine`** and **`Variable Declaration`**). Given the complexity of Invocations, some design choices are explained in later sections to make this section less convoluted.

The Publisher and Handler Invocations initiate communication with other systems, and their syntax reflects that Publishers only have inputs, whereas Handlers have both inputs and outputs. The Publisher Invocation allows corresponding Subscriber Action Transitions in other AS instances’ Controller to traverse, where the Handler Invocation allows Handler Action Transitions in other AS instances’ Controller to traverse. Section 2.3.3.4 explains

ACS' communication mechanisms (semantics) in detail.

The Variable Selector (abbreviated "VSL" and previously introduced in and explained after Table 2.8) selects the input data for Invocations. The variations of **`InputSelector`**, **`TargetSelector`**, and **`HandlerOutput`** in Handler Invocations are used to configure how to send data, select receiver system instances, and handle/receive output returned from the receiver system(s). The valid combinations of these depend not only on each other, but also on the definition of the invoked Handler. See Section 2.3.3.4 for more details.

The **`TargetSelector`** selects which Handler on which system (based on a system Port) to invoke, as well as the instance(s) of said system that should be involved in communication based on the **`ID`** component. This component works similar to the "Instance Selector" (which is introduced in and described after Table 2.8), but also allows using Single and Group Tokens to select the receiving instance(s).

The Lock and Unlock Invocations temporarily acquire exclusive access for the parent AS instance to communicate with one instance (recall that Controller States can be `"lockable"`). Group-locks are not supported due the complexity of the semantics. The `lock` and `unlock` operations are placed on Invocation Transitions instead of states since not all choices leaving a state necessarily requires locking/unlocking. They further use `ID` to select a target in the same way as Handler Invocations (see above). Section 2.3.3.3 explains the semantics of Locking in full detail.

The syntax for Function and Sub-Machine Invocations reflects that both take data inputs and produce data outputs. The output of such Invocations can either be `WAIT` if the output type is `void`, or a list if variables if it's non-`void`. Functions and Sub-Machines must always wait for completion, whereas Handler Invocations can output `SKIP` to not wait for an Invocation to finish (applicable only to `void`-output handlers). Functions are assumed to not break time constraints. Invoking Sub-Machines "executes" their State Machine.

Finally, a Variable Declaration either composes a Record or an Array Type from existing Variables, where Variable Selectors are used to select the record field and array element values. There is a variation with and without a "Type Name" for Records and Arrays to create an instance of an anonymous Record/Array or of a declared/names Record/Array Type (see Section 2.4.1). After the Invocation Transition, the variable "$name_{Variable}$" can be used like any other variable.

Rule 26 *The number of inputs (and outputs) on a Invocation must correspond exactly to the number of inputs (and outputs) defined on the Publisher, Handler, Function, or Sub-Machine that is invoked.*

Overall Structure

A Communicator and all its Sub-Machines must be single state machines where all states are reachable from their start state, and since verification must be able to guarantee that Communicators terminate at some point, Communicators/Sub-Machines can neither be cyclic nor recursive. This gives rise to Rule 27. Furthermore, to avoid unused (but defined) Publishers that could give the designer the wrong idea about a system's capabilities, Rule 28 requires all Publishers be used in an Invocation.

Rule 27 *All Start COM States and Intermediate COM States must have an outbound Invocation Transition, and all End COM States and Intermediate COM States must have an inbound Invocation Transition. However, no cycles may exist in Communicators or Sub-Machines, and no (indirectly) recursive invocations of Sub-Machines are allowed.*

Rule 28 *Only the AS that defines a Publisher can invoke that Publisher, and all such Publishers must be used on one or more Invocation Transitions.*

2.3.3.3 Control, Connection, and Data Flow

Whereas data flow is explained in Section 2.3.1, control and connection flow are yet to be described in detail, which this section does.

Control Flow

Control flow is the passing of control between states and transitions, between Controllers and Communicators, and between systems. Each AS instance has a single control flow starting in its Controller's start state. Control is always returned to the source Controller, Communicator, or system when passed between either of these components (similar to a "call stack" in programming languages).

Decision making and communication behavior in ACS are inherently non-deterministic, so all "valid choices" across all Controllers, Communicators, and systems are made simultaneously during the verification phase. However, constraints such as Cardinalities, Time Constraints, Tokens, and the structure of Controllers/Communicators restrict the set of valid choices to approach some intended behavior. We use non-determinism since any predefined ordering has no guarantee to be representative of a designer's intentions.

At the lowest level, control alternates between a state and a transition. At a higher level, certain Action invocations in Controllers can temporarily pass control to the start state of a Communicator, in which certain Invocation Transitions can temporarily pass control to

Sub-Machines or other systems using Handler Invocations. Publishers do not pass control to other systems since they do not receive responses.

For Handler Invocations, two control flows are required to perform the invocation. The first control flow comes from the AS whose Communicator performs the Invocation, where the second control flow comes from the AS whose current CON State must have an outbound Action Transition that allows the invocation of said Handler. Section 2.3.3.4 explains how a Handler on multiple system instances can be invoked simultaneously, in which case invoking system's control flow is split and passed to all target instances.

Subscriber Actions also require two control flows, however, the Publisher Invocation needed to invoke a Subscriber never knows if any Subscriber reacts to it or not, and thus only needs one control flow to Invoke.

Finally, Lock/Unlock and lockable CON States affect control flow by temporarily allowing only a single AS instance to invoke Events on the (to-be) locked AS instance. Locking requires the to-be locked AS' Controller to be in a lockable state and the locking Communicator to make a Lock Invocation. Unlocking requires just one control flow and must be done at any time before the Communicator reaches an end state. All remaining Actions and Invocations require only a single control flow.

Rule 29 *All target AS instances for a Handler Invocation must be in a state that allows the given Handler to invoke. The target AS instance of a lock operation must be in a lockable state. Only Subscriber Actions that are currently available for traversal in their respective Controllers react to corresponding Publisher Invocations.*

Rule 30 *All lock operations must be followed by a corresponding unlock operation in all Communicator paths after the lock operation. All unlock operations must be preceded by a corresponding lock operation in all Communicator paths leading to the unlock operation.*

Connection Flow

Connection flow is the passing of Tokens between Communicators throughout a Controller, where Tokens represent connections to specific system instances as described in Section 2.3.2. A Token remembers the instance of the outermost system that the Token refers to, since the outermost system is observed as a single/atomic unit. Thus, if a Token refers to the single port of an AS, one specific instance of that AS is remembered.

However, if a Token refers to a Port on a Composite or Reference, it will only remember the Composite or Reference instance; not the AS instance on the inside that is aliased (see Figure 2.9) with the parent system's Port, even if there are multiple instances of the aliased AS. Thus, connection initiated using such a Token only guarantees that communication reaches the same Composite or Reference instance; not the the same AS instance on the inside.

Subscriber Actions may use Tokens to determine the instance(s) of a system from which to listen for published data (see Table 2.7), but can also listen for data on all publisher-system instances. The ability to use Tokens to filter messages from certain systems increases the flexibility and strength of the publish-subscribe mechanism which is explained in depth in the following section.

2.3.3.4 Communication Mechanisms

ACS have four fundamental **`Communication Mechanisms`** that are culminations of all three flows (see Section 2.3.3.3). These mechanisms are: **`Publish`**, **`One-to-One`**, **`Many-to-Many`**, and **`One-to-Many`**, where the "X-to-Y" mechanisms are all based on Handlers. This section is not self-contained and builds upon the prior knowledge presented throughout Section 2.3, since too many details would otherwise have to be introduced simultaneously.

Handler Invocations use optional square brackets ("`[]`") in the syntax to indicate which of the three "X-to-Y" mechanisms to use. Table 2.9 presents this syntax. The presence of square brackets around the `InputSelector` indicates that "X is Many" and around the `TargetSelector` indicates that "Y is Many". Conversely, no square brackets around these, respectively, indicate that "X is One" and "Y is one". Thus, while "Many-to-One" is supported by the syntax, this is not a valid communication mechanism.

Furthermore, whether or not some communication mechanism can be used depends on the Link Cardinality of the system being communicated with and the method used to identify the instance(s) thereof. Table 2.10 shows an overview of which Link Cardinalities and instance identification (`ID`) methods that are compatible with which communication mechanisms under which circumstances. There is one exception: if One-to-Many's output is `WAIT`, the *-Link-Cardinality is invalid for the `ID = 'all'` case.

	Compatible Link Cardinalities for target system			**Supported `IDs`**
Mechanism	**Always**	**`ID = Token`**	**`ID = 'all'`**	
Publish	$\mathbb{Z}^+$, `N`, `*`			
One-to-One	$\mathbb{Z}^+$	$\mathbb{Z}^+$, `N`, `*`		$\mathbb{Z}^+$, Single Token
Many-to-Many	`N`	`N`, `*`	$\mathbb{Z}^+$, `N`	Group Token, `'all'`
One-to-Many	`N`	`N`	$\mathbb{Z}^+$, `N`, `*`	Group Token, `'all'`

Table 2.10: Which Communication Mechanisms that can be used with which Link Cardinalities and instance identification (`ID`) method. Gray cells have no effect. Remember, there is a difference between a $\mathbb{Z}^+$-Cardinality and a $\mathbb{Z}^+$-`ID`.

As seen in the table above, the Z-Link-Cardinality cannot be used when a Group Token is used to identify the target system instances. This is because Group Tokens cannot be used with the Z-Link-Cardinality (see Rule 23). In the following, the Publish mechanism is called "indirect" since the receivers know sender(s), whereas the Handler-based mechanisms are called "direct" since the sender knows the receiver(s).

The Indirect Communication Mechanism

The Publish Communication Mechanism relies on two conditions. First, a Subscriber Action Transition in a Controller that listens for data on a Publisher from one or more instances of the Publisher's system of definition. Second, a Publisher Invocation on one of said system instances that publishes data to the Publisher over a Link that connects the sender to the receiver. If no Subscribers are listening, none are invoked.

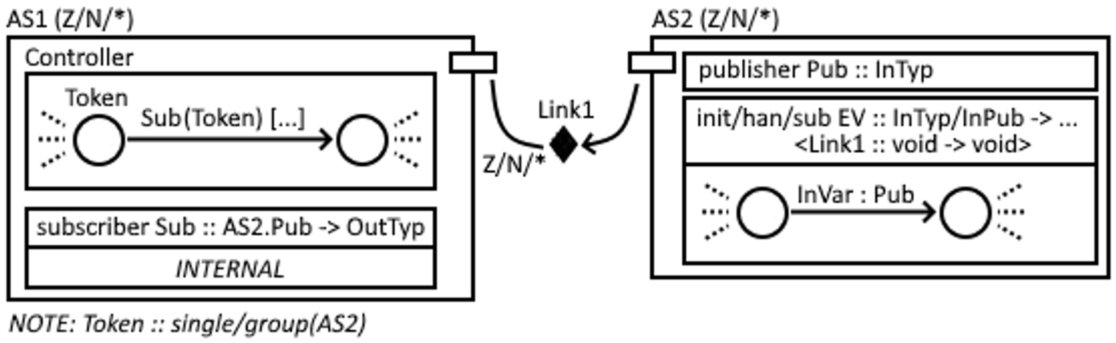


Figure 2.18: Approximate structure to allow the Publish Communication Mechanism.

Since there are no time constraints on Publisher Events, it is assumed that the time the Publisher takes to send out all data does not break the time constraint on the Communicator that invoked the Handler.

Publish can be used with all Link Cardinalities, so the published data can be directed to a fixed number of system instances ($\mathbb{Z}^+$), an unknown number of system instances (`N`), or all system instances (`*`). The two first cases may sound out of place, since if the sender does not know the receivers, how could the message be directed to only specific receivers? That directed behavior is instead a property of the medium/Link that carries the message, since this would know all receivers and thus have the knowledge to do so.

Alternatively, Publish could simply not work with $\mathbb{Z}^+$ and `N`-Cardinalities, or could treat them as a `*`-Cardinality, thus sending the message to all receivers in all cases. However, these two alternatives break the promise of Links: that Communicators communicate with exactly the number of instances required/allowed by the Cardinality. Therefore, we decided on the semantics outlined in the previous paragraph.

The Three Direct Communication Mechanisms

One-to-One, Many-to-Many, and One-to-Many mechanisms rely on two conditions. First, there is a Handler Action Transition in the Controller of all receiver system instances that are all currently possible to traverse from their Controllers' current state. Second, a corresponding Handler Invocation in a Communicator that communicates over a Link which connects the sender system to the receiver system. Unlike with Publish, all receiver systems must respond to the Handler Invocation (except for with `SKIP` described below).

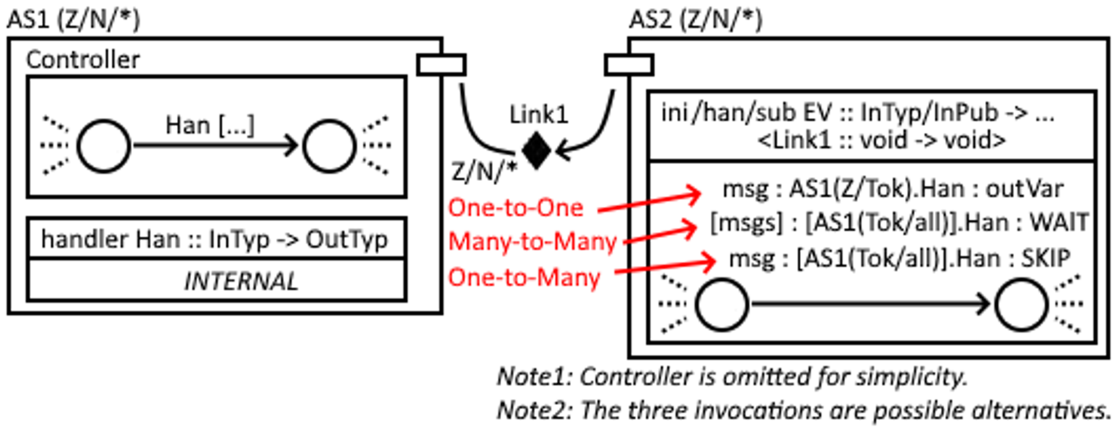

AS1 (Z/N/*)
Controller
Han [...]
handler Han :: InTyp -> OutTyp
INTERNAL
Link1
Z/N/*
AS2 (Z/N/*)
ini /han/sub EV :: InTyp/InPub -> ...
<Link1 :: void -> void>
One-to-One
Many-to-Many
One-to-Many
msg : AS1(Z/Tok).Han : outVar
[msgs] : [AS1(Tok/all)].Han : WAIT
msg : [AS1(Tok/all)].Han : SKIP
Note1: Controller is omitted for simplicity.
Note2: The three invocations are possible alternatives.

Figure 2.19: Approximate structure to allow the "X-to-Y" Communication Mechanisms.

Handler Declarations have both input and output Data Types, and the allowed output(s) of a Handler Invocation (see `HandlerOutput` in Table 2.9) depends on the output Data

Type. If the output Data Type is *not* `void`, the output is a list of Variables in which to store the output data, however, if it *is* `void`, the output can either be `SKIP` which does not wait for a response, or `WAIT`, which waits for a response. When `SKIP` is used, it is not an error if a target does not react to the Invocation. As Table 2.10 states, there is a case in One-to-Many communication where only `SKIP` is allowed (elaborated later).

When there are "Many" inputs (indicated by "`[]`" around the input), the Data Type of the input must be "array of Handler input(s)". Similarly, if there are "Many" targets, the *output* Data Type(s) become "array of Handler output(s)", since multiple receivers outputs a response. "Many targets" allows anywhere from zero to all instances in the SoS to be involved in communication. An empty array is output with zero instances.

`One-to-One` communication sends a single message to a single target system instance, which can be selected either using a numeric ID (only usable with Z-Link-Cardinality) or a Single Token which can be used with any Link Cardinality to the target system. Due to the single target behavior, neither the Group Token nor the `all` instance identifiers can be used in One-to-One.

`Many-to-Many` communication means to "send many individual messages to equally many individual target system instances". That is not "send all messages to all receivers" as the name could be misunderstood as. We thus assume that the sender knows the exact number of targets and prepares an equal number of messages to send.

`One-to-Many` has two flavors. When the output is `SKIP`, One-to-Many sends a single message to all target system instances that can react to the Invocation regardless of the Link Cardinality since nothing apart from the invoked Handler must be known about the targets. When the output is a Variable list or `WAIT`, the number of targets must be known in order to receive responses, in which case only Z and **`N`**-Link-Cardinalities are allowed.

With many receivers, it might be necessary to send individual messages to each. We assume that communication is initiated with all target system instances simultaneously, such that the delays between established communications does not need to be considered on top of the target Handler's time constraint. With the `SKIP`-output and many targets, we assume that sending data does not break the parent Communicator's time constraint.

2.3.4 High-Level Features and Limitations

Section 2.2.3 describes how the structural model can model systems and Links between these at any scale and irrespective of the physical platforms on which they are implemented, which the behavioral model builds upon.

ACS' Communication Mechanisms allow modeling the communication of "data" between a "sender" and "receiver" over some "medium", based only on whether the sender knows the target or vice versa, since this property remains the same regardless of implementation. A client computer could send a message to a server, or an employee could make a report to the manager. Smartphones and statistics services could listen for weather updates from a weather station, or adults and kids could listen for the sound of the ice cream truck.

The Communication Mechanisms combined with Links further abstract away how data/-communication is routed between the sender and receiver.

One weakness, however, is that one instance or configuration of communication can possibly map to multiple routing mechanisms with different implications, but this cannot be discerned in the model. If any one instance of **N**-many system instances is communicated with, would this be done using unicast or a anycast? If there are multiple instances of the AS that is aliased with the Port of a Composite or Reference, how is the specific instance selected (e.g., round-robin) when communicating with said Port on the Parent system?

Problem 2 from out 9th semester project [11, p.36] describes the need for a communication model that is agnostic to specific implementations or platforms, such that a model is not restricted to specific technologies or approaches. While the behavioral model does achieve this, it instead sometimes becomes ambiguous as to how a model should be implemented, which is an undesirable property of a specification. Section 2.4.4 describe how the data model also contribute to this problem.

2.3.5 Relation to Real Communication Protocols

The Communication Mechanisms described above in Sections 2.3.3.4 and 2.3.4 are based on universal assumptions about the nature of communication, and are thus detached from typical implementations/methods of communication. As hinted to in Section 2.3.4, the various configurations and combinations of Cardinalities and Communication Mechanisms can be mapped to concrete communication protocols and routing schemes that belong to various domains like computer networking, human-to-human interaction, or similar.

It is, however, easiest to map to networking concepts, since these have been standardized. For the purpose of this section, communication can be split into two parts: a procedure to select one or more receivers and a procedure to transfer data. To select receivers, these must have identities. To transfer data, we need a data exchange standard. To move data from sender to receiver(s), we need a procedure to route data to the correct destination.

The internet uses 4 general Routing Schemes [24] – namely "Unicast", "Broadcast", "Multicast" [25], and "Anycast" [26] – which provide different ways of selecting receivers and routing data to these. The receivers' identities are thus IP addresses or similar identifiers associated with networked computers. Finally, the standard for exchanging data can map to transport protocols such as TCP [27] or UDP [28].

One-to-One could use Unicast or Anycast to select one target instance and use TCP to communicate between systems. In case the output is `SKIP`, UDP could be used since no responses are needed. Many-to-Many is equivalent to a multiple One-to-One communications, since individual messages are sent to each system instance. One-to-Many could use Broadcast or Multicast to select all target instances in a SoS, or it could use Multicast to select a proper subset thereof. Only UDP supports these two protocols.

The Publish mechanism could map to Broadcast or Multicast, since receivers could filter broadcast messages or opt into receiving Multicast messages (IP Multicast [25]). However, published messages in ACS cannot "linger", such that even late-coming Subscribers can get the published data. Some publish-subscribe systems such as MQTT (specifically the broker [29]) allow this, which suggests that ACS' Publish mechanism lacks functionality.

2.3.6 Changes since the old version

ACS' initial Communication Mechanisms were based on the (then) two Link Cardinalities: Z-cast and `*`-cast. With the addition of the **`N`**-Link-Cardinality came the **`N`**-cast, and after deliberation about the flexibility of these mechanisms, we discarded these Cardinality-based mechanisms in favor of the "X-to-Y" mechanisms from Sections 2.3.3.4 and 2.3.4. The Publish mechanism was introduced to make better use of the new Boundary Links.

Tokens and the notion of connection flow were added to make the preservation of connections to system instances explicit, since the ability to remember a system for later operations is essential to model certain kinds of communication. Another reason Tokens were accepted is that they allow Subscribers to filter published messages.

Sometimes, multiple Handler/Subscriber Actions in a Controller must be invoked in sequence by one specific system instance. However, it was previously possible for another system instance to "steal" one of said invocations while the first system got ready to invoke the next Event. Thus, locking was added to avoid this by allowing temporary exclusive access to an AS instance.

Since the Record Data Type (see Section 2.4.2.3) contains fields, it would make sense to be able to reference field values in Communicators. As such, we added the "Variable Selector", which also inspired the "Variable Declaration" that creates a Record or Array Variable based on existing Variables. Thus, if the input Data Type of an Event requires that existing variables are composed into a Record or Array, this can be modeled explicitly.

This version of ACS adds explicit declarations for Functions and Sub-Machines to reduce the risk of these getting the incorrect Data Types due to implicit declarations. Previously, Functions were implicitly declared on Invocation transitions where the input Data Type(s) were determined by input Variables and the output Data Type(s) were explicitly stated in-line. Similarly, a Sub-Machine's input and output Data Types were both implicitly defined, where an explicit Sub-Machine diagram would model its behavior.

“Error handlers” have been removed, but we plan to re-introduce them again in the future. An Error handler is an Action that is traversed when an error happens (similar to the “try-catch” construct seen in programming languages). However, the semantics thereof and the definition of “error” are both vague. For example, do we handle internal and/or communication errors? Also, what assumptions can be made about the frequency and types of errors? Answering all these questions could turn into its own project.

2.4 The Data Model

The data model does not represent any systemic part of a modeled SoS but instead exists to support the type system used by the behavioral model, in the form of new type definitions. As described in Section 2.3 in more detail, types are used by events and communication constructs to indicate what values they expect and require in order to perform their intended functionality, as well as what values it will generate as a result. This can be used to verify whether certain chains of communication are valid, track the flow of data, and to help identify potential emergent behavior.

A major point to note is that the data model is heavily abstracted, due to the design motivations behind ACS. The data model does not concern itself with specific values – As it lacks the logic to generate them – but instead considers values as a value-space, and whether certain value-space are compatible with each other.

A type definition is comprised of a structure and a set of constraints on said structure. The type structure is used to define the composition of the type such as what value-space it has, what its base is (if applicable), and whether it has any fields (if applicable). Constraints are used to limit the possible value-space of a type based on physical and theoretical properties of the systems that will use said type. A physical property could be that a system runs on a 32-bit architecture, while a theoretical property could be a function that requires negative integers, which the surrounding data flow must keep in mind, to avoid errors.

As mentioned in Section 2.1.3, every SoI and AS has a “Type Document” inside which **`type declarations`** can be placed. Creating type declarations is the process of binding names to type definitions. In order to express a type definition, ACS provides a small DSL with which to express structure and constraints. The language (and therefore the data model) is purely textual – as opposed to the diagrammatic approaches of the structural and behavioral models – due to problematic interactions between the diagrammatic approach and ACS’ categories of types. Due to the resulting limitations of this individual approach, ACS features multiple mechanisms for reducing redundancy, namely the pre-defined and global types, as well as the **`type propagation`** mechanism. This mechanism serves the purposes of ensuring that systems have access to their required types, and helps the user identify whether their intentions match the model they have created.

- Section 2.4.1 will present the type system, and describe how it works.
- Section 2.4.2 will detail the elements available for defining a type's structure, as well as what constraints are available for said elements.
- Section 2.4.3 will go into detail about the exact mechanisms of the type system's propagation feature, and the reasoning behind its inclusion.
- Section 2.4.5 will touch on how ACS' type system has changed since its last edition. This does not go into any detail about the current system, but provides a historic overview, in case the additional perspective would be of interest.

2.4.1 The Type System

The purpose of ACS' type system is to ensure that the modeled events and chains of communication, separate from their behavioral validation, will receive both a compatible type structure and that the received value-spaces do not contain unsupported values. Types in ACS are divided into four categories:

Pre-defined : Pre-defined types are defined by a specific ACS implementation, and can be re-used across models and their subsystems.

Global : Global types are defined for a single model (emplaced in said model's SoI's type document), and are types that can be used by all subsystems in said model.

Scoped : Scoped types are defined in the type documents of ASs, and can only be used by said system and any systems the types have propagated to (Futher explanations on this topic will follow)

Anonymous : Anonymous types are unnamed, and are defined dynamically when specifying the type of an event or communication construct. As such a type cannot referenced, it cannot be extended or specialized.

With a large model, with many subsystems can come a large set of types in use. Many types will not be used by most systems of a model, and it would thus be a waste of both space and potential verification opportunities to allow systems unilateral access to all types. Likewise, some types will be used my a select multitude of systems, and must therefore both be available on these systems and verifiable as the same types. Lastly, as an extension of the last two points, types used in communication must be available on the systems purporting to use them. ACS, therefore, includes a **`type substitution`** mechanism and a **`type propagation`** mechanism to handle these issues. The first mechanism is used to identify when differing types can be used instead of the formal type that has been defined for the behavioral component. This increases the flexibility of ACS. The latter mechanism is used to spread types to those systems, which – based on their communication chains – should have them available for usage. This allows for further verification of communication chains. Both mechanisms are designed to be handled by an IDE or supporting tool, and are thus more complicated than what is expected of a novice user to manually apply on a practical level. As mentioned earlier, the type propagation mechanism is described in further detail in its own section.

Both of these mechanisms are further complicated by ACS' support of type inheritance, which moderately affect how each mechanism function. ACS Supports two forms of inheritance; specialization and extension. The first form is used to create a new type, with a more restricted value-space than the original, by adding (or tightening existing) constraints. The second form can likewise be divided into two forms. The first of these is only applicable to types which uses ACS' record element in their structure, and adds additional fields to the record, expanding its value-space. However, a key fact to note is that it does not expand the value-spaces of existing fields, nor rename these. The last form involves the addition of array or nullable type structure elements to a type. These additions will increase the original value-space, or break the interface provided by the base type. In the list of type categories, a type can always inherit from a category listed above it as well as from any of the fundamental types. So an anonymous type can be derived from all types and a global type can only be derived from other global types or a pre-defined type.

The type substitution mechanism behaves differently, depending on which types the actual and formal types belong to, and their internal structure. Generally, if a type is derived from another, then it can always (unless it has been made nullable or an array) substitute the original. Otherwise, the mechanism can either choose to decide based on *structure* or on *name*.

Structural comparison is performed when the formal type is either a pre-defined type, a global type, or an anonymous type. In all three categories, as long as the interface are compatible, then any actual type of any of the four categories can be used instead. By interface, it is meant that all type elements in the formal type exist in the actual type. That is, the same fundamental type (if not a record), the same fields – both in their type structure and name – (if a record), being nullable or not, and the same number of array dimensions being present in both the formal and actual types. Finally, the constraints of the actual type must limit its value-space to be a subset of the formal type's. Some examples are shown in Listing 3.

```
int64 = num {.[-2^63; 2^63-1], .(0)}
int32 = int64 {.[-2^31; 2^31-1]} //Can be substituted for int64
uint64 = int64 {.[0;2^64], .(0)} //invalid type. constraint adds new values to type
uint64 = num {.[0;2^64], .(0)} //Cannot be substituted for int64
int32v2 = num {.[-2^31; 2^31-1]; .(0)} //can be used to substitute both int64 and int32.
int32a = int32[] {} //cannot be used to substitute int32

rec = (size int32, name char[]) {}
rec2 = (rec: weight int64) {} //Can be used as substitution for rec
rec3 = (size int32v2, name char[], weight int64) {} //Can substitute rec and rec2.
rec4 = (height int32, designation char[]) {} // Cannot be used as substitution for rec
rec5 = (size int32, name char[])? {} // cannot be used as substitution for rec
```

Listing 3: Examples of type substitution using comparison by structure.

Comparison by name is only used when the formal type is a scoped type. In this case, instead of looking at the structure and value-space of a type, the comparison is performed using the names of the types and their bases. This means that in order to validly substitute a scoped type, the actual type must be derived from the formal type. A final note, if the actual type (or one of its bases) is an array or nullable extension of the formal type, then the type substitution will not be valid.

2.4.2 Type Structure Elements

As described earlier, a type in ACS is composed of a type structure and a set of constraints. The structure is described using a set of **`elements`**, which can be combined to define structures, with unique constraints for each element. The goal of the following subsections will be to detail these elements and their corresponding constraints, explain how to use them, and the reasoning behind their inclusion in ACS.

2.4.2.1 Fundamental Types

Fundamental types are included in ACS to provide a set of large value-spaces from which new types can deliminate their own value-spaces. The fundamental types have therefore been chosen – in combination with the remaining type structure elements – to provide a user with the ability to express a large variance in types. However, some common types such as a *set* are impossible to express. The reasoning behind this choice is three-fold: it keeps the DSL compact, direct support is unnecessary for ACS' verification processes due to values having been abstracted away, and it can be represented using other, existing elements as the ultimate responsibility for some data structures fall on those parts of a system that have been abstracted away. This does mean that some potential errors cannot be caught by the type system and the effectiveness of code generation is reduced, but this was deemed to be acceptable, based on experiences made with the last version of ACS' data model.

As mentioned in Section 2.1.3, ACS includes three "*fundamental types*": ***number***, ***character***, and ***boolean***, refered to as "`num`", "`char`", and "`bool`", respectively. Table 2.11 provides an overview of the fundamental types, their value-space, and what constraints exist for said type.

Fundamental Type	Values	Constraint	Explanation of Constraint
num	Any real number between $-\infty$ and ∞, with infinite precision.	[$\mathbb{R}$;$\mathbb{R}$]	Used to indicate valid range of values.
		($\mathbb{N}$)	Used to indicate number of digits in fractional part of a real number. Setting this to 0 transforms the number into an integer.
char	Any character in an encoding of choice. The exact encoding is left to the implementer/user.	"$\overrightarrow{REGX}$"	Used to indicate valid intervals of values for the char.
bool	True, False	N/A	N/A

Table 2.11: A table over the fundamental types in ACS, what values they represent, and what constraints can be added onto the types when creating new ones.

The fundamental type **`num`** is used to represent the value-space of all possible numbers, both real numbers and integer numbers. The type supports two constraints. The first is the **`range`** constraint expressed using an interval ".[min_val;max_val]". The interval is inclusive, and defines a sub-range of the base type's value-space, which the new type will support. This constraint is particularly relevant when creating types which are limited by the physical hardware they are used by, such as 32-bit architectures or more resource-limited platforms such as IoT-devices. The second constraint is the **`precision`** constraint, which is used to specify the number of decimal digits that the type will support. This can be done with a simple integer ".(int)" This is relevant when defining whether the new type will be an integer (and therefore support 0 decimal digits) or a real number (and at what point round it will become relevant).

The fundamental type **`char`** is used to represent any characters in some character encoding.

ACS does not specify a standard encoding, which is instead up to the implementer or user. As a result of this, it is possible to use characters such as non-latin letters, emojis, kanji ideograms, and more. The type does not represent strings, only individual characters, and the former can instead be defined by applying the `array` element (more on this in Section 2.4.2.4) to the `char` type. This fundamental type can be constrained using the **`regex`** constraint. This constraint allows a user to express a set of ranges in the encoding, in a syntax similar to that of regex, which restricts to value-space to said ranges. Ranges are indicated by writing a dash between the two ends of the range. Individual characters are simply written in. There are no spaces or similar between characters, as they may be present in the encoding. Quotation marks and backslash have to be escaped though. For a quick example, the constraint “.”$Char1 - CharNCharICharK$”” restricts the value space to the characters from *Char1* to *CharN*, *CharI*, space, and *CharK* in the encoding.

The fundamental type **`bool`** is used to represent boolean values. It differs from the other two fundamental types in several distinct ways: Its value-space is limited only to the two values `true` and `false`, there are no applicable constraints on the type, and consequently it is not possible to specialize the type.

Listing 4 provides a few type declarations as examples.

```
        uint64 = num { .[0;2^64], .(0)} //Definition of an unsigned 64-bit integer
        alphanum = char { ."a-zA-Z0-9"} //definition of an alphanumeric character.
        altbool = bool {} //This creates an alias for the fundamental type.
                                Useful if some types should not be substitutable.
```

Listing 4: Examples of how to declare new types using the fundamental types and their constraints.

The fundamental types `num` and `char` are included as their value-spaces allows for the definition of common types such as strings, integers, and real numbers. The type `bool` was made a fundamental type – even though it could have been a pre-defined enumerated type instead – due to its common usage in many programming languages and to assist eventual code generation efforts. Other types were considered, but rejected as they could be defined using the available types and elements or were too complex for the current intentions of ACS.

Beside fundamental types, an ACS implementation can, as mentioned previously, include pre-defined types, though these must be definable using the DSL itself, and are therefore mostly limited to expressing subsets of the existing value-spaces (It is possible to create new value-spaces using enumeration, but this comes with its own limitations, as explained in Section 2.4.2.2). The intention behind pre-defined types are to be able to define types used across multiple models without having to regularly re-define them. No pre-defined types are included in ACS as a standard. Pre-defined types can be used in type definitions in an identical manner to fundamental types, by calling them by name.

2.4.2.2 Enumeration

In addition to the pre-defined value-spaces provided by the fundamental types, a user can define an enumeration type, which has a custom value-space. When defining such a type, it is therefore necessary manually define the values of the value-space, which must be a set of identifiers. To specify the structure of an enumeration type, list the values inside of a pair of curly-brackets like this: `{value1, value2, ..., valueN}`. Listing 5 provides an example of an enumeration's declaration.

```
        state = {on, off, stand-by, erroneous}{}
```

Listing 5: Example of how to define an Enumeration type. Note that both the structure and constraints require curly-brackets in this case.

Enumeration types have a unique constraint, where it is possible to limit the value-space of the type to a specified subset. This can be useful if different systems can output different values, which should otherwise be treated the same. The syntax of the constraint is to write the intended subset inside a pair of curly-brackets. An example of this constraint is shown in Listing 6.

```
        drill_state = state {{on, off}} //Limits value-space to on and off
```

Listing 6: Example of how to use a constraint on an enumeration type. Notice the extra set of curly-brackets used as part of the enumeration constraint.

2.4.2.3 Records

A "**`record`**" is a composite data structure that a user can specify using the DSL, which is composed of a set of named, typed fields. Records can be used to reduce the number of variables floating around in the ACS model and emulate data structures used by communication protocols or the modeled system, which are more complex than the fundamental types.

There are two methods for defining the type structure of a record. The first is a list of fields encapsulated in a pair of parenthesis. Meanwhile, the second method is very similar but includes a **`base`** record. This second method serves to extend an existing record, which allows the user to specify new fields, as well as new constraints on both the old and new fields.

To set a constraint on a field, it is necessary to preface the constraint with the field's name. If the record as a base, then the constraints must be prefaced by "._base" before naming the field to constrain. If one wishes to constrain the base's base, then one must repeat "._base" twice instead, and so on.

Listing 7 contains two example records, one for each method of defining the structure. The first one is the `Message` record from the running example, illustrating how to define a record, while the second example is an extension of the `Message` record, which illustrates how to do so, as well as how to apply constraints on a base's fields.

```
Message = (           // A record
    String message,   // A field with type "String"
    Urgency urgency,  // A field with type "Urgency"
    num? deadline     // A field with type "nullable number"
) {
    .message._elem."a-zA-Z ",  // Constraint array element values
    .message.[[1;100]],        // Constraint array size
    .deadline.[0;+inf],        // Constraint number-value
    .deadline.(0)              // Decimal precision = '0' (integer)
}

Message2 = (Message,
    int8 messageSize,
    String sender
) {
    ._base.deadline.[0;2^32],   //A constraint on a base field
    .sender._elem."a-zA-Z "     //A constraint on a new field.
}
```

Listing 7: Repeat of running example, as well as example of extending a record.

2.4.2.4 Arrays and Nullable

The two final elements of the type system are **`array`** and **`nullable`**. These are appended to the tail of a type's structure, and convey a structural effect on everything to the left of it, which is called its **`element type`**. When constraining an array or nullable type, it is possible to add additional constraints to the element type by writing **`._elem`** before the specific constraint. This is similar to using **`._base`**.

An **`Array`** is used to indicate that the type definition will represent multiple values of the element type at once, in an array structure. This is indicated by adding an "[]" to the structure. It is possible to make the array multidimensional by adding additional square bracket pairs to the structure. Arrays have a **`length`** constraint, which makes it possible to define a minimum and maximum number of elements for each dimensions. This is done using an interval notation similar to that of `num`'s `range` constraint, but with a range for each dimension, all encapsulated by a pair of brackets. A range must be provided for each dimensions, but both the lower and higher ends can be left empty, if no boundary is necessary. For an example, see Listing 8. This two-dimensional array of int32s can have up to 5 arrays of int32 in its outermost dimension, and must have between 8 and 25 int32 in each of those arrays.

```
dim2int32 = num[,] {._elem.(0) ._elem.[-2^31;2^31-1] .[[;5], [8;25]]}
```

Listing 8: Example of constraining a 2-dimensional array and its element type. The array's first dimension can have at most 5 entries, while the second dimension must have between 8 and 25 entries.

There is a single case were the usage of array is enforced, which is that of the many-to-many invocation. This invocation requires that the input be an array(s) of the target event's input type(s), which means that the invocation's input type must have precisely one array-dimension more than the event specifies. Note that ACS cannot ensure that there will be enough elements in the input, if a constraint has been placed in said array's length. The current version will simply assume that there will be enough.

A **`nullable`** type means that there may not be a value to (part of) a variable. Since this will only be possible in some contexts – and not all contexts will be able to handle null-values – this must be manually defined on the relevant types. This is indicated by adding a "?" to the structure of the type definition. There exists no unique constraints for `nullable`.

Since both `array` and `nullable` affect every part of the structure to the left of their usage, the order that they themselves is used is important. A nullable array of integers and an array of nullable integers are two very different types. Listing 9 contains a few examples of type declarations making use of array and nullable, as well as the combination thereof.

```
nuint64 = uint64?{} //A nullable uint64

anuint64 = nuint64[]{} //An array of nuint64

asnuint64 = nuint64[,]{} //A 2-dimensional array of nuint64

nauint64 = uint64[]?{} // a nullable array of uint64

nullableArrayUint64 = uint64?[]{} //an array of nullable uint64
```

Listing 9: Examples of nullable types and array types. Notice that the order of nullable and array change the meaning of the structure.

2.4.3 Propagation of Types

Many types will find their usage spread across multiple systems as they are made part of messages send around. It is important for the systems to have the type available. One option would be to have every type available everywhere, but that would be to discard an opportunity for verification and could interfere with code generation for systems with limited resources. This, therefore, leads to the creation of the type propagation mechanism. This mechanism is intended to be run before the type checker, which will allow the latter to also check if the specified types are available and if there are any ambiguity about which types to use.

The propagation mechanism functions by *copying* types from systems that hold them to systems that do not, if they fulfill a set of criteria. This allows those systems to use the type anywhere in their behavioral model. If a type does not propagate to a system, then that implies that the relevant data never reaches said system, which informs the user about the nature of the model. A type has what is called a *fully-qualified name*, which is composed of a chain of system names, describing the genealogy (i.e. the SoI, composites, and finally AS) of the type. This is the formal name of said type. However, in order to ease the textual burden on the user, it is not necessary to use the fully-qualified name of a type, only as much as is necessary to avoid ambiguity. Multiple systems can define types with the same names. If multiple of these propagate towards a system, then said system must specify in further detail which of said types it intends to use.

In order for a type to propagate from one system to another, there are a few criteria. Firstly, the two systems must be connected with a link. From here, there are two options. Firstly, the holder of the type can initiate communication with the other system, and will invoke a handler or subscriber event that has the type as either input or output. If this is the case, then propagation will occur. Secondly, the other system can initiate communication with the type holder, and will invoke a handler or subscriber which has the type as either an input or output type. If that is the case, then the type will propagate. Note that while it is possible to propagate a type from a reference, it is not possible to propagate a type to one. The exact propagation process between two systems is described by the pseudocode in Algorithm 1.

Algorithm 1: Pseudocode for the function `CanPropagate`. Checks whether a type will propagate from system S to system S_{Link}. The type can either be pushed to the new system or pulled by the new system. The helper function *CanInitiateComm(s, t)* checks if there is a link l, which both s and t are connected to, and in which s has an active connection. The helper function *InvokesHandlerOrSubscriberWithType(s, t, type)* checks if system s invokes an event on system t which uses type *type*.

Input: Type *T*, host Atomic System or Reference S, receiving Atomic System or Reference S_{Link}

Output: Boolean value indicating whether propagation can occur or not.

```
1 if CanInitiateComm(S, $S_{Link}$) then
2     if InvokesHandlerOrSubscriberWithType(S, $S_{Link}$, T) then
3         return True;
4 if CanInitiateComm($S_{Link}$, S) then
5     if InvokesHandlerOrSubscriberWithType($S_{Link}$, S, T) then
6         return True;
7 return False
```

An issue, besides the propagation of individual types, is the issue of bases and fields. A type must, by necessity, have access to those types that it makes use of, in order to be available for usage. The actual check for their presence can be performed by the type checker, but the propagation mechanism must ensure that they propagate alongside the derived type. This process is called **`piggy-backing`**. The propagation mechanism must, therefore, be extended to handle this situation.

The propagation algorithm will start by selecting a random system in the model, and attempt to propagate every type declared in its type document. For each type, it will check every connected system, and use the above-described criteria to decide whether to propagate or not. If it propagates to a system, it will recursively attempt to propagate further from said system. It cannot propagate to a system that already has the type. If the type propagates to a system that has types that inherit from said type, it will then propagate those types first. If propagating all derived types that could be reached, it will then propagate the original base type to each system that the derived types have propagated to.

This ensures that all derived types which can be validly reached by the base type will finish their own propagation before the original will continue. The base has been propagated to these new systems, and can now propagate further from these as well. This allows for broader usage of types, as well as the extraction of data from variables.

This expansion to the propagation mechanism is described by Algorithms 2 and 3. The former describes the outer-most loops of the propagation algorithm and is what is initially executed when starting the mechanism. The latter algorithm describes the recurring function `Propagate` which is recursively called by itself and handles the propagation of derived types and the piggy-backing process itself.

Algorithm 2: Pseudocode outermost part of the propagation algorithm.

Input: ACS model *M* and its set of dependencies D

```
foreach $AS \cup REF\ S \in M$ do
    foreach $Type\ T \in AS.type\_doc.Select(x \rightarrow x.propagation == No)$ do
        Propagate(T, S);
    end
end
return;
```

Algorithm 3: Pseudocode for the function `Propagate`. Propagates scoped types to other systems. Ensures the most specialized/derived will finish propagating first. The helper function $T_S.Uses(T)$ checks whether a type T_S uses the type T as an ancestor or in a field.

Input: Type *T*, Atomic System or Reference S

```
T.Add(S);
S.Add(T);
T.Propagation = STARTED;
foreach $Type\ T_S \in S.Type\_Doc.Select(x \rightarrow T_S.Uses(T))$ do
    if $T_S.Propagation == NO$ then
        Propagate($T_S$, S);
    else if $T_S.Propagation == STARTED$ then
        throw CycleError($T_S$, T)
    foreach $System\ S_P \in T_S.Systems$ do
        Propagate(T, $S_P$);
    end
end
foreach $System\ S_{Link} \in S.ConnectedSystems.Select(x \rightarrow T \notin x)$ do
    if $CanPropagate(T, S, S_{Link})$ then
        Propagate(T, $S_{Link}$);
end
return;
```

2.4.4 High-Level Features and Limitations

Section 2.3.4 describes how the behavioral model can model the transfer of data between any sender and receiver, while being agnostic to the capabilities of the transfer medium and communicating parties. As such, the data model should be equally agnostic to the structure of data and the methods of transfer, while still being able to represent as many concepts as possible.

ACS' type system is intended to represent data/information which can be archived in different formats and used for communication, rather than physical objects themselves. Thus, the behavioral model is also only intended to model the transfer of data, rather than the movement of physical objects. Concepts that require "gradual/continuous transfer" such as fluids or electricity are especially not suited to be represented by this type system.

2.4.5 The Old Data Model

The data model has been heavily reworked since the initial publication of ACS. This section is therefore intended to exposit on the historical state of the model, and how it has changed. The section can be freely skipped if such information is of no interest.

The goal of the old data model was to create *Data Transfer Objects* (DTOs) using other user-defined DTOs and a set of *pre-defined* types. These pre-defined types were slightly similar to the current concept of fundamental types, but had different complications. The set of pre-defined types was composed of 6 element types, 3 collection types, and 2 special types.

Element	Collection	Special
bool	list	byte-sequence
byte	set	void
char	tuple	
int		
real		
string		

Table 2.12: The pre-defined types of the old data model, divided into categories.

Table 2.12 lists the set of pre-defined types. As can be observed, the choice of pre-defined types is very programming-oriented, and contains some redundancy. The decision concerning which types were included was based on a mix between what was deemed necessary to cover most types of messages and what was necessary from a code-generation perspective. In order to use a collection type, it was necessary to indicate its element type using syntax similar to C#. The special type `void` could not be set as a variable type, but was instead used to indicate that there were no input or output for some event, function, or machine.

DTOs were similar to the current records, being struct-like containers, with named fields of static types. The type of a field could be either a pre-defined type or another DTO. Differing from the current, textual method, DTOs were specified using a single, simplified class diagram to indicate composition and inheritance. There were no official method for indicating which atomic system a type belonged to, and despite them being defined in a centralized location, there were no global types in the old model.

Propagation existed in nearly the same state as in the current version, with the rules for when a single type could propagate being the same. The difference is instead in the surrounding algorithm. The old algorithm did not handle the piggy-backing of types, other than ensuring that specializations had access to their base and field types, and thus did not allow using said types for further communication.

A final change was in the typing of events and their components. In the old data model, events were still fully typed, but functions, sub-machines, and consequently variables were all loosely typed, to various degrees. Functions had only their output typed, and the input type was thus dependent on the type of the variables being used as input on the first usage. Similarly, the typing of sub-machines were fully dependent on variables, with their input typing being dependent on the first usage – as with functions – and the output typing being dependent on the type of the variables being output by the machine. While the typing of variables have not fundamentally changed and is still dynamic rather that static, the more rigorous typing is intended to increase a user's overview. Due to the potential for using the same variable names for different types, the prior type system included enforcement of assigning a type to variables in the symbol table only once.

2.5 High-Level Verification and Analysis

The goal of high-level verification is to prove the following three properties about an ACS model: 1) all Time Constraints can be satisfied, 2) all branches of communication can be reached, and 3) there is a SoS instance that satisfies properties (1) and (2). If these properties hold true, it means that all communication will be feasible in at least one implementation (or instantiation) of the modeled SoS, given that the modeled communication and constraints can be achieved in practice.

ACS is a framework that assists the SoS modeling process and helps identify high-level errors in communication logic. It is not intended to be a self-contained SoSE methodology or similar, and it never attempts to verify any concrete details about a SoS such as the actual data communicated or the correctness of internal system behavior. Furthermore, ACS cannot tell whether a model is representative of reality or a designer's intentions.

Based on Time Constraints, communication branches, and Cardinalities, ACS aims at computing ratios between the numbers of system instances that – when adhered to by an instantiation/implementation – would lead to valid communicative behavior. These ranges will help the designer determine whether the current design supports the practical loads planned for the SoS.

2.5.1 Possible Approaches

The previous iteration of ACS [12] provides a mapping to UPPAAL [7], where the Controller and Communicators for every Atomic System are mapped to a UPPAAL template and multiple queries. The ACS model is valid if the UPPAAL model satisfies all queries. The mapping can handle models where Atomic Systems have any Z-Cardinality and Composites have a 1-Cardinality (references are unsupported). Also, there is an error when mapping a fork from a Controller where each choice has different Time Constraints.

High-level verification is currently based on UPPAAL's semantics, but we plan to make separate, formal semantics to separate ACS from other formalisms. Furthermore, these semantics must support all (combinations of) ACS constructs. There appear to be multiple ways to achieve this, but more time and research than available are needed to produce a working approach. The rest of this section outlines some general ideas.

We could iterate through the possible configurations of system instances in an order that guarantees to eventually visit all configurations. For each configuration, trace all possible communication chains (see Section 2.1.2) and ensure that it is possible to satisfy all time constraints in these and that all transitions are traversed at least once. If both conditions are satisfied, the model is valid. However, this approach cannot automatically terminate as it cannot determine when all plausible configurations have been exhausted.

Alternatively, all three properties mentioned above depend on each other in some way, but by making certain initial assumptions, we could split the verification into smaller parts and combine them later. For example, by assuming that all communication branches are reachable, it could be possible to make static time safety analysis based only on Time Constraints, which if it fails, guarantees that the model is invalid. The individual parts may be incorrect or inaccurate on their own, but combining the parts should correct this.

3 Syntax and Semantics

The previous chapter served as an introduction to every feature in ACS, and present everything using concrete and informal descriptions. The purpose of this chapter is to serve as a formal introduction to the syntactic components of ACS, the validation of ACS models, and how the execution of an ACS model should be understood.

Firstly, the chapter will present the two different syntaxes which collectively serve as the formal syntax of the ACS framework. These give a name to the individual syntactic components of the ACS language and their compositions.

Secondly, the chapter will present the validation and type system of ACS models, which are used to check the semantic meaning of an ACS model and to validate the communication and data flow, respectively.

Thirdly, the chapter will present a small semantic, which expresses how an ACS model should be understood, and what the various components would do, if a model was executed.

Lastly, the chapter presents an UML profile, which maps the ACS language onto UML.

Section 3.1 presents information about the notations used throughout the following chapter. Sections 3.2 and 3.3 describes the mathematical model and the abstract syntax of ACS, which detail the relationship between components and precisely define the contents of these, respectively. Section 3.4 defines how an ACS model should be understood, and how the various syntactic elements affect said understanding. Section 3.5 presents rules for validating the semantic meaning of an ACS model's structural and behavioral models, and Section 3.6 will present the same for type and token usage. Finally, Section 3.7 describes the ACS UML profile.

3.1 Conventions

The following sections of this chapter all concern themselves with presenting the syntax and semantics of ACS. This requires some notations to convey said information. The goal of this section is to discuss the notation choices made, and present some of the recurring notations in a single section.

3.1.1 Syntax notation

As with any language, there are different sets of requirements of a syntax, depending on its use case. A user-directed (or **concrete**) syntax should be easy to learn, read, write, and understand. To accommodate such, ACS has been designed to use as minimal a syntax as possible – given its functionality – and with a focus on quickly being able to gain an overview and understanding of models. The latter has led to the diagram-centric nature of ACS, while the former has led to the minimalist and terse nature of the textual definitions used.

However, this creates problems with the specification of ACS' language components, their semantics, and their validation rules. We are not aware of any methods for directly specifying any of these for a diagram. To draw on existing knowledge bases and help future users more easily understand any such specifications, we introduce two alternative, textual, specification-oriented definitions.

The first definition is mathematically-based and is used to define (nearly) every component that exists in the ACS language, and how they relate to each other. This definition is based around the structure of an ACS model, using nested components to create a model-tree. It is intended less to carry a full specification on its own, but instead to support the understanding of the second definition. This mathematical definition is described in full detail in Section 3.2.

The second definition is an abstract syntax in the form of a context-free grammar. This definition was chosen as it allows for the construction of a parse tree (and consequently, an abstract syntax tree), which opens up several opportunities for specifying semantic rules. The syntax is expressed in a notation similar to Backus–Naur form, but with the addition of a vector notation for describing lists. This addition is not necessary to describe ACS but instead exists to simplify the eventual semantics, though it does have an effect the available options for formally expressing syntactic rules. The context-free grammar is detailed in Section 3.3.

3.1.2 Semantic notation

With the decision to use an abstract syntax based on context-free grammars, the obvious choice for a semantic would be an operational semantic of some sort, either big-step or small-step and apply that to an AST. However, there are some issues with the usage of vectors instead of recursion for handling lists.

Our first solution was therefore to propose an alteration to big-step semantics, which would enable it to process vectors. The specification for this alteration can be seen below.

$$\overrightarrow{Env_1} \vdash \left[\overrightarrow{SYN}, \overrightarrow{Env_2}\right] \longrightarrow \overrightarrow{Env_2'} \iff \overrightarrow{Env_1} \vdash \left\langle SYN^{(i)}, \overrightarrow{Env_2^{(i)}} \right\rangle \longrightarrow \overrightarrow{Env_2^{(i+1)}}$$

$$\textbf{where } i \in [0\,;\, n-1] \textbf{ and } n = \left|\overrightarrow{SYN}\right|$$

$$\textbf{and } \overrightarrow{Env_2^{(0)}} = \overrightarrow{Env_2} \textbf{ and } \overrightarrow{Env_2'} = \overrightarrow{Env_2^{(n)}}$$

This alteration describes how a rule should treat a vector. Using a vector is synonymous with iteratively processing each element and generating new, updated environments which can be used in the processing of the following element until all elements of the vector have been processed, and the final iteration of the environments has been generated.

An example of this new notation in use is shown below in rule *S1-SoI*, which is the first of several rules needed to validate a SoI-node in the AST.

[S1-SoI]

$$\frac{env_{sc}' \vdash \left[\overrightarrow{TDC},\ env_T\right] \longrightarrow_{S1} env_T' \quad env_{sc}' \vdash \left[\overrightarrow{SSM},\ env_S, env_T'\right] \longrightarrow_{S1} (env_S', env_T'') \quad env_{sc}' \vdash \left[\overrightarrow{LNK},\ env_S'\right] \longrightarrow_{S1} env_S''}{env_{sc} \vdash \left\langle \texttt{SoI}\,\texttt{(}name_{SoI}\texttt{)}\,\texttt{[}\overrightarrow{name_{INT}}\texttt{]<}\overrightarrow{name_{LNK}}\texttt{>\{}\overrightarrow{TDC}\texttt{,}\ \overrightarrow{LNK}\texttt{,}\ \overrightarrow{SSM}\texttt{\}},\ env_S, env_T \right\rangle \longrightarrow_{S1} (env_S''', env_T'')}$$

$$\textbf{where } env_{sc}' = env_{sc}.Enter(name_{SoI})$$

$$\textbf{and } \overrightarrow{qname_{INT}} = env_{sc}'.Qualify(\overrightarrow{name_{INT}}) \textbf{ and } \overrightarrow{qname_{LNK}} = env_{sc}'.Qualify(\overrightarrow{name_{LNK}})$$

$$\textbf{and } env_S''' = env_S''\left[name_{SoI} \mapsto {}_{SoI}[\bullet, \overrightarrow{qname_{LNK}}, \overrightarrow{qname_{INT}}]\right]$$

$$\textbf{and } env_S''[name_{SoI}] \mapsto NULL$$

$$\textbf{and } \forall qn \in \overrightarrow{qname_{INT}}.\left(env_S'''[qn] \mapsto {}_{AS}[\ldots]\right) \textbf{ and } \forall qn \in \overrightarrow{qname_{LNK}}.\left(env_S'''[qn] \mapsto {}_{LNK}[\ldots]\right)$$

The semantic rule *S1-SoI* is expected to be one of the simpler ones that need to be written. Using big-step semantics with the proposed vector-notation, it specifies the first step of how to validate a stand-alone model. However, the rule quickly becomes very complex, and it is expected that all other rules would be equally-as or more complex, as they have to validate more complex behavior. This One potential solution would be to remove the list-notation from the abstract syntax in order to enable the usage of normal big-step semantics. However, while this should reduce the overall complexity of the individual rules, it would simply spread out across a larger number of rules. Additionally, this leads to a second problem. ACS models are very context-sensitive, with components depending on details of a large number of other components above, below, and surrounding the target node in the AST. Neither big-step nor small-step semantics is ideal for handling these cases, as it would lead to very large side-conditions and very convoluted rules.

As an alternative to using the list-aware operational semantics, we instead decided on using two forms of pseudocode. Both are intended to use the abstract syntax as an AST, and operate upon it, although they may not necessarily be used in that precise context. The first form of pseudocode is intended to be used to specify high-level behavior, or when including low-level details would divert attention away from the intended takeaway. It replaces these details with helper functions, which textually define the intended behavior, using a combination plain English and references to concepts explained in Chapter 2. Section 3.4 uses this notation. The second form of pseudocode is intended to be used when it is necessary to describe low-level details. It uses a notation similar to C# or java so that behavior can be explicitly detailed step-for-step. It does make usage of some smaller helper functions, when not using them would shift the focus onto the wrong parts of the behavior. This style of pseudocode is used in Sections 3.6 and 3.5.

3.2 Mathematical Definition

The mathematical definition is one of two formal definitions of ACS' components. The mathematical definition and the abstract syntax are intended to be companions to each other as each describes the components using different notations, which capture differing details. The mathematical definition is object-oriented and makes use of referential objects in order to describe relationships between components. The definition is based on one presented in our first report on ACS[30], which has been modified to reflect the changes made since its publication. This includes accounting for the new communication mechanisms, tokens, and the new type system.

A component definition is composed of the object's name, a combined short-hand and set-name, and its constituent components. A constituent component may represent a nested component or indicate a relationship to said component. The exact nature of these relationships is not defined by the mathematical definition, and are instead described by the abstract syntax model (See Section 3.3) or the operational semantics (See Section 3.4). The mathematical definition instead denotes the connectivity of a full model. Note that individual objects do not have an identifying name, but are instead referred to by their usage, similar to object-oriented programming. An example of this is shown in Table 3.1 for '*System-of-Interest*', which contains two sets of links. The second of these sets is intended to indicate which links from the first set are boundary links.

Tables 3.1 & 3.2 makes use of a unique notation to designate certain concepts. Referential objects are indicated by encapsulating the object in parenthesis (I.e. "`(SoI)`"). A lowered 's' (I.e. $\texttt{SoI}_s$) denotes a set of components rather than a single one. A vector arrow (I.e. $\overrightarrow{SoI}$ denotes a list of elements. Notation elements encapsulated in single-qoutes (I.e. '*') indicate a terminal. The notation will be reiterated in the following subsections during the explanations of each component.

Due to the large number of components, the definition is split into two. The first part, explained in Section 3.2.1, will detail the "*Primary*" components, which are used to form the rough shapes of the structural and behavioral models. Section 3.2.2 will detail the more numerous "*Secondary*" components, which are used to define the details of a model. Components will be referred to using their shorthand throughout these sections.

3.2.1 Primary Components

The primary components of ACS are those that are diagrammatic in nature. Table 3.1 provides definitions for the primary components.

Model	=	`MODL`	=	$((SoI,\ SoI_s)_s)$
System-of-Interest	=	`SoI`	=	$(SSM_s,\ LNK_s,\ AS_s,\ LNK_s,\ TYP_s)$
Link	=	`LNK`	=	$((AS, CAR, CTP)_s)$
Composite	=	`COMP`	=	$(SSM_s,\ AS_s,\ LNK_s,\ LNK_s,\ CAR)$
Atomic System	=	`AS`	=	$(CON,\ EV_s,\ TYP_s,\ CAR)$
Reference	=	`REF`	=	$(SoI,\ CAR)$
Controller	=	`CON`	=	$(CONS,\ CONS_s,\ ATS_s)$
Action Transition	=	`ATS`	=	$(CONS,\ CONS,\ ACT)$
Communicator	=	`COM`	=	$(LNK,\ \overrightarrow{TKT},\ \overrightarrow{TKT},\ CC_s,\ MAC)$
Machine	=	`MAC`	=	$(COMS,\ COMS_s,\ ITS_s)$
Invocation Transition	=	`ITS`	=	$(COMS,\ COMS,\ IVK)$

Table 3.1: Formal definitions of the primary components of ACS. The list is separated into structural and behavioral components.

The table is divided into sections, grouping the components based on which category (the recurring structural, behavioral, data divisions) they belong to. There are no primary data components. The **MODL** component is unique, as it does not belong to any category, model, or layer, and instead exists above all of them.

- A **MODL** object consists of a set of **SoI** dependency declarations (SoI, $\text{SoI}_S)_S$). Each dependency declaration consists of the System-of-Interest (SoI) and its set of dependencies (SoI_S). All dependencies must have a dependency declaration of their own.

The primary structural components relate to the model in the following ways:

- A **SoI** object consists of its set of internal sub-systems (SSM$_S$), its set of links (LNK$_S$), the subset of the subsystems which serve as interfaces for the SoI (AS$_S$), the subset of links which serve as boundary links (LNK$_S$), and the set of global type declarations.
- A **LNK** object consists of a set of link connections ((AS, CAR, CTP)$_S$). A link connection consists of the connected port (AS), the cardinality (CAR), and the communication type (CTP).
- The **COMP** object consists of its set of internal sub-systems (SSM$_S$), its set of links (LNK$_S$), the subset of the subsystems which serve as interfaces for the composite (AS$_S$), the subset of links which serve as boundary links (LNK$_S$), and the composite's cardinality (CAR).
- An **AS** object consists of a controller (CON), a set of event declarations (EV$_S$), a set of local type declarations (TYP$_S$), and the atomic system's cardinality (CAR).
- A **REF** object consists of the referenced system (SoI) and the cardinality of said system (CAR).

The primary behavioral components relates to the model in the following ways:

- A **CON** object consists of an initial controller state (CONS), a set of controller states – including the initial state – (CONS$_S$), and a set of action transitions (ATS$_S$).
- An **ATS** object consists of a *start* controller state (CONS), an *end* controller state (CONS), and an action (ACT).
- A **COM** object consists of a connected link (LNK), a vector of input-tokens ($\overrightarrow{TKT}$), a vector of output-tokens ($\overrightarrow{TKT}$), a set of communicator constructs (CC$_S$), and a main machine (MAC).
- A **MAC** object consists of an initial communicator state (COMS), a set of communicator states – including the initial state – (COMS$_S$), and a set of invocation transitions (ITS$_S$).
- An **ITS** object consists of a start communicator state (COMS), an end communicator (COMS) state, and an invocation (IVK).

3.2.2 Secondary components

The secondary components represent purely textual components. As a result, the mathematical components are very close to the concrete syntax, in order to carry the necessary information. Additionally, secondary components can vary widely and have large value-spaces. Therefore, they the secondary components have an additional piece of notation. The '|' indicates an "either" choice, e.g. in **OCOM** = $COM \mid \varepsilon$, OCOM can be either a COM or nothing. Table 3.2 shows the secondary components. It divides the secondary components into five sections grouped together based on their usage.

The secondary structural components relate to the model in the following ways:

- A **SSM** component can be either an atomic system (AS), a composite (COMP), or a reference (REF).
- A **CTP** component can be either the "active" terminal, the "reactive" terminal, or the "reflexive" terminal.
- A **CAR** component can be any number in the set of positive integers ($\mathbb{Z}^+$), the 'N' terminal, the '*' terminal, or left blank (ε).

The secondary event components relate to the model in the following ways:

- An **EV** component can be a publisher (PUB), a handler (HAN), an initiator (INI), or a subscriber (SUB).
- A **PUB** object consists of a vector of input types ($\overrightarrow{TYP}$).
- A **HAN** object consists of a vector of input types ($\overrightarrow{TYP}$), a vector of output types ($\overrightarrow{TYP}$), and optionally a communicator (OCOM).
- An **INI** object consists of a set of input types ($\overrightarrow{TYP}$), a set of output types ($\overrightarrow{TYP}$), and a communicator object (COM).
- A **SUB** object consists of a publisher to listen for (PUB), a vector of output types ($\overrightarrow{TYP}$), and optionally a communicator.
- An **OCOM** component can be either a communicator (COM) or nothing at all.

The secondary controller components relate to the model in the following ways:

- A **CONS** object can be either a vector of tokens ($\overrightarrow{TK}$), or a vector of tokens ($\overrightarrow{Tk}$) and a 'lockable' terminal.
- An **ACT** component consists of a vector of tokens ($\overrightarrow{TK}$), a delay or an event (DEV), a controller state (CONS), a comparator terminal (CMP), a vector of timing constraints ($\mathbb{N}$ $UNIT$, and a vector of output tokens ($\overrightarrow{TK}$).
- A **DEV** component can be an initiator (INI), a handler (HAN), a subscriber (SUB) and target publishing system (SST), or a delay symbolized with the terminal ε.

Sub-system	=	**SSM**	=	$AS \mid COMP \mid REF$
Communication type	=	**CTP**	=	$\texttt{'active'} \mid \texttt{'reactive'} \mid \texttt{'reflexive'}$
Cardinality	=	**CAR**	=	$\mathbb{Z}^+ \mid \texttt{'N'} \mid \texttt{'*'} \mid \varepsilon$
Event	=	**EV**	=	$PUB \mid HAN \mid INI \mid SUB$
Publisher	=	**PUB**	=	$(\overrightarrow{TYP})$
Handler	=	**HAN**	=	$(\overrightarrow{TYP}, \overrightarrow{TYP}, OCOM)$
Initiator	=	**INI**	=	$(\overrightarrow{TYP}, \overrightarrow{TYP}, COM)$
Subscriber	=	**SUB**	=	$(PUB, \overrightarrow{TYP}, OCOM)$
Optional Communicator	=	**OCOM**	=	$COM \mid \varepsilon$
Controller State	=	**CONS**	=	$(\overrightarrow{TK}) \mid (\overrightarrow{TK}, \texttt{'lockable'})$
Action	=	**ACT**	=	$\overrightarrow{TK}\ DEV\ CONS\ CMP\ \overrightarrow{\mathbb{N}\,UNIT}\ \overrightarrow{TK} \mid \texttt{'!'}$
Delay and Event	=	**DEV**	=	$INI \mid HAN \mid SUB\ SST \mid \varepsilon$
Subscriber Target	=	**SST**	=	$TK \mid \texttt{'ALL'}$
Comparator	=	**CMP**	=	$\texttt{'<'} \mid \texttt{'<='} \mid \texttt{'='} \mid \texttt{'>='} \mid \texttt{'>'}$
Unit	=	**UNIT**	=	$\texttt{'d'} \mid \texttt{'h'} \mid \texttt{'m'} \mid \texttt{'s'} \mid \texttt{'ms'} \mid \texttt{'us'}$
Communicator State	=	**COMS**	=	$(\overrightarrow{V}, \overrightarrow{TK}) \mid (\overrightarrow{V\ FS}, \overrightarrow{SSM\ TKE\ AS}) \mid (\,)$
Token Extractor	=	**TKE**	=	$\mathbb{Z}^+ \mid \texttt{'ALL'}$
Communicator Constructs	=	**CC**	=	$(\overrightarrow{TYP}, \overrightarrow{TYP}) \mid (MAC, \overrightarrow{TYP}, \overrightarrow{TYP})$
Invocation	=	**IVK**	=	$\overrightarrow{V\ FS}\ LOC\ OUT \mid \overrightarrow{V\ FS}\ CC\ \overrightarrow{V} \mid V\ \overrightarrow{V\ FS}$
				$\mid \texttt{'lock'}\ AS\ ID \mid \texttt{'unlock'}\ AS\ ID$
Location	=	**LOC**	=	$PUB \mid SSM\ ID\ AS\ HAN$
Identity	=	**ID**	=	$\mathbb{Z}^+ \mid TK \mid {'}\texttt{ALL}{'}$
Output	=	**OUT**	=	$\overrightarrow{V} \mid {'}\texttt{SKIP}{'} \mid {'}\texttt{WAIT}{'}$
Variable	=	**V**	=	(TYP)
Field Selection	=	**FS**	=	$TYP\ FS \mid \varepsilon$
Type	=	**TYP**	=	$FT \mid (\overrightarrow{\texttt{EE}}) \mid TYP\,{'}?{'} \mid TYP\,{'}\texttt{[]}{'}$
				$\mid (TYP, CST_s) \mid (BASE, \overrightarrow{TYP})$
Fundamental Type	=	**FT**	=	$\texttt{'num'} \mid \texttt{'char'} \mid \texttt{'bool'}$
Base	=	**BASE**	=	$TYP \mid \varepsilon$
Constraint	=	**CST**	=	$\overrightarrow{PTH}\ PPT$
Path	=	**PTH**	=	$TYP \mid \texttt{'base'} \mid \texttt{'elem'}$
Property	=	**PPT**	=	$\mathbb{R}_1\ \mathbb{R}_2 \mid \mathbb{N} \mid \overrightarrow{\mathbf{Char}_1, \mathbf{Char}_2} \mid \overrightarrow{\texttt{EE}}$
Token Type	=	**TKT**	=	$(SSM, AS, \texttt{'single'}) \mid (SSM, AS, \texttt{'group'})$
Token	=	**TK**	=	(TKT)

Table 3.2: The formal definitions of the secondary components of ACS. The list of declarations is separated into structural, event, controller, communicator, and type components.

- An **SST** component can be either a token (TK) or the terminal '*'.
- A **CMP** component can be any of the less-than ('<'), less-than-or-equal ('<='), equal ('='), greater-than-or-equal ('>='), or greater-than ('>') terminals.
- A **UNIT** component can be any of the day ('d'), hour ('h'), minute ('m'), second ('s'), millisecond ('ms'), or microsecond ('us') terminals.

The secondary communicator components relate to the model in the following ways:

- A **COMS** object can consist a vector of variables ($\overrightarrow{V}$) and a vector of tokens ($\overrightarrow{TK}$), or it can consist of a vector of variables and field selections ($\overrightarrow{V\ FS}$) and a vector of subsystems, token extractors, and port atomic systems ($\overrightarrow{SSM\ TKE\ AS}$), or it can be an empty object.
- A **TKE** component can be any positive integer number ($\mathbb{Z}^+$) or the terminal 'ALL'.
- A **CC** object can be one of two options. The first is a function object which consist of a vector of input types ($\overrightarrow{TYP}$) and a vector of output types ($\overrightarrow{TYP}$). The second option is a sub-machine object ("(MAC, $\overrightarrow{TYP}$, $\overrightarrow{TYP}$)"), which consist of a corresponding machine (MAC), a vector of input types ($\overrightarrow{TYP}$), and a vector of output types ($\overrightarrow{TYP}$). Note that both function and sub-machine are components in ACS, but have been suburdinated by the Communicator Construct component for the sake of the mathematical definition.
- An **IVK** component has a composition corresponding to each type of invocation.
  - It can consist of a vector of variables and field selections ($\overrightarrow{V\ FS}$), a location (LOC), and an output (OUT).
  - It can consist of a vector of variables and field selections ($\overrightarrow{V\ FS}$), a communicator construct (CC), and a vector of output variables ($\overrightarrow{V}$)
  - It can consist of a declared variable (V) and a vector of variables and field selections ($\overrightarrow{V\ FS}$).
  - It can consist of a 'lock' terminal, a target atomic system (AS), and an identifier (ID) to select target instances.
  - It can consist of a 'unlock' terminal, a target atomic system (AS), and an identifier (ID) to select target instances.
- A **LOC** component can be a publisher (PUB), consist of a port (SSM AS) and a handler (HAN), or consist of

- An **ID** component can any positive integer number ($\mathbb{Z}^+$, a token (TK), or the terminal '`ALL`'.
- An **OUT** component can be a vector of variables ($\overrightarrow{V}$), the '`SKIP`' terminal, or the '`WAIT`' terminal.
- A **V** object consists of a type (TYP).
- A **FS** component can be either a type and another field selection (TYP FS) or blank (ε).

The secondary type components contain the unique element **EE**. This component represents user-defined values of an enumerated type, and thus cannot have a deterministic composition in this definition. The type components relate to the model in the following ways:

- A **TYP** object has a composition corresponding to each type element.
  - It can consist of a fundamental type (FT).
  - It can consist of a vector of enumerated values ($\overrightarrow{EE}$).
  - It can consist of another type object (TYP) and the '?' terminal.
  - It can consist of another type object (TYP) and the '`[]`' terminal.
  - It can consist of a type to constrain (TYP) and a set of constraints (CST_S).
  - It can consist of a list of field types ($\overrightarrow{TYP}$) and optionally a base type (BASE).
- A **FT** component can be either a '`num`' terminal, a '`char`' terminal, or a '`bool`' terminal.
- A **BASE** component can be either a type (TYP) or left blank.
- A **CST** component consists of a a vector of paths ($\overrightarrow{PTH}$) and a property (PPT).
- A **PTH** component can be a type (TYP), a '`base`' terminal, or a '`elem`' terminal.
- A **PPT** component has a composition corresponding to each constraint.
  - It can consist of two real numbers ($\mathbb{R}_1\mathbb{R}_2$).
  - It can consist of any non-negative integer ($\mathbb{N}$)
  - It can consist of a vector of paired characters ($\overrightarrow{Char_1\ Char_2}$).
  - It can consist of a vector of enumerated elements ($\overrightarrow{EE}$).

- A **TKT** object can consist of a port (SSM, AS) and the '`single`' terminal, or it can consist of a port (SSM, AS) and the '`group`' terminal.
- A **TK** object consists of a token type.

3.3 Abstract Syntax

The abstract syntax is the second of ACS' two syntactic definitions. This syntax is a lot more detail-heavy than the mathematical definition, and is intended to be used by the semantic specifications, but there is a strong resemblance between the two definitions. The syntax can be used for specification due to the ability to use it to form a parse tree (or more specifically, an "*Abstract Syntax Tree*"), for which rules can be specified for the individual tree-nodes. This section does not intend to detail the precise meaning of each non-terminal and production, which are instead left up to Chapter 2 and Sections 3.4 and 3.5. However, the intended meaning and usage of each production can be gleamed by comparing them with the concrete syntax.

The syntax is a context-free grammar, written in notation nearly identical to Backus-Naur form (BNF), but with an added vector-notation for lists. This notation has been included to simplify the semantics, but the syntax can be rewritten into BNF (although in that case it would benefit from being written in extended Backus-Naur form (EBNF) instead) [31]. The grammar requires an LL(3) parser in order to be parsed, due to the construction of certain non-terminals that will be detailed later in this section. The grammar can be rewritten into an LL(1) grammar (This can be performed at the same time as the conversion to BNF) by splitting and combining the responsible non-terminals into a number of new ones.

This section is divided into groups based on the syntax component's usage. Each group is further divided into two part. The first part introduces all of the syntactic categories that are a part of said group. The second part introduces the rules for each non-terminal. A rule is composed of one or more production, which are the various options for a definition provided by the rule. Elements of syntactic categories may be used before their formal introduction. Non-terminals are referred to by their name. Terminals highlighted in blue, to identify them. So as an example, take a loot at the definition for the model symbol:

$$MODL := \overrightarrow{\texttt{(}\overrightarrow{name_{SoI}}\texttt{)}\ \texttt{:}\ SoI}$$

In this definition '**(**', '**)**', and '**:**' are terminals, while '$name_{SoI}$ and SoI are non-terminals.

There is an attempt at systematically using terminals. **[** and **]** are used to indicate that whatever name symbol is in-between is for a variable. **<** and **>** are used to indicate that whatever name symbol is in-between is for a token or that the contents are related to links. `(` and `)` and `{` and `}` are used more nebulously, and with all of these having some usages inherited from the concrete syntax.

There are a few syntactic categories that do not belong to any grouping, but are instead used throughout this section. These are the **Name** and **Qname** categories shown in Syntactic Categories 1.

Syntactic Categories 1

$name \in$ ***Name***	*The category of names*
$qname \in$ ***QName***	*The category of qualified names*

Where we have that: **Name** $\subset$ **QName**. To quickly refresh, the name of a component is the one directly written by a user, while the qualified name contains the entire path from the SoI to the component.

Finally, the some non-terminals make use of the following sets:

- $\mathbb{N} = \{0, 1, 2, \ldots\}$. The set of natural numbers. Not to be confused with the cardinality `N`.
- $\mathbb{Z}^{+} = \{1, 2, 3, \ldots\}$. The set of positive integers.
- $\mathbb{R} = \{n \mid n \text{ is a number}\}$. The set of real numbers.
- **Char** is the set of all characters.

Structure

This group of syntactic elements are those used to specify the structural layer of a model. The syntactic components introduced and detailed in this grouping are detailed in Syntactic Categories 2.

Syntactic Categories 2

$MODL \in$ ***Model***	*The category of models*
$SoI \in$ ***SoI***	*The category of systems of interest*
$COMP \in$ ***Comp***	*The category of composite systems*
$AS \in$ ***AS***	*The catategory of atomic systems*
$REF \in$ ***Ref***	*The category of references*
$LNK \in$ ***Link***	*The category of links*
$SSM \in$ ***SubSys***	*The category of sub-systems*
$PORT \in$ ***Port***	*The category of system-port selectors*
$CAR \in$ ***Car***	*The category of cardinalities*
$CTP \in$ ***ConTyp***	*The category of connection types*

The definition of the corresponding non-terminals has been divided into two. Grammar Rules 1 defines those structural non-terminals that provide shape of a model, and which were previously diagrammatic in nature, while Grammar Rules 2 defines those remaining components which do not belong to another grouping, but are made use of by this grouping.

Grammar Rules 1

$$
\begin{aligned}
MODL &:= \overrightarrow{\mathtt{(}\overrightarrow{name_{SoI}}\mathtt{)}\ \mathtt{:}\ SoI} \\
SoI &:= {}_{\mathtt{SoI}}\mathtt{(}name_{SoI}\mathtt{)[}\overrightarrow{name_{INT}}\mathtt{]<}\overrightarrow{name_{LNK}}\mathtt{>\{}\overrightarrow{TDC}\mathtt{,}\ \overrightarrow{LNK}\mathtt{,}\ \overrightarrow{SSM}\mathtt{\}} \\
COMP &:= {}_{\mathtt{COMP}}\mathtt{(}name_{COMP}\mathtt{,}\ CAR\mathtt{)[}\overrightarrow{name_{INT}}\mathtt{]<}\overrightarrow{name_{LNK}}\mathtt{>\{}\overrightarrow{LNK}\mathtt{,}\ \overrightarrow{SSM}\mathtt{\}} \\
AS &:= {}_{\mathtt{AS}}\mathtt{(}name_{AS}\mathtt{,}\ CAR\mathtt{)\{}\overrightarrow{TDC}\mathtt{,}\ \overrightarrow{EDC}\mathtt{,}\ CON\mathtt{\}} \\
REF &:= {}_{\mathtt{REF}}\mathtt{(}name_{REF}\mathtt{,}\ CAR\mathtt{)\{}name_{SoI}\mathtt{\}} \\
LNK &:= {}_{\mathtt{LNK}}\mathtt{(}name_{LNK}\mathtt{)\{}\overrightarrow{\mathtt{(}PORT\mathtt{,}\ CAR\mathtt{,}\ CTP\mathtt{)}}\mathtt{\}}
\end{aligned}
$$

The **MODL** non-terminal is the start symbol of any ACS model. This rule covers the necessary syntax to designate the vector of dependency declarations.

Notice that the list of systems that the SoI is dependent on are listed before the SoI's own syntax. This is a repeated structure throughout the abstract syntax, where smaller, less complex symbols have been moved to the front, while heavier, more complex syntax has been moved to the back of a production.

Another repeated structure can be seen in **SoI**, **COMP**, **AS**, and **REF**, where there is a strict enforcement in the order in which all sub-components of a specific type (I.E. all type declarations of a **SoI**) must be constructed before moving on to link constructions, which must be completed before finally getting to subsystem construction.

Grammar Rules 2

$$\begin{aligned} SSM &:= AS \mid REF \mid COMP \\ PORT &:= qname_{AS} \mid qname_{SSM}\,\texttt{[}name_{AS}\texttt{]} \\ CAR &:= \mathbb{Z}^{+} \mid \texttt{N} \mid \texttt{*} \mid \varepsilon \\ CTP &:= \texttt{reactive} \mid \texttt{active} \mid \texttt{reflexive} \end{aligned}$$

Most of the rules in Grammar Rules 2 covers parts of the concrete syntax, which were already textual, and does not vary much from the concrete syntax as a result, other than being divided into a set of rules. This is a repeated pattern throughout the abstract syntax.

The two productions of the **Port** rule require at least an LL(2) parser, and is thus is one of the rules that would have to change in a rewrite. This could be done by combining the two productions, as they share the *qname* non-terminal, and use either another rule or EBNF to construct the optional interface specifier.

The abstract syntax for **CAR** still features the empty cardinality for links, as while it is syntactic sugar, ACS does not specify a default value, so a placeholder must be added instead.

Events

This grouping of syntactic categories and rules concern themselves with event declarations. It is a small grouping as communicators and all relevant syntax belong to another category. Syntactic Categories 3 contains those syntactic categories defined by this grouping, and Grammar Rules 3 contains the rules for the syntactic elements.

Syntactic Categories 3

$EDC \in$ **EDecl**	*The category of event declarations*
$OCOM \in$ **OptCom**	*The category of internal and explicit communicators*

Grammar Rules 3

$$
\begin{aligned}
EDC := & \ \texttt{initiator}\ name_{INI}\ \texttt{::}\ \overrightarrow{TYP_{IN}} \rightarrow \overrightarrow{TYP_{OUT}}\ COM \\
& \mid \texttt{handler}\ name_{HAN}\ \texttt{::}\ \overrightarrow{TYP_{IN}} \rightarrow \overrightarrow{TYP_{OUT}}\ OCOM \\
& \mid \texttt{subscriber}\ name_{SUB}\ \texttt{::}\ qname_{PUB} \rightarrow \overrightarrow{TYP_{OUT}}\ OCOM \\
& \mid \texttt{publisher}\ name_{PUB}\ \texttt{::}\ \overrightarrow{TYP_{IN}}
\end{aligned}
$$

$$OCOM := \texttt{\{internal\}} \mid COM$$

All **EDC** productions are unambigious from each other due to the unique starting terminal. As the corresponding concrete syntax is textual, it has been lifted for these rules.

Controllers

This grouping covers the broad number of syntactic elements required to construct a controller and all of its dependent components. This includes states, transitions, actions, and timing-related elements. Syntactic Categories 4 defines the relevant syntactic categories and components.

The rules presented in Grammar Rules 4 highly resemble the equivalent component definitions from the mathematical definition, and are designed using the same ideas. States and transitions are in lists as it makes them easier to iterate though, although this is not entirely ideal for more context-sensitive analysis, but the issue is handled in an early state of the semantic analysis (See Section 3.4.1).

Syntactic Categories 4

$CON \in \textbf{\textit{Cont}}$	*The category of controllers*
$ATS \in \textbf{\textit{ATran}}$	*The category of action transitions*
$CONS \in \textbf{\textit{Cons}}$	*The category of controller states*
$ACT \in \textbf{\textit{Action}}$	*The category of actions*
$DEV \in \textbf{\textit{Dev}}$	*The category of delays and events*
$SST \in \textbf{\textit{SubTgt}}$	*The category of subscriber targets*
$CMP \in \textbf{\textit{Cmp}}$	*The category of comparators*
$UNIT \in \textbf{\textit{Unit}}$	*The category of time units*

Grammar Rules 4

$$\begin{aligned} CON &:= \texttt{\{}\overrightarrow{CONS}\texttt{,}\ \overrightarrow{ATS}\texttt{\}} \\ ATS &:= \texttt{(}name_{CONS\_START}\texttt{,}\ name_{CONS\_END}\texttt{,}\ ACT\texttt{)} \end{aligned}$$

All of the syntax rules in Grammar Rules 5 are heavily based on textual parts of the concrete syntax, which much of the construction is lifted from, and has then been adapted to the grammar's form. **CONS** and **DEV** are two other rules which productions require an LL(2) parser in order to avoid ambiguity when parsing it. With respect to **DEV**, then there are no unique identifiers to differentiate what component a name belongs to.

Grammar Rules 5

$$\begin{aligned} CONS &:= name_{CONS}\texttt{(}\overrightarrow{name_{TK}}\texttt{)} \mid name_{CONS}\texttt{(lockable,}\ \overrightarrow{name_{TK}}\texttt{)} \\ ACT &:= \overrightarrow{name_{TK\_IN}}\ \texttt{:}\ DEV\ \texttt{[}name_{CONS}\ CMP\ \overrightarrow{\mathbb{N}\ UNIT}\texttt{]}\ \texttt{:}\ \overrightarrow{name_{TK\_OUT}} \\ DEV &:= name_{EV} \mid name_{SUB}\texttt{(}SST\texttt{)} \mid \varepsilon \\ SST &:= name_{TK} \mid \texttt{all} \\ CMP &:= \texttt{<} \mid \texttt{<=} \mid \texttt{=} \mid \texttt{>=} \mid \texttt{>} \\ UNIT &:= \texttt{d} \mid \texttt{h} \mid \texttt{m} \mid \texttt{s} \mid \texttt{ms} \mid \texttt{us} \end{aligned}$$

Tokens

The tokens grouping is smallest and least complicated one, with no outstanding details to chronicle. Syntactic Categories 5 contains the syntactic category of tokens, and Grammar Rules 6 contains the rule for the **TKT** non-terminal.

Syntactic Categories 5

$TKT \in \textbf{\textit{TokTyp}}$ *The category of token types*

The **TKT** rule contains the production '`void` ' as

Grammar Rules 6

$$TKT := {}_{\texttt{SINGLE}}\texttt{(}PORT\texttt{)} \mid {}_{\texttt{GROUP}}\texttt{(}PORT\texttt{)} \mid \texttt{void}$$

Communicators

This grouping contains the rules and non-terminals relating to communicators, and any elements that such may use, such as machines, states and transitions, invocations, and communicator constructs. Syntactic Categories 6 contains the long list of syntactic categories that are defined by this grouping.

Syntactic Categories 6

$COM \in$ ***Commu***	*The category of communicators*
$MAC \in$ ***Mac***	*The category of machines*
$ITS \in$ ***ITran***	*The category of invocation transitions*
$COMS \in$ ***Coms***	*The category of communicator states*
$VSL \in$ ***VarSel***	*The category of variable selectors*
$TKS \in$ ***TokSel***	*The category of token selectors*
$ISL \in$ ***InsSel***	*The category of instance selectors*
$CC \in$ ***ComCon***	*The category of communicator constructs*
$IVK \in$ ***Invok***	*The category of invocations*
$OMM \in$ ***OMVar***	*The category of "one or many messages"*
$OMT \in$ ***OMTgt***	*The category of "one or many targets"*
$TGT \in$ ***Target***	*The category of invocation targets*
$LOC \in$ ***Loc***	*The category of locations*
$ID \in$ ***Ident***	*The category of instance identifiers*
$OUT \in$ ***Out***	*The category of output configurations*
$VAL \in$ ***Value***	*The category of anonymously defined values*

The three rules defined by Grammar Rules 7 all resemble the structure used by the mathematical formula, but with some terminals injected to separate various elements and indicate an end to their construction.

Grammar Rules 7

$$COM := \texttt{<} name_{LNK} \; \texttt{::} \; \overrightarrow{TKT_{IN}} \texttt{→} \overrightarrow{TKT_{OUT}} \texttt{>\{} \overrightarrow{CC}, MAC \texttt{\}}$$
$$MAC := \texttt{\{} \overrightarrow{COMS} \texttt{,} \; \overrightarrow{ITS} \texttt{\}}$$
$$ITS := \texttt{(} name_{COMS\_START} \texttt{,} \; name_{COMS\_END} \texttt{,} \; IVK \texttt{)}$$

As the syntactic elements defined by the rules in Grammar Rules 8 concern themselves with the textual invocations, this part of the abstract syntax has been lifted from the concrete syntax, where applicable, and adapted. The rules for **COMS** and **TGT** both require an LL(3) parser to unambiguously parse, as some of their productions are identical identical until the third symbol is reached. As with earlier cases, this can be rewritten to an LL(1) by fusing the common parts, and making any differences into optional constructions using EBNF.

Grammar Rules 8

$$
\begin{aligned}
COMS &:= name_{COMS} \mid name_{COMS}\texttt{[}\overrightarrow{name_{VAR}}\texttt{]<}\overrightarrow{name_{TK}}\texttt{>} \mid name_{COMS}\texttt{[}\overrightarrow{VSL}\texttt{]<}\overrightarrow{TKS}\texttt{>} \\
VSL &:= name_{VAR}\overrightarrow{\texttt{.}name_{FLD}} \\
TKS &:= qname_{AS}\texttt{(}ISL\texttt{)} \mid qname_{SSM}\texttt{(}ISL\texttt{)[}name_{PORT}\texttt{]} \\
ISL &:= \mathbb{Z}^{+} \mid \texttt{all} \\
CC &:= \texttt{function}\ name_{FUN}\ \texttt{::}\ \overrightarrow{TYP_{IN}} \rightarrow \overrightarrow{TYP_{OUT}} \\
&\quad \mid \texttt{machine}\ name_{MAC}\ \texttt{::}\ \overrightarrow{TYP_{IN}} \rightarrow \overrightarrow{TYP_{OUT}}\ MAC \\
IVK &:= \overrightarrow{VSL_{IN}}\ \texttt{:}\ name_{PUB} \mid OMM\ \texttt{:}\ OMT\ \texttt{:}\ OUT \mid \texttt{lock}\ LOC \\
&\quad \mid \texttt{unlock}\ LOC \mid name_{VAR}\ \texttt{=}\ VAL \\
OMM &:= \texttt{[}\overrightarrow{VSL}\texttt{]} \mid \overrightarrow{VSL} \\
OMT &:= \texttt{[}TGT\texttt{]} \mid TGT \\
TGT &:= name_{CC} \mid LOC\texttt{.}name_{HAN} \\
LOC &:= qname_{AS}\texttt{(}ID\texttt{)} \mid qname_{SSM}\texttt{(}ID\texttt{)[}name_{PORT}\texttt{]} \\
ID &:= \mathbb{Z}^{+} \mid name_{TK} \mid \texttt{all} \\
OUT &:= \overrightarrow{name_{VAR\_OUT}} \mid \texttt{WAIT} \mid \texttt{SKIP} \\
VAL &:= \texttt{(}\overrightarrow{name_{FLD}\ \texttt{=}\ VSL}\texttt{)} \mid name_{REC\_TYP}\texttt{(}\overrightarrow{name_{FLD}\ \texttt{=}\ VSL}\texttt{)} \\
&\quad \mid \texttt{[}\overrightarrow{VSL}\texttt{]} \mid name_{ARR\_TYP}\texttt{[}\overrightarrow{VSL}\texttt{]}
\end{aligned}
$$

Types

The final grouping is of those syntactic elements relating to types and the type system. As the concrete type system is already textual and very minimal in its expression, little has changed from the concrete syntax to the abstract syntax. The syntactic categories are defined Syntactic Categories 7.

Syntactic Categories 7

$TDC \in$ ***TDecl***	*The category of type declarations*
$TYP \in$ ***Type***	*The category of (data) types*
$TBB \in$ ***TBlk***	*The category of type building blocks*
$CST \in$ ***Cons***	*The category of type constraints*
$PTH \in$ ***Path***	*The category of type (nesting) paths*
$PPT \in$ ***Prop***	*The category of constraint properties*

In Non-terminals Grouping 9, the rule for **TBB** is another which requires some level of look-ahead. In theory it should only require an LL(2) parser to handle any ambiguity, but due to the recursive nature of the rule, it is possible that more are required, if a number of array-elements are needed. In the third construction of the rule for **PPT**, $REGX$ refers to the syntax for single-character matching from Regular Expressions.

Grammar Rules 9

$$
\begin{aligned}
TDC &:= name_{TYP}\ \texttt{=}\ TBB\ \ \texttt{\{}\overrightarrow{CST}\texttt{\}}\\
TYP &:= TBB\ \ \texttt{\{}\overrightarrow{CST}\texttt{\}} \mid \texttt{void}\\
TBB &:= qname_{TYP} \mid \texttt{char} \mid \texttt{num} \mid \texttt{bool}\\
&\quad \mid TBB\texttt{?} \mid TBB\texttt{[]} \mid \texttt{\{}\ \overrightarrow{name_{SMB}}\ \texttt{\}}\\
&\quad \mid \texttt{(}\overrightarrow{name_{FLD}\ TBB}\texttt{)} \mid \texttt{(}qname_{BASE}\ \texttt{:}\ \overrightarrow{name_{FLD}\ \ TBB}\texttt{)}\\
CST &:= \overrightarrow{PTH}\texttt{.}PPT\\
PTH &:= \texttt{.}name_{FLD} \mid \texttt{.\_base} \mid \texttt{.\_elem}\\
PPT &:= \texttt{[}\mathbb{R}_1\texttt{;}\mathbb{R}_2\texttt{]} \mid \texttt{(}\mathbb{N}\texttt{)} \mid \texttt{"}\overrightarrow{REGX}\texttt{"} \mid \texttt{[[}\mathbb{N}_1\texttt{;}\mathbb{N}_2\texttt{]]} \mid \texttt{\{}\overrightarrow{name_{SMB}}\texttt{\}}
\end{aligned}
$$

3.4 Execution of a model

Chapter 2 goes into detail about how each individual component of ACS should be understood and how certain parts interact. However, this information is split across

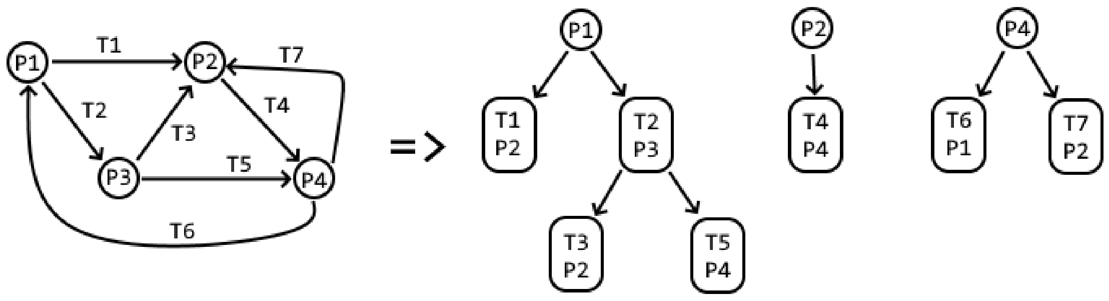


Figure 3.1: Example of a graph transition tree representation of a cyclic graph.

the entire chapter, and delivered piece-meal. The purpose of this section is therefore to provide explicit semantics for the execution of a model.

All executable behavior in an ACS model belong to Atomic Systems, which manifest in the form of the Controller and the Communicators associated with the Atomic System's Events. Therefore, in order to execute a model, it is only necessary to execute the behavior of every single Atomic System in the model concurrently.

The following sections assume that the ACS model in question has been successfully verified beforehand. Additionally, note that many decisions depend on details that have been abstracted away from the model as non-deterministic choices. An actual implementation and execution would therefore require some method for including said decision-logic.

3.4.1 Graph Transition Trees

The abstract syntax (see Section 3.3) describes Controllers and machines as lists of states and transitions, which makes iterating though them easy, but complicates analysis, context-aware validation, and operational semantics. To solve this problem, the first iteration of ACS [30] introduced the concept of *Graph Transition Trees*. The goals of these trees are to ensure that the states and transitions need to only be analyzed once and that the graph of the state-chart diagrams is easy to navigate.

This new representation transforms the graph into a forest of trees rather than the two lists of states and transitions. Figure 3.1 illustrates this representation.

Firstly, a tree is created for the initial state and each *merge state* (that is, a state where two or more transitions enter). In Figure 3.4.1, that would be the states P1, P2, and P4. Then, a child-node is added to the root for each transition that leaves the root. Child-nodes contain both the transition itself and its end-state. This is repeated for the child-nodes as well, until the end-state of a transition either has no outgoing transitions or the end-state is the root of a tree (either the initial state or a merge state).

3.4.2 Semantics for a Controller

An Atomic System will start from the initial state of its Controller. It will then non-deterministically choose transitions when appropriate or await the non-deterministic decisions of other Atomic Systems that may attempt to invoke Events on this Atomic System. A pseudocode for this behavior is shown in Algorithm 4.

Algorithm 4: Pseudocode for the execution of a controller.

Input: A controller *Con*, A dictionary mapping states in *Con* to graph transition trees *Trees*

```
Let Tree = Trees[Con.InitialState];
Let Current = Tree.Root;
while True do
    Let Paths = Current.Children;
    Let Decision = AwaitDecision(Paths); //Decision is a tree node.
    if Decision == null then
        Break;
    else
        Let Result = Execute(Decision.Transition);
        if Result is Failure then
            Break;
        else
            Current = Decision;
            if Current.Children.Count == 0 then
                Tree = Trees[Current.State];
                Current = Tree.Root;
end
```

As can be seen in the Pseudocode in Algorithm 4, a large part of the execution of a controller depends on the two helper-functions *AwaitDecision* and *Execute*.

The *AwaitDecision* function is intended as a stand-in for the missing decision logic. It is therefore non-deterministic in its execution and results. The *Await* part of the name is used to indicate that if no path is available at the current time, due to requiring prompting from another system, then the execution of the function will hold until either another system invokes an Event on one of the paths, or the timing constraint of the last available path closes. A path with a delay-Event can always be taken, but will lock the choice to said path. If no path can be taken, then the execution of this Atomic System will have failed, and with it, the entire model. It therefore terminates the execution. Theoretically, the decider could decide to fail on purpose, but that goes against the intentions of ACS, and is thus discarded as an option. The decider will always make a choice that avoids errors, if possible.

The *Execute* function is used to begin executing the delay or Event of the chosen transition path. The execution must finish according to the provided timing constraint. A less-than (<, <=) timing constraint gives an upper limit for execution time which must be kept, but the exact execution time is based on non-deterministic choices. An equal (=) timing constraint gives both an exact execution time, but can also provide an exact limit, that can be overshot and result in failure. If undershot, then the execution will wait the remainder out. A greater-than (>=, >) timing constraint provides a minimum time that the execution will take. The exact execution time is based on non-deterministic choices. If undershot, then the system will wait. Any wait-times are either true busy-waits or represent some internal computation. If overshot, then the execution will have failed, and it will report as such to the Atomic System, which will shut down, terminating the model execution as a whole.

- A *delay*-transition will wait for a set amount of time based one the provided time interval.
- An *initiator*-transition will begin executing the Communicator belonging to this Event.
- A *handler*-transition or a *subscriber*-transition will begin executing its Communicator, if it has one. If not, then it will busy-wait according to the timing constraint.

As with the *AwaitDecision* function, the *Execute* function will not attempt to intentionally fail. It will always strive to both keep the timing constraint and make non-deterministic choices that do not needlessly busy-wait.

3.4.3 Operational Semantics for a Communicator

A communicator's execution is started by the Atomic System's Controller deciding on a transition, and beginning the execution of it. It will start by attempting to execute invocations, and non-deterministically decide on branching choices. Invocations can only be executed if the target system(s)'s Controller is in a state which allows for the invocation of the specified event. Depending on the invocation, the execution may either await its completion or continue on. If the invoked element is a machine, then the flow of execution will transfer over to the new machine, and return when completing it. The execution of a communicator completes when reaching an end-state. The controller of the Atomic System may halt the execution if the execution runs over time, which leads to failure. The pseudocode for this process is described in Algorithm 5.

Algorithm 5: Pseudocode for the execution of a communicator.

Input: A communicator *Comm*, A list of token types *ITok*

Output: Output specified by event declaration of event containing *Comm*

```
Let MainMachine = Comm.MainMachine;
Let maintree be the graph transition tree of MainMachine;
Let Current = maintree[MainMachine.InitialState]
Let Variables = {};
Let Tokens = GenerateTokens(MainMachine.InitialState, Comm, ITok);
while Current.Children.Count != 0 do
    Let Paths = Current.Children;
    Let Decision = ChoosePath(Paths);
    Let Invocation = Decision.Transition.Invocation;
    if Invocation is a lock invocation then
        GetLock(Invocation.Location, Comm.Link, Tokens);
    else if Invocation is an unlock invocation then
        Unlock(Invocation.Location, Comm.Link, Tokens);
    else if Invocation is an variable declaration then
        Variables.AddRange(CreateVariable(Invocation, Variables));
    else if Invocation is an publisher invocation then
        Publish(Invocation, Comm.Link);
    else if Invocation is a function invocation then
        Variables.AddRange(InvokeFunction(Invocation, Variables));
    else if Invocation is a machine invocation then
        Variables.AddRange(ExecuteMachine(Invocation, Variables, Tokens));
    else if Invocation is a handler invocation then
        if Invocation.Targets is a Many then
            if Invocation.Inputs is a Many then
                Variables.AddRange(InvokeM2MEvent(Invocation, Variables,
                  Comm.Link, Tokens));
            else
                Variables.AddRange(InvokeO2MEvent(Invocation, Variables,
                  Comm.Link, Tokens));
        else
            Variables.AddRange(InvokeO2OEvent(Invocation, Variables, Comm.Link,
              Tokens));
    Current = Decision;
    if Current.Children.Count == 0 then
        if Trees.Contains(Current.State) then
            Tree = Trees[Current.State];
            Current = Tree.Root;
        else
            return ( Current.State.Output, Current.State.Tokens);
end
```

The algorithm introduces a set of variables on Line 4. It is used to store variable objects, and retrieve the information therein, as the invocations only contain the variable names. *Tokens* is introduced to serve a similar role for token usage.

The communicator's execution makes use of a wide range of helper-functions:

GenerateTokens(Initial state, Comm, list of inputted token types) This helper function combines the token information provided by the three parameters into proper tokens that can be used. It takes the name of tokens from the initial state, the token type from the communicator, and the precise instances to target from the inputted list provided by the controller. The generated tokens are then outputted.

ChoosePath(List of TreeNodes) This helper function takes a list of tree nodes (that is, a list of potential transitions and end-states), and non-deterministically chooses one to traverse.

GetLock(Invocation, Link, list of tokens) This helper function extracts the target system instance from the invocation, using either a token or an identifier, and attempts to lock it for future use. If it cannot be locked at the current time, then the function will wait until it is available. This may lead to failing a timing constraint.

Unlock(Invocation, Link, list of tokens) This helper function extracts the target system instance from the invocation, using either a token or an identifier, and unlocks it for usage by other instances.

CreateVariable(Invocation, List of variables) This helper function creates and returns a new variable of a type and expected value as defined by the invocation.

Publish(Invocation, Link) This helper function publishes the data held in the variables designated by the invocation, using the publisher specified in the invocation, publishing over the provided link. The published message reaches every system instance connected to the link.

InvokeFunction(Invocation, List of variables) This helper function takes the variables specified in the invocation and generates a list of output variables based on the function declaration and the names provided by the invocation. The invoked function takes a non-deterministic amount of time to execute.

ExecuteMachine(Invocation, list of variables, list of tokens, Comm) This helper function will execute another machine. The behavior of this machine is identical to Algorithm 5, but with some small changes. They do not share their variables, but they do share the tokens retrieved on Line 5. Additionally, an invoked machine cannot output new tokens either; only variables.

InvokeM2MEvent(Invocation, Link, list of variables, list of tokens) This helper function invokes an Event on another system instance connected to the communicator's link, and sends the input variables though the invocation. The precise Event, target system, and system instance are specified in the invocation. System instance can come from a token, which is why *Tokens* is included as well. When the Event terminates, it will return the data retrieved from the invocation as new variables, named according to the invocation.

InvokeO2MEvent(Invocation, Link, list of variables, list of tokens) This helper function invokes the same Event on a number of system instances connected to the communicator's link, and sends the same set of input variables to each as part of the invocation. The list of tokens is included as the target instances may be identified using a token. When the Event terminates, it will return the data retrieved from the invocation as new variables, named according to the invocation.

InvokeO2OEvent(Invocation, Link, list of variables, list of tokens) This helper function invokes the same event on a number of system instances connected to the communicator's link, but sends a custom set of input data to each as part of the invocation. The list of tokens is included as the set of target instances may be based on a prior usage. When the Event terminates, it will return the data retrieved from the invocation as new variables, named according to the invocation.

The execution of some helper functions may spawn new threads independent of the main thread described by the algorithm. *ExecuteMachine*, *InvokeFunction*, *InvokeO2OEvent*, *InvokeO2MEvent*, and *InvokeM2MEvent* all execute invocations that allows for skipping the function's termination, and continuing on while it runs. As **`SKIP`** can only be used when no new variables are generated, the functions will spawn and start the new thread, and then return to the main loop. The new thread will execute the invocation and terminate.

3.5 Validation Rules

Despite the somewhat restrictive nature of ACS's syntax, both concrete and abstract, it is still possible to construct a wide variety of syntactically correct models. However, a large subset of these models would be semantically meaningless or self-contradicting. It is therefore necessary to introduce a set of validation rules which can specify the valid subset of syntactically-correct models, which is the goal of this section.

The validation rules discussed can be seen as a formalization of the rules interspersed throughout this report, and specifically the rules introduced in Sections 2.2 and 2.3. The specification of type-related validation is described in Section 3.6.

The validation of an ACS model is divided into two phases. The first phase, the "*Independent structural validation*" phase concerns itself with validation that ACS elements are internally consistent and valid. A small example would be that a machine had the minimum amount of states and transitions. The second phase, the "*Cross-referential integrity validation*" phase concerns itself with validating that the cross-element references of an ACS model are valid. A small example of this would be that a system trying to invoke an Event on another system must share a Link that enables this communication.

The pseudo-code expressing the validation rules of an ACS model is structured according to a visitor pattern, visiting individual nodes in order to validate them. Due to the size of the pseudo-code, only snippets of it will be shown.

3.5.1 Independent Structural Validation

The primary focus of this first phase of model validation is to discover whether the individual nodes of the AST are meaningfully formed, with respect to themselves. This isolation is not absolute, and is instead more focused non-reference-based validation, as references require at least one parse in order to build a symbol table of Systems, Types, and Links. Building this table is the secondary goal of the first phase. The table takes the form of two environments, which store data about important elements in the AST, to enable easier access in the second phase. It is necessary to use the fully-qualified name to look-up information using the environments, which can limit access, if only a partial name is supplied.

The first environment is the system environment *EnvSY*, which stores data about Systems-of-Interest, Composite systems, References, and Atomic Systems. The data stored in the environment is slightly formated to enable easier look-up. The different system types have different entry-types in the environment. A System-of-Interest's entry is composed of the set of Boundary Links, the set of interface systems, the set of Links, and the set of subsystems. A composite's entry has the same structure, but with an added Cardinality. The entries for References and Atomic Systems feature some changes from their AST node counterparts, as they contain a list of Links to which the systems are connected. The second environment is the Link environment *EnvL*, which stores data about Links; the entries of which have the same form as their AST counterpart.

$$\begin{aligned}
EnvSY \vdash qname \rightarrow {} & [BND\_LNK_s,\ INT_s,\ LNK_s,\ SSM_s] && (SoI)\\
& \cup [BND\_LNK_s,\ INT_s,\ LNK_s,\ SSM_s,\ CAR] && (COMP)\\
& \cup [CAR,\ SoI,\ LNK_s] && (REF)\\
& \cup [EDCL_s,\ LNK_s,\ TDC_s,\ CAR] && (AS)\\
EnvL \vdash qname \rightarrow {} & [\overrightarrow{PORT,\ CAR,\ CTP}] && (LNK)
\end{aligned}$$

To list some examples of validation rules, this phase checks that Systems-of-Interest and Composites are Systems-of-Systems, that they have one or more interfaces, that their

```
void VisitNode(MachineNode node)
    if(node.States.length < 2)
        throw Error("...");
    if(node.Transitions.length < 1)
        throw Error("...");
    if(!node.States.Filter(state => state is a InitialState).length != 1)
        throw Error("...");
    if(!node.States.Filter(state => state is a EndState).length == 0)
        throw Error("...");
    foreach(State s in node.States)
        if (s is an InitialState)
            if(node.Transitions.Any(state => state.destination == s))
                throw Error("...");
        else if (s is a EndState)
            if(node.Transition.Any(state => state.source == s))
                throw Error("...");
        else
            if(node.Transitions.Filter(state => state.source == s).length == 0)
                throw Error("...");
            if(node.Transitions.Filter(state => state.destination == s).length == 0)
                throw Error("...");
    ...
```

Listing 10: Pseudo-code snippet showcasing part of the semantic rule validating the non-referential elements of a machine node. Error-messages have been cut due to lack of space.

indicated interfaces and Boundary Links exist, that Links have enough connected systems, that some system can initiate communication of said Link, that no more than one Link Connection connects the same Port and Link Hub, that there is at least one transition and one state in a Controller, and many more, small rules.

An example can be seen in Listing 10, which shows part of the VisitNode function for a machine node. The rule checks that there are at least two states, a transition, exactly one initial state, at least one end-state, that all states (except the initial state) has a transition which enters it, that all states have a transition which leaves it (except end-states), and more, which is not included in the snippet. Following each check, there is a detailed error-message which indicates where the error is and the precise reason behind the error.

Another, smaller example can be seen in Listing 11, which also includes the addition of a new entry to the EnvSY environment. The Atomic System validation rule is relatively simple, since most of the more customizable parts of a model occurs in its Controller and Communicators. It is enough to check that the system has at least one Event, as otherwise

```
void VisitNode(AtomicSystemNode node){
    if(node.EventDeclarations.length == 0)
        throw Error("...");
    foreach(EventDeclNode edecl in node.EventDeclarations)
        if(edecl.CommunicatorNode is not null)
            VisitNode(edecl.CommunicatorNode);

    VisitNode(node.ControllerNode);
    envSY[node.fullName] = (node.EventDeclarations,
                        node.EventDeclarations.Select(edecl => edecl.Link),
                        node.TypeDeclarations,
                        node.Cardinality
                        );
}
```

Listing 11: Pseudo-code snippet showcasing the semantic rule validating the non-referential elements of an atomic system node. Error-messages have been cut due to lack of space.

it would be unable to receive or initiate communication.

3.5.2 Cross-referential integrity Validation

The focus of the second phase of model validation is to validate the referential relations of the model. Many parts of a syntactically-valid model will refer to other components by name only. Part of this phase is to check whether these names exist, and if they are what the original system claim them to be.

A few examples would be to check that Link Connection Ports connect to actual systems, that systems targeted by an invocation exists, is connected to the specified Link, and contains the invoked Event, that all declared Events are used by a system, that all invocable Events can be invoked, that System and Link Cardinalities are compatible, that there are no islands in any Controller or Communicator, that Controllers are circular, and many more.

A small example of the pseudo-code is shown in Listing 12, where the validator checks whether every dependency actually exists somewhere in the model. However, most other validation rules are far more complex. To give an example, the validation of a Link node is a 4-phase process spanning 90 lines. The lines covering the second phase are shown in Listing 13. The focus of this phase is to check that the specified Ports of the Link exist.

```
void VisitNode(ModelNode node){
    modeledSystems = node.Dependencies.Select(dep => dep.system.name);
    foreach((SoINode system, list<string> dependencies) dependencyDecl in node.Dependencies)
        foreach(string dependency in dependencyDecl.dependencies){
            if(!modeledSystems.Contains(dependency)){
                throw Error("...");
            }
        }
        VisitNode(system);
    }
}
```

Listing 12: Pseudo-code snippet showcasing the semantic rule for validating a model node, using referential details.

```
...
set<SubsystemNode> neighborsystems = node.parent.Subsystems;
set<string> PossibleConnections;
if(isBoundary)
    neighborsystems.AddRange(node.parent.parent.Subsystems);

foreach(SubsystemNode option in neighborsystems){
    if(option is CompositeNode comp)
        PossibleConnections.AddRange(comp.Interfaces);
    else if(option is ReferenceNode ref)
        PossibleConnections.AddRange(envSY[ref.SoI].Interfaces);
    else
        PossibleConnections.Add(option.name);
}
foreach(LinkConnection LC in node.LinkConnections){
    if(LC.Port is a AtomicSystemPort){
        if(!PossibleConnections.Any(sys => sys.fullname == LC.Port.System))
            throw Error("...");
    }
    else{
        if(!PossibleConnections.Any(sys => sys.fullname == LC.Port.Interface))
            throw Error("...");
    }
    ...
```

Listing 13: A pseudo-code snippet of part of the validation rule of a link node. The snippet specifically shows the lines related to the second phase.

3.6 Type System

The type system of ACS serves the role of verifying the data flow of ACS. This involves checking whether systems have access to the types they use, checking that tokens are created, used, and destroyed according to the established rules, and that invocations receive variables with the correct types. Types refer to Data Types, where as Tokens refer to Token Types.

The type-checker can be said to be composed of three phases.

1. **Type Propagation**: The first phase is responsible for performing the type propagation algorithm described in Section 2.4.3. More precise semantics for how this phase is executed has already been covered in that section with Algorithms 1, 2, and 3.

2. **Type Usage**: The second phase involves checking that all types used by an Atomic System have correctly propagated to that system. That is, check that all input and output types are available for the Atomic System.

3. **Token Usage**: The third phase concerns itself with checking that tokens are correctly used.

4. **Type-checking invocations**: The fourth and final phase involves type-checking Communicators, to ensure that all invocations are type-correct.

Phases 2, 3, and 4 can be performed at the same time, as they mostly concern themselves with the same sections of the AST. However, the arrangement of the AST does enforce some structure on the order in which the parts relevant to each phase will be visited.

The following subsections will go into detail about each phase (beside phase 1, which has already been detailed). The pseudo-codes used to define the type checker are very long, and the sections will therefore instead feature snippets of these alongside textual explanations. All three phases make use of graph transition trees to traverse the state-chart machines, but the differences between the graphs result in different approaches to the traversal, in combination with the differing goals of each phase.

3.6.1 Validation of type usage

The process of validating all type usages requires visiting the AST nodes for all Event declarations in an Atomic System and each invocation inside the various communicators. For each of these, when encountering a new type for the first time, it should be examined for any and all nested type usages. That is, all base types and field types should be visited as well, to ensure that these type constructions are possible, following the end of the propagation.

When a new type is encountered using the above method, then the type-checker will look up its name in the list of types which the Atomic System has available. To quickly refresh, this list was built by the propagation mechanism in the previous phase, where it updated the set of local types. If a name has been partially-qualified to avoid ambiguity, then it becomes necessary to iterate through the types manually, to find which (if any) fits the partial-qualification.

If a type, at any point, isn't available, then the type-checker will throw an error message, but will not terminate the validation, as this error does not hinder further validation.

No pseudo-code snippet is provided for this phase, due to its simplicity.

3.6.2 Validation of token usage

The process of validating the token usage of an Atomic System requires visiting each state and transition in the Controller, as well as the initial state, every transition, and every end-state in the Communicator, as these are are the nodes in the AST which can make usage of a token.

The phase is divided into two sub-phases for each Atomic System, due to the differences in token usage and the separation between the two. The first focuses on validating the Controller and its related AST nodes, while the second sub-phase focuses on validating the Communicator and its related nodes.

3.6.2.1 Controller token usage

To refresh, tokens exist on Controller-states and may be outputted by Action Transitions. A token is used validly in an Action Transition when the token is present on the transition's source-state. The token's present on the destination-state of a transition must be a subset of the source-state's tokens and the tokens outputted by the transition. A pseudo-code snippet describing this can be seen in Listing 14, which shows how tokens are extracted from Actions. The Event associated with the action provides a list of output token types and which can be combined with the action's list of token names to create new token instances, which are then saved.

As the verification of elements in the graph depend on those that came before, it is necessary to traverse the graph in order. Merge-states will necessarily be encountered multiple times, and are deemed valid if every path leading to the state is able to construct the equivalent set of tokens on the state. This is described by Listing 15, which describes how the type-checker validates states. If it is the first time the state is reached, it checks

```
List<Token> AccessableTokens = StateTokens[current.State].Copy();

if(action.DEV is not PublisherNode){
    if(action.DEV.Communicator is not null){
        List<TokenTypeNode> outputtypes = action.DEV.Communicator.OutputTokens;
        List<string> outputnames = action.OutputTokens;
        if(outputtypes.Count != outputnames.Count)
            throw Error("...");
        foreach(string name in outputnames)
            if(AccessableTokens.Any(tok => tok.name == name))
                throw Error("...");
        for(int i = 0; i < outputtypes.Count; i++)
            AccessableTokens.Add(new Token(outputnames[i], outputtypes[i]));
    }
}
```

Listing 14: Pseudocode snippet describing how to compute the available set of tokens for a transition's destination-state.

that it has accessible tokens with corresponding names. If so, then it saves this set on the state. For any further visits with the state, the type-checker goes through the saved list of tokens on the state, and checks whether the list of accessible tokens produced by the path contains equivalent tokens of the same name and type.

3.6.2.2 Communicator token usage

A Communicator makes usage of tokens in three places; on the initial-state, on the end-states, and as an instance-identifier in invocations. The type-checker starts by creating token instances based on the Communicator's list of input token types and the initial state's list of token names. These tokens can then be used as identifiers throughout the main machine and the related sub-machines. The type-checker must check whether the token type correspond with the system-type that the invocation targets. Listing 16 describes the process of validating the usage of a token as an identifier.

When reaching an end-state of the main machine, it has the option of outputting a list of token selectors. These must be type-checked to be equivalent with the list of output token types defined on the Communicator. A sub-machine do not create new token instances when started, nor does it output token instances when terminating. Listing 17 describes the process of validating the end-states.

```
//Processing state for the first time
if(!StateTokens.Contains(current.State))
    List<string> tokennames = child.State.Availabletokens;
    List<Token> SurvivingTokens;
    foreach(string tokenname in tokennames){
    if(!newlyList.Any(token => token.name == tokenname))
        throw Error("...");
    else
        SurvivingTokens.Add(newlyList.SingleOrError(token => token.name == tokenname));
        StateTokens[current.State] = SurvivingTokens;
//Meeting state again
else
    foreach(Token tok in StateTokens[current.State]){
        if(!newlyList.Any(t => tok.Equals(t)))
            throw Error("...");
```

Listing 15: Pseudo-code snippet describing how the tokens on states are validated.

```
Dictionary<string, Token> Tokens;

Location loc = invocation.TargetSelector.Location;

if(loc.ID.Type is a TokenIdentifier)
    if(!Tokens.Any(tok => tok.key == loc.ID))
        throw Error("...");
    Token ID = Tokens[loc.ID];
    if(loc.port != null)
        if(ID.TokenType.Equivalent(loc.port))
            throw Error("...");
    else
        if(ID.TokenType.Equivalent(loc.system))
            throw Error("...");
```

Listing 16: Pseudo-code snippet describing how token usage in invocations are validated

```
if(node.OutputTokens.Count != child.State.OutputTokens.Count)
    throw Error("...");
for(int k = 0; k < node.OutputTokens.Count; k++){
    OutputtedTokenType = GenerateTokenType(child.State.OutputTokens[k]);
    if(OutputtedTokenType.NotEquals(node.OutputTokens[k])){
        throw Error("...");
```

Listing 17: Pseudo-code snippet describing how the output of new tokens from a communicator is validated.

```
//It is a root
if(orderedTrees.Any(t => t.root.State == child.State))
    //if first encountered, save as is
    if(!StateVariables.Contains(child.State))
        StateVariables[child.State] = AccessibleVariables;
    //If encountered for a second time
    else
        StateVariables[child.State] =
            Intersection(StateVariables[child.State], AccessibleVariables);
```

Listing 18: Pseudo-code snippet describing how the set of accessible variables is computed for merge-nodes, to avoid ambiguous variable usage.

3.6.3 Type-check of invocations

This last phase is the largest of the phases, as it involves type-checking the entirety of every Communicator, to ensure that the data-flow is valid. This involves validating the initial state, the end-states, the event invocations, the function invocations, and the machine invocations.

Type-checking the accessibility of variables in a machine is more complicated than that of tokens in the controller. As with tokens in the Controller, the set of accessible variables for a state are still those of the previous state combined with those outputted by the connecting invocation transition. However, it is necessary to know precisely which variables are unambiguously accessible at a given node, following a merge-node as not all paths to this node may feature the same variables. The type-checker does still make usage of graph transition trees, but requires a more methodological approach. The prior usages would use a queue to add encountered tree-nodes to in a breath-first order, and slowly begin to jump between trees as leaf-nodes were systematically encountered, and new trees processed. However, jumping between trees is inadvisable, as it is necessary to find all non-ambiguous variables for a merge-node before type-checking its transition tree. All transition trees for both the main-machine (and any sub-machines, when type-checking those) are topologically sorted, as they are guaranteed to be non-cyclical so a cyclic dependency is impossible. They are then in order validated. When a merge-node is processed for the first time, the type-checker simply saves its current set of variables on the state. Whenever it reaches the state again, it will instead take the intersection between the variables currently stored on the state and the list of accessible variables generated by the current path. Doing this for every possible path to the merge-node will produce the set of unambiguously-accessible variables. This is described in the pseudo-code snippet in Listing 18.

Input and output variables of a machine function are similar to how token input and output were described in the earlier subsection. Input variable objects are created by combining the names stored on the initial state with the input types defined on the Event/Communicator-construct (depending on if it is a main or sub-machine). These initial variables are then accessible in all states of the machine. For output, the designated variables are type-checked with the output types designated for the machine. As ACS supports type substitution, type-checking variables are more complicated than checking for identical types. A more complex set of rules is therefore necessary. Listing 19 contains a specification of these rules.

1. $env_T, env_{CST} \vdash T \sqsubseteq T \Leftrightarrow \forall c_f \in env_{CST}[rhs].(\exists c_a \in env_{CST}[lhs].(c_a.\overrightarrow{PTH} = c_f.\overrightarrow{PTH} \wedge c_a.PPT \mathrel{\dot{\sqsubseteq}} c_f.PPT))$
2. $env_T, env_{CST} \vdash T_a\mathbf{?} \sqsubseteq T_f\mathbf{?} \Leftrightarrow env_T, env_{CST} \vdash T_a \sqsubseteq T_f$
3. $env_T, env_{CST} \vdash T_a \sqsubseteq T_f\mathbf{?} \Leftrightarrow env_T, env_{CST} \vdash T_a \sqsubseteq T_f$
4. $env_T, env_{CST} \vdash T_a\mathbf{[\,]} \sqsubseteq T_f\mathbf{[\,]} \Leftrightarrow \forall c_f \in env_{CST}[rhs].(c_f.\overrightarrow{PTH} = \emptyset \Rightarrow \exists c_a \in env_{CST}[lhs].$
   $(c_a.\overrightarrow{PTH} = \emptyset \wedge c_a.PPT \mathrel{\dot{\sqsubseteq}} c_f.PPT))$
   $\wedge\ env_T, RemoveTopLevel(env_{CST}) \vdash T_a \sqsubseteq T_f$
5. $env_T, env_{CST} \vdash \mathbf{(}\overrightarrow{F_1}\mathbf{)} \sqsubseteq \mathbf{(}\overrightarrow{F_2}\mathbf{)} \Leftrightarrow |\overrightarrow{F_1}| = |\overrightarrow{F_2}|$
   $\wedge\ \forall x.(\overrightarrow{F_1}[x].name = \overrightarrow{F_2}[x].name$
   $\wedge\ env_T, GetFor(env_{CST}, \overrightarrow{F_1}[x].name) \vdash \overrightarrow{F_1}[x].T \sqsubseteq \overrightarrow{F_2}[x].T)$
6. $env_T, env_{CST} \vdash \mathbf{(}n_q\ \mathbf{:}\ \overrightarrow{F_1}\mathbf{)} \sqsubseteq \mathbf{(}n_q\ \mathbf{:}\ \overrightarrow{F_2}\mathbf{)} \Leftrightarrow |\overrightarrow{F_1}| = |\overrightarrow{F_2}|$
   $\wedge\ \forall x.(\overrightarrow{F_1}[x].name = \overrightarrow{F_2}[x].name$
   $\wedge\ env_T, GetFor(env_{CST}, \overrightarrow{F_1}[x].name) \vdash \overrightarrow{F_1}[x].T \sqsubseteq \overrightarrow{F_2}[x].T)$
   $\wedge\ env_T, GetFor(env_{CST}, \mathbf{\_base}) \vdash n_q \sqsubseteq n_q$
7. $env_T, env_{CST} \vdash \mathbf{(}n_{q1}\ \mathbf{:}\ \overrightarrow{F}\ \mathbf{)} \sqsubseteq n_{q2} \Leftrightarrow env_T, GetForLHS(env_{CST}, \mathbf{\_base}) \vdash n_{q1} \sqsubseteq n_{q2}$
8. $env_T, env_{CST} \vdash n_{q1} \sqsubseteq n_{q2} \Leftrightarrow n_{q1} \neq n_{q2} \wedge env_T[n_{q1}].scoped \wedge env_T[n_{q2}].scoped$
   $\wedge\ env_T, env_{CST}[lhs \mapsto env_{CST}[lhs] \cup env_T[n_{q1}].cst]$
   $\vdash env_T[n_{q1}].syn \sqsubseteq n_{q2}$
9. $env_T, env_{CST} \vdash T \sqsubseteq n_q \Leftrightarrow !env_T[n_q].scoped \wedge env_T, xpnd(env_T, env_{CST}, T, n_q) \vdash xpnd(T) \sqsubseteq xpnd(n_q)$
10. $env_T, env_{CST} \vdash n_q \sqsubseteq T \Leftrightarrow !env_T[n_q].scoped \wedge env_T, xpnd(env_T, env_{CST}, n_q, T) \vdash xpnd(n_q) \sqsubseteq xpnd(T)$

Listing 19: Formal specification of the rules for substitution of types.

3.7 UML Profile for ACS

To reap the benefits of UML's features and support, as well as to serve as a foundation for an IDE for ACS, we create a UML profile for ACS in which the ACS concepts are mapped to extensions of UML concepts. The profile is created using the software called "*Eclipse Modeling Tools v. 2022-03*" [19] (see Section 1.2), since this supports compiling the profile and many more customizations into a plugin for the "*Eclipse Papyrus v. 2022-03*" [20] UML modeling tool (see Section 1.2). Whereas this section describes the profile formally, Chapter 4 describes the implementation of the Papyrus-based IDE.

The UML profile includes all Structure, Behavior, and Data components. These parts are described separately in the following three subsections. The profile includes only few semantic rules in the form of OCL constraints. Most semantic rules are implemented separately as described in Chapter 4. It is recommended to read Chapter 2 before reading this section.

3.7.1 Structure

ACS' "Structure Diagram" is mapped onto UML's "Component Diagram", since the Component Diagram elements closely resemble the ACS components. Figure 3.2 shows this Structure mapping from ACS to UML, where the elements with colored markings show how the Structural components relate to – and uses – the Behavior and Data layers. Figure 3.2 does not show or mark the Behavioral components that use Structure components, but Figure 3.3 in the following section does.

An "**`ACS_Model`**" is an extension of the "*Model*" metaclass which is the top/root element of Papyrus projects. It contains a single root SoI and a list of SoS/SoI names that the root depends on. These are the names/SoIs that a Reference system inside the root SoI is allowed to reference.

The four system types ("**`SoI`**", "**`Reference`**", "**`Composite`**", and "**`Atomic System`**") are all extensions of the UML "*Component*" element, and "**`Port`**" and "**`Link Hub`**" are extensions of the UML "*Port*" element. The reason is that UML Components can natively have UML Ports placed on their edges/boundary, and as such closely resembles how systems have Ports and Link Hubs on their boundaries. The abstract stereotypes "**`Container`**" and "**`Subsystem`**" correspond to the categories seen in Figure 2.5 from Section 2.2.

ACS' **`Link Connection`** and **`Interface Connection`** are extensions of the UML "*Association*" element, since these are used to connect two UML Ports (and thus also ACS **`Ports`** and **`Link Hubs`**).

However, since the **`Link Hub`** stereotype extends the UML Port element, it cannot be placed

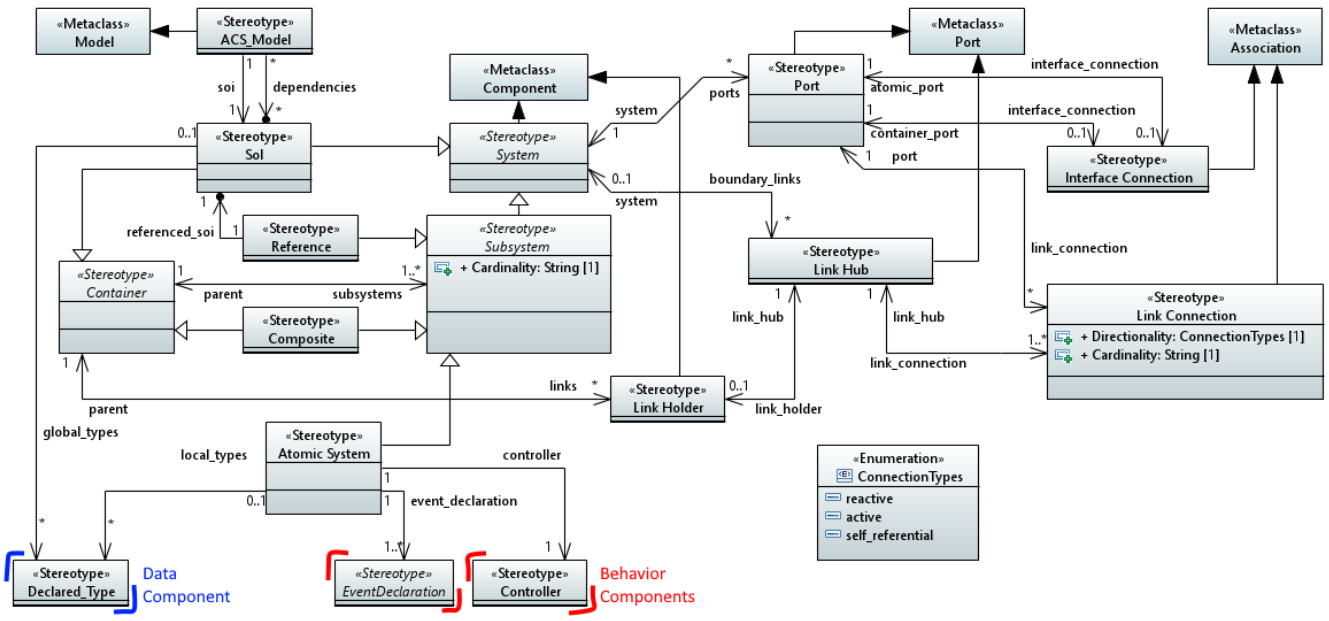


Figure 3.2: The part of the UML profile that contains the structural ACS components.

freely inside a UML Component; only on the box that surrounds the UML Component. Thus, the **`Link Hub`** stereotype cannot be used as a normal/free Link; only as a Boundary Link. We work around this limitation by making the "**`Link Holder`**" stereotype that extends the UML Component element, and which has no other purpose than to be placed inside Container System stereotypes and hold "Free Links".

3.7.2 Behavior

ACS' "Controller Diagram" and "Event Diagram" are both mapped onto UML's "State Machine Diagram", since the Behavioral ACS components map fairly cleanly onto UML's State Machine related elements. Figure 3.3 shows the Controller Diagram related stereotypes to the right and the Event Diagram related stereotypes to the left. The line between "Port" and "TokenSelector" separates these fairly well. Similar to above, profile elements with colored markings show how the Behavioral components relate to – or use – the Structure and Data layers.

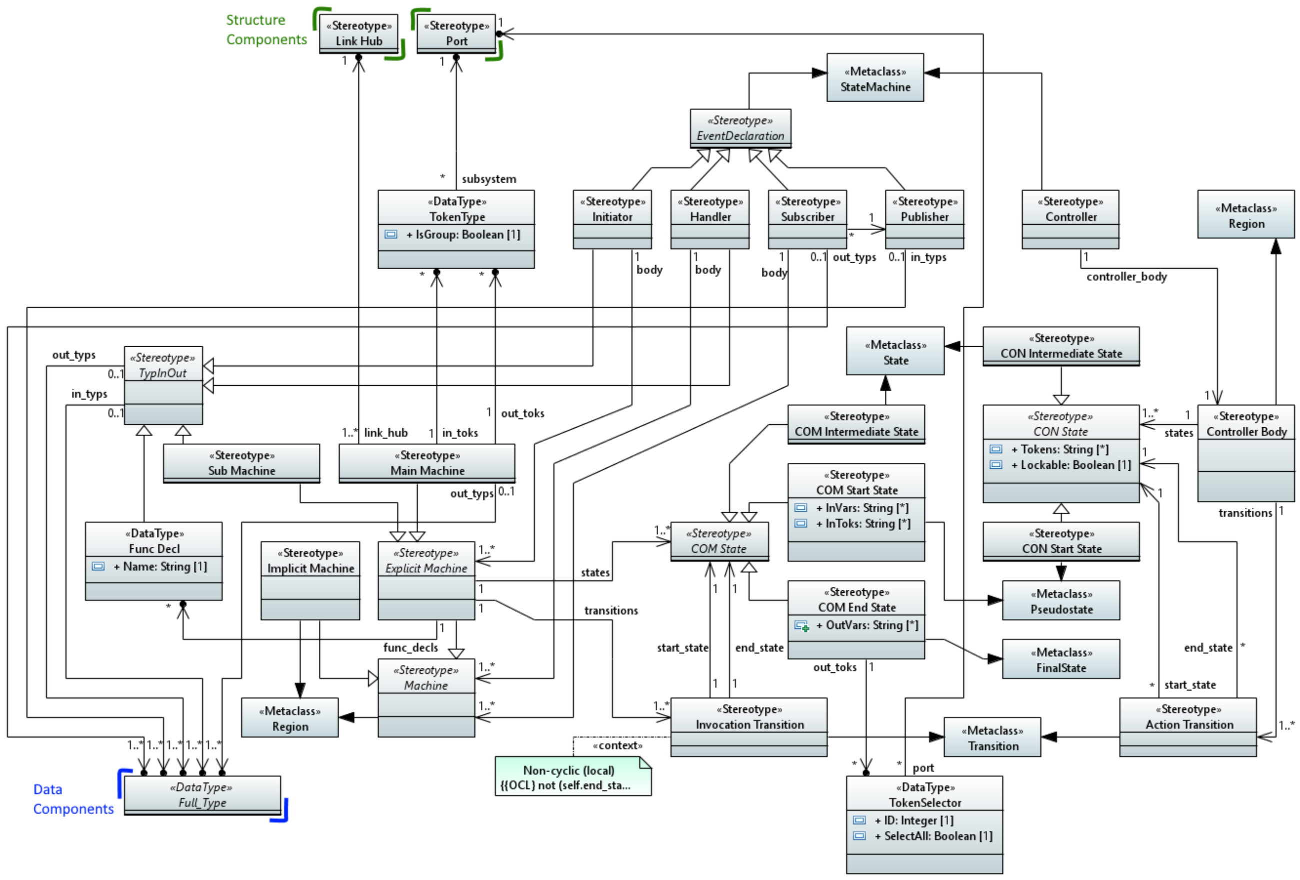


Figure 3.3: The part of the UML profile that contains the behavioral ACS components. The far-right side contains the stereotypes related to Controller Diagrams. Most of the image to the left contains the stereotypes related to Event Diagrams.

A “**`Controller`**” is an extension of UML’s “*State Machine*” element. State Machines in UML have “*Regions*” to contain (and group) machine states and transitions, so the additional “**`Controller Body`**” stereotype that extends UML’s Region element is made for this purpose. The remaining stereotypes in the Controller-section of Figure 3.3 model the Controller’s **`Start State`**, **`Intermediate State`**, and **`Action Transition`**. The **`Start State`** is an extension of UML’s “Pseudostate” to make Papyrus use the correct drawing style.

An “**`EventDeclaration`**” is an abstract stereotype that is an extension of UML’s “State Machine” element, so it serves both as an Event Declaration and an Event Diagram (or Communicator). **`EventDeclaration`** has four specializations corresponding to the four Event types. Each Event is its own stereotype since they may be structured very differently compared to each other. The abstract “**`TypeInOut`**” stereotype unifies stereotypes with both input and output types to reduce the number of associations that need to connect to the “**Full_Type**” stereotype.

Event Diagrams that are not internal (meaning they have a Communicator) all have one “**Main Machine**” and zero or more “**Sub Machines**” that are both modeled as extensions to UML’s “*Region*” element. The same goes for “**Internal Machine**” that marks an Event as internal. Since Functions in Events do not have behavior (but rather serve as placeholders for business logic), these are modeled using the “**Func Decl**” Data Type that will be configured in a textual menu. Apart from having an “**End State**”, Communicator/Event states and transitions are modeled similarly to those of the Controller.

An important detail is that the “**Action Transition**” and “**Invocation Transition**” stereotypes model their “action” and “invocation” properties as a string using the “*name*” property that is build into every UML element (even stereotypes). By using the name property, the actions and invocations will be visible in the model as a single string written over the transition.

However, this does not explicitly model the various dependencies these two stereotypes have on Tokens, Ports, Events, Machines, Functions, Fields in Records, and Declared types. Since modeling these dependencies would only further complicate the profile, and since using the name property is more user-friendly and manageable, we went with the name solution.

3.7.3 Data

ACS’ “Type Document” is mapped onto UML’s “Class Diagram”. Figure 3.4 shows the model elements related to structuring types in the bottom, those related to constraints in the top-right, and those for declaring and defining types in the top-left. Since the Data Layer does not make use of any Structural or Behavioral components, Figure 3.4 does not have any color marked elements.

Apart from the “**Declared_Type**” stereotype which extends UML’s “*Class*” element and contains an instance of “**ACS_Type**”, all other Data related concepts are modeled as UML “Data Types”. Since ACS’ type system allows anonymous types to be created (even using declared types) in-place anywhere a type is used, the profile must represent ACS types in a way that can be instantiated anywhere. UML Data Types are perfect for this since they can be configured graphically (in Papyrus at least) with values that are instantiated on a case-by-case basis.

Alternatively, types could be modeled using strings, but since the dependencies between the Data related model elements aren’t too complicated, we decided to go with an explicit model.

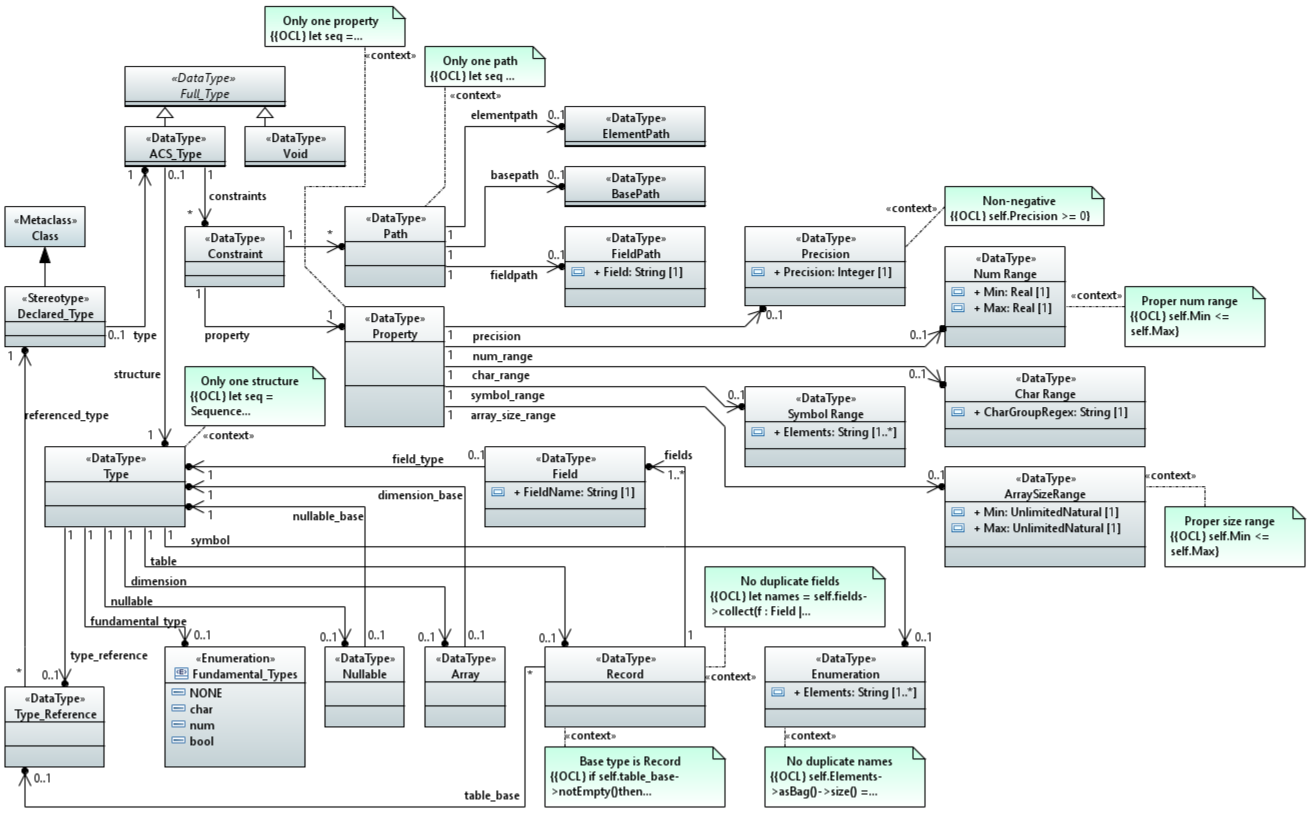


Figure 3.4: The part of the UML profile that contains the data ACS components. The bottom part describes the structure of types. The top-right part describes the structure of constraints. The top-left part describes the combined types and constraints.

While we initially wanted to model "**`Type`**" with a specialization for each specific type (numbers, arrays, etc.), Papyrus does not support using a specialized type in place of a generalized type. To work around this, we model **`Type`** as containing zero or one of every specialized type, where only one of the fields can be non-empty (checked by the "*Only one structure*" OCL constraint). This is, of course, not desirable and should be changed when possible.

The same problem/solution goes for "**`Path`**" and "**`Property`**" that are part of "**`Constraint`**". **`Path`** can only contain one of the three types of path, whereas **`Property`** can only contain one of the five properties that can be constrained. This is likewise checked by the "*Only one property*" and "*Only one path*" OCL constraints.

Since types are structured in a recursive manner (akin to "`array[nullable[integer]]`"), some specialized types contain a **`Type`** instance to continue the recursive cycle. Furthermore, **`Type_Reference`** is used to reference a type by its name, which is modeled using a reference to a **`Declared_Type`**, which is defined inside a SoS or an AS (see Figure 3.2 again).

In ACS, anonymous types can be the special type **`void`** that represents “no type”. Declared types cannot be **`void`**, and therefore **`Full_Type`** is added at the top-left of Figure 3.4 which can be either **`void`** or an **`ACS_Type`** (Figure 3.3 shows how **`Full_Type`** is used). The attentive reader might notice that this kind of specialization is the thing that Papyrus does not support, however, in the exact menus in which this field shows up, specializations *can* be used in place of their generalizations.

4 Dedicated ACS tool

This chapter explains the “why” and “how” that a dedicated ACS modeling tool was created, with the goal of supporting ACS modeling. Following that, a set of usability tests are performed with the tool, with the intention to produce insights into both the ACS tool and ACS as a concept.

While it is possible to use ACS without dedicated tools (e.g. pen and paper equivalent), we considered the benefits of building a dedicated ACS tool:

- Fast design iterations
- Source of truth
- Better support

A Software-based tool would additionally provide a convenient way to reach collaborators in the global community that might benefit from it.

There exist several open-source modeling editors[32, 20, 33], that might be utilized to build the ACS tool. These existing projects represent an opportunity to capitalize on existing code and established best practices, however, they also represent less creative freedom as the existing projects inevitably have varying restrictions from the existing code base.

While it is possible to build a tool without relying on existing projects, there would be a risk that unforeseen issues halt development or result in major flaws in the resulting product. Using existing and established tools significantly mitigates the risk, to development issues.

More than ten domain specific modeling projects (and various other projects) have some software support for their graphical notation using Eclipse Papyrus[34]. Papyrus has been supported for a long time, evident by the approximately 17.000 commits it has received since 2009[35]. We consider papyrus to be a mature and well-tested tool, suited to create the ACS tool. For this reason, the dedicated ACS tool and IDE is implemented using the Papyrus framework. The ACS IDE’s source code and binary distributions are available on GitHub [5].

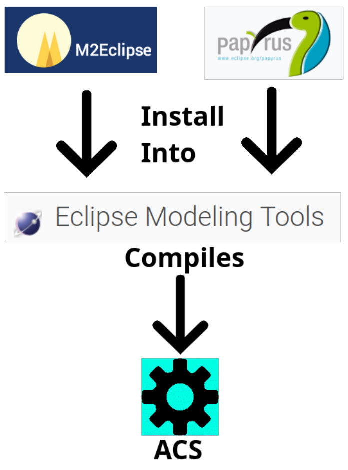


Figure 4.1: The configuration used during development of the ACS tool(EMT[19], M2Eclipse[36] and Papyrus[20].

4.1 Configuring the Eclipse Modeling Tools

Eclipse Modeling Tools (EMT), also described in Section 1.2, can be configured to support the plugin development process for papyrus. The configuration we used was to install EMT as the main IDE, then install Papyrus in the IDE using an Eclipse Update Site, and finally installing M2Eclipse using the Eclipse Marketplace. The complete configuration is shown in Figure 4.1.

This setup deviates from the "Papyrus Developer Guide"[4] from Eclipsepedia. The Developer guide suggests to use Eclipse Committers edition[37] or use an Oomph setup model[38] (A pre-configured eclipse environment). However, after encountering problems with the Oomph installer on a Linux-based OS, we used the above setup (Figure 4.1) following a recommendation from a developer of papyrus.

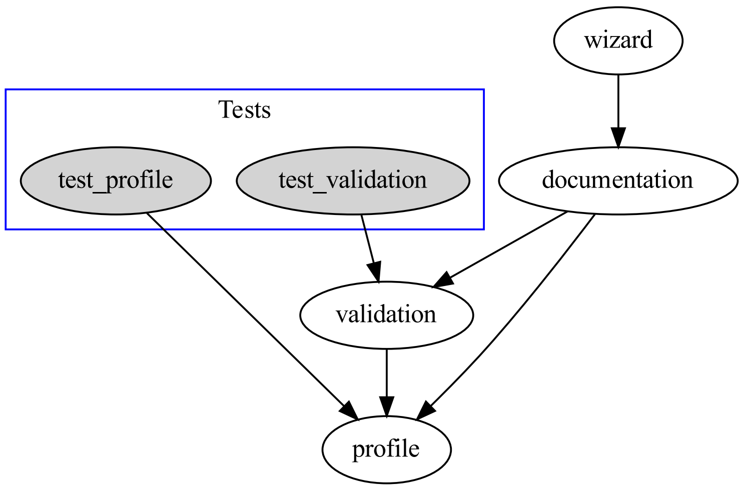


Figure 4.2: Plugin dependency graph for documentation and test plugins. An arrow denotes a dependency between two plugins by pointing towards the dependency.

4.2 Plugin Structure

The implementation is divided into different plugins. Each plugin is responsible for some unit of behavior that is unique to that plugin. While dependencies between plugins do occur, they are relatively few and explicitly defined. Documentation and testing-related dependencies are shown in Figure 4.2, a full dependency graph is included in Appendix B for completeness, but not all plugins are explained in detail.

4.2.1 ACS Profile

The profile plugin contains the model depicted in Figures 3.2/3.3/3.4. This model is used to generate java classes for each ACS object. The classes are used throughout most of the other plugins, which depend depend on the profile plugin.

4.2.2 Validation

The validation plugin contains constraint rules to validate an ACS model. The rules are applied using a “visitor-like” pattern. The constraints implement the constraint-interface

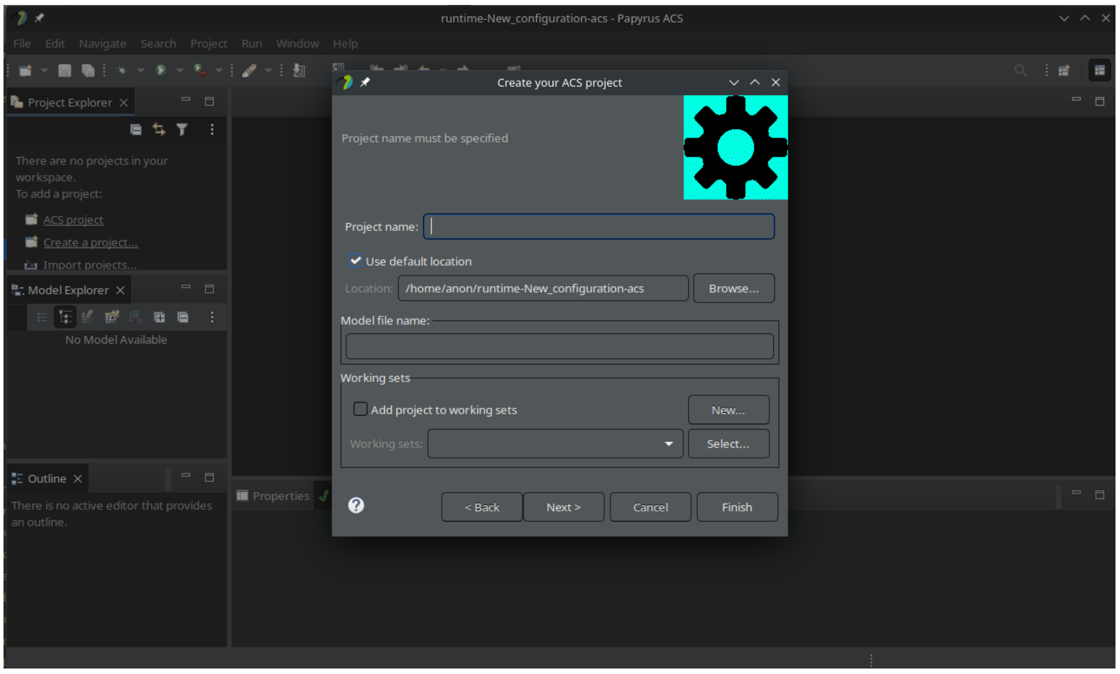
```java
1  public interface ConstraintInterface {
2
3      /*Return false if the target doesn't satisfy the constraint, true otherwise*/
4      public boolean satisfies(EObject target);
5
6      /*Returns an error MSG for the constraint
7       * If target is null the MSG will be a generic error MSG for this constraint
8       * */
9      public default String getErrorMSG(EObject target) {
10         return Utils.getMSG(this);
11     }
12
13     /*Returns a short description of the rationale behind this constraint*/
14     public default String getRationale() {
15         return "Rationale missing for now.";
16     };
17
18     /*Returns a list of Classes this constraint applies to*/
19     public LinkedList<Class<?>> appliesTo();
20 }
```

Listing 20: The ConstraintInterface that defines a constraint for validation.

shown in Listing 20, it makes validation simple for any ACS object. A manager class first checks if the constraint applies to the object (Line 19), then it checks if the object satisfies the constraint (Line 4), then it asks for any potential error messages if the object failed to satisfy the constraint (Line 9). The validation plugin depends on the profile plugin to have the class definition of the ACS objects it validates (Figure 4.2).

4.2.3 Documentation

The documentation plugin is responsible for generating documentation on the IDE. The IDE provides two documents; a “Quick Guide” and a “Full Guide”. The plugin generates the two files and places them inside the project folder upon project creation. The “Quick Guide” is a static file that is manually updated, and intended to provide a small guide to create a minimal ACS model. The “Full Guide” primarily provides a list of all possible error messages, as well as the rationale behind them. Each error message in the “Full Guide” also presents a list of ACS objects that the error can concern. The “Full Guide” is mostly generated from the same constraints that validate ACS objects. This ensures a correspondence between the validation plugin and the documentation plugin, as the documentation is in essence always up-to-date wrt. observable validation behavior. This is reflected in Figure 4.2 where the documentation plugin depends on both validation and profile plugins.



There are no projects in your workspace.
To add a project:

No Model Available

There is no active editor that provides an outline.

Project name must be specified

Project name:

[✓] Use default location

Location: /home/anon/runtime-New_configuration-acs

Model file name:

Working sets

[ ] Add project to working sets

Working sets:

Figure 4.3: A custom create project wizard from the finished ACS tool.

The wizard plugin is responsible for making custom windows in the IDE and facilitate the appropriate action from the window interaction. The wizard plugin depends on the documentation plugin (seen in Figure 4.2), as the documentation is technically generated when the “finish” button is pushed on a window managed by the wizard plugin, shown in Figure 4.3.

4.2.4 Testing

The two test plugins displayed in the square in Figure 4.2 depend only on the plugins they are testing. The test plugins contain some unit and integration tests that seek to ensure the plugin behaves as expected, the tests run when the project is compiled and provide an automated guarantee that some important functionality continues to work as expected.

4.2.5 Styling

By default ACS objects look exactly like the UML elements they are made from. This is not ideal as ACS elements intentionally look distinct from UML. We used a feature similar to Cascading Style Sheets (CSS) included in Papyrus to alter the look of the UML objects.

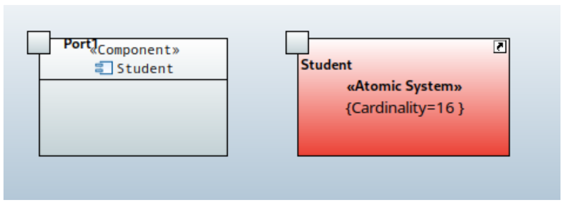


Figure 4.4: An Atomic System with Port, styled on the right, unstyled on the left.

This styling is defined in the "css" and "architecture" plugins respectively. An example of the "css" styling applied can be seen in Figure 4.4 where color, composition and labels are changed.

4.2.6 Palette

The palette plugin is responsible for presenting the relevant ACS elements in their relevant context, since ACS features four diagrams (Structure, Event, Controller and Type). The palette is presented in right side of the diagram editor in Figure 4.5.

4.3 Verification Capabilities

Implementing verification in the tool introduced additional considerations in terms of what a semantic rule should mean in the IDE context. While the formal syntax provides an absolute definition of correctness (i.e. a binary decision of inclusion in the ACS model domain), it describes no error resolution or guidance that could inform a user on a course of action upon being informed their model wasn't included in the ACS model domain (Or informally the model was faulty).

The tool introduces constraints, which apply to specific ACS objects and produce an error message for that object if the constraint fails. The purpose of the error message is to guide a user towards building a valid ACS model from ACS objects.

The verification in the IDE is made from the theoretical definitions discussed in Chapters 2 and 3. An example of this is cardinality verification. Listing 21 shows a snippet of java code that is used to verify that combinations of cardinalities are valid. Referring back to Table 2.2, there is a clear correspondence between the designed and implemented rules. The table specifies that a '*' link cardinality accepts any System cardinality, depicted in Listing 21, on lines 27-28. A 'N' link cardinality accepts only 'Z+' and 'N' System cardinalities, depicted in Listing 21, on lines 18-25. Lastly 'Z+' link cardiniality (dependent on directionality and 'Z+' concrete values) otherwise accepts 'Z+' and 'N' System cardinalities, shown on lines 3-15. This verifies that some aspects of communication are

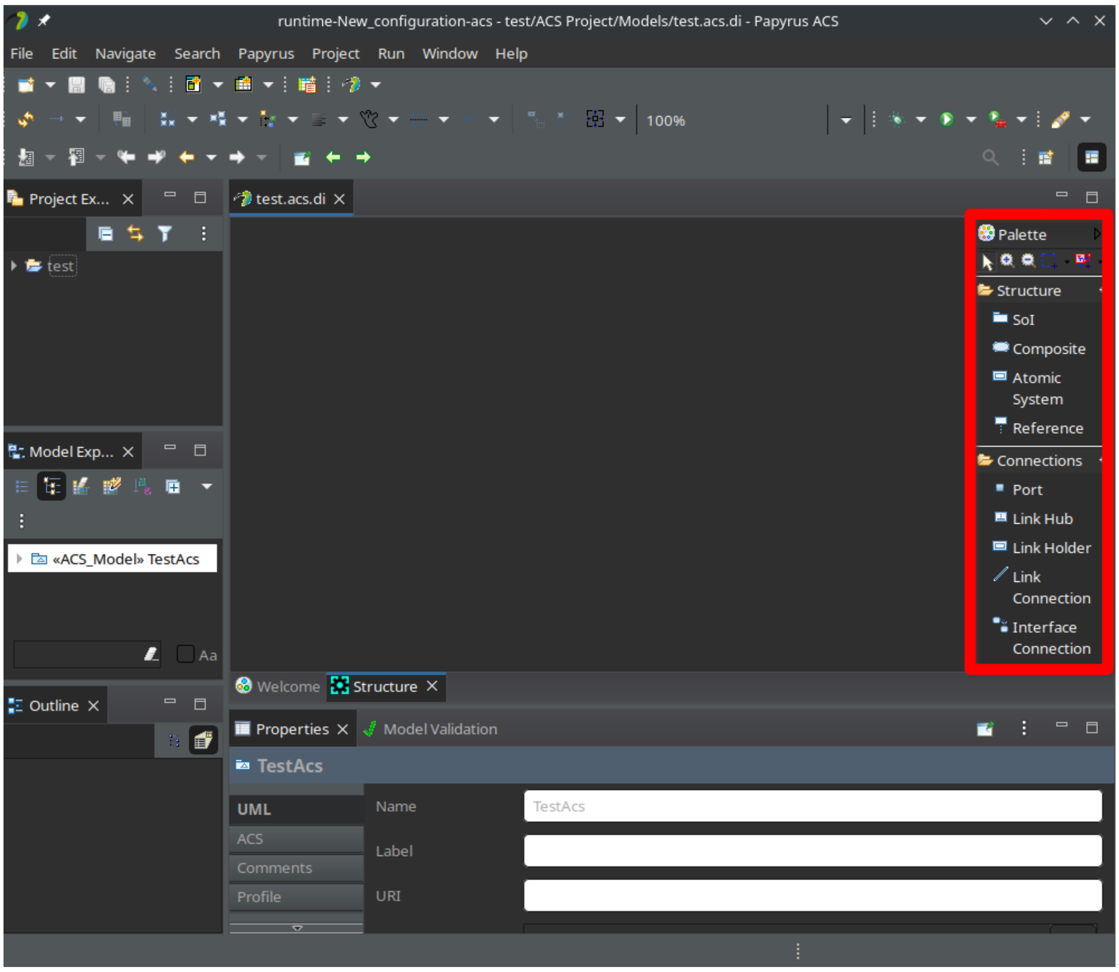


Figure 4.5: The palette made for structure diagrams marked by a red rectangle.

at least possible, e.g. a link doesn't require more participating systems than exists (Line 9).

4.4 Usability Testing

We perform three qualitative usability tests – each lasting one hour – to determine the user-friendliness of the ACS IDE. Two of the participants are experienced Papyrus users and one knows about Papyrus. All of them work with SoSs as a part of their job, either in practical or theoretical settings. The test scope only extends to the parts of the IDE that are currently implemented, which in rough terms is creating an ACS project, building an ACS model, and verifying structural correctness.

4.4.1 Test Setup

We use *Microsoft Teams* to facilitate remote usability sessions, since the participants live and work in a country other than Denmark. The participants can either download the ACS IDE themselves, or use MS Team's remote control feature to use the IDE through the facilitator's computer. However, remote controlling provides a slightly subpar experience, due to input lag.

A participant, facilitator, and observer are present during each session. The facilitator instructs, guides, and inquires with the participant, and the observer takes notes about the participant, facilitator, and session in general. Given the participant's consent, we record the session for later review and delete the recording upon project completion.

4.4.2 Test Plan

The usability test is split into three phases, where we plan the phases to last, respectively, 15, 30, and 10 minutes, leaving 5 minutes as a buffer during the hour-long test:

- In phase one, we inquire the participant about their experience with SoS development, and introduce the participant to the basic knowledge needed to make ACS models. We also provide them with a *reference sheet* to use if necessary.
- In phase two, the participant is exposed to the ACS IDE and is instructed to perform various tasks that leads to the creation of the ACS model described in Section 4.4.2.2. The participant is encouraged to think out loud while working.
- In phase three, we ask the participant about what they liked/disliked, what they would change, and about other, more specific topics we are interested in knowing

```
        if (linkcar.matches("[0-9]+")) {
            if (systemcar.matches("[0-9]+")) {
                if (link.getDirectionality() == ConnectionTypes.SELF_REFERENTIAL)
                    if (Integer.parseInt(systemcar) >= Integer.parseInt(linkcar) +
                         1)
                        return "";
                    else return "Link cardinality is too big (remember self
                        referential link cardinality must be one smaller than
                        system cardinality)";
                if (Integer.parseInt(systemcar) >= Integer.parseInt(linkcar))
                    return "";
                else return "Link cardinality is bigger than system.";
            } else if (systemcar.equals("N")){
                return "";
            } else if (systemcar.equals("*")) {
                return "System is * cardinality so the link must be * also.";
            } else {
                return "System Cardinality must be either [0-9]+ or 'N' or '*'
                    nothing else is allowed";
            }
        } else if (linkcar.equals("N")) {
            if (systemcar.matches("[0-9]+")) {
                return "";
            } else if (systemcar.equals("N")) {
                return "";
            } else if (systemcar.equals("*")) {
                return "System is * cardinality so the link must be * also.";
            } else {
                return "System Cardinality must be either [0-9]+ or 'N' or '*'
                    nothing else is allowed";
            }
        } else if (linkcar.equals("*")) {
            return "";
        }
        return "Link Cardinality must be either [0-9]+ or 'N' or '*' nothing else
            is allowed";
```

Listing 21: Snippet of cardinality verification on Link Connections and Systems.

the effects of. If relevant, we follow up on comments they make during the test.

4.4.2.1 Phase one - Introduction

To gain a better understanding of the participants and how they usually handle SoS development, we first inquire about the following questions:

1. How experienced are you with SoS development as a whole?
2. How do you currently do SoS design? I.e. what tools and processes do you use?
3. On a scale from 0 to 10 wrt. usability, how would you rate your current tools and what do you base this rating on?
4. What properties should a tool or method be able to guarantee about a SoS in your opinion?
5. Would you like a tool to verify that possibly time-critical communication between systems is feasible?
6. How experienced are you with the Eclipse Papyrus modeling tool and when do you use it?

Next, we introduce the participant to the ACS framework using slides that contain quick descriptions and use examples of the components needed for the test. Since the participants will need an introduction to ACS, we only present the IDE syntax and only the theory relevant to build a simple model, while still experiencing the majority of the IDE's features. We make the participant aware of this. The slides double as the reference sheet.

For the Structure layer, we introduce Systems (minus References), Links (minus Boundary Links), and Cardinalities (minus `N` and $Z > 1$). For the Behavior layer, we introduce Handlers, Initiators, Controllers, Communicators (minus sub-machines and functions), and the one-to-one communication mechanism. For the Data layer, we introduce the number and nullable types, and the precision constraint.

4.4.2.2 Phase two - Using the IDE

We introduce a simple library SoS that the participant will model throughout the test, but not in any ACS related syntax or terms. The tasks listed below gradually leads the participant from starting the IDE to the creation of a model of this SoS. Thus, they only have to think about using the IDE, and not how to use ACS. The plan is to fully model the structure and data layers, but only cover the `Borrower`'s behavior to reduce the time spent on the behavior layer.

`LibrarySoS`: Any number of `Borrowers` can exchange their current `Books` for new `Books` at a single `Library`, which has one `FrontDesk` (interface) and one `BookStorage`. `Borrowers` visit the `Library` at most once a day and will wait at most one hour in the queue. The `FrontDesk` will sometimes go to the `BookStorage` to fetch books for the `Borrowers`. The `Book` Data Type is modeled as a single integer ID, since the type system implemented in the IDE is cumbersome to use.

Startup:

1. Download and open the ACS IDE. Alternatively, open it on the facilitator's computer.
2. Create a new ACS project called "`LibrarySoS`".
3. Navigate to the Structure Diagram.

Structure:

1. Add a SoI component called "`LibrarySoS`".
2. Add an AS called "`Borrower`" with a "star" System-Cardinality to the SoI.
3. Add a Composite called "`Library`" with a System Cardinality of "1" to the SoI.
4. Add an AS called "`FrontDesk`" with a System Cardinality of "1" to `Library`.
5. Add an AS called "`BookStorage`" with a System Cardinality of "1" to `Library`.

6. Add an Interface Connection from `FrontDesk` to `Library`.

7. Connect the `Borrower` and `Library` with a Link called "`BorrowLink`".
8. Make the Link Connection to `Library` "reactive" with a Link Cardinality of "1".
9. Make the Link Connection to `Borrower` "active" with a Link Cardinality of "star".

10. Connect the `FrontDesk` and `BookStorage` with a Link called "`StorageLink`".
11. Make the Link Connection to `BookStorage` "reactive" with a Link Cardinality of "1".
12. Make the Link Connection to `FrontDesk` "active" with a Link Cardinality of "1".

13. Validate the model.

Behavior:

1. Add a Controller Diagram to `Borrower` and navigate to it.

2. Apply the `Controller` stereotype to the pre-defined state machine, and the `Controller Body` Stereotype to the region inside the state machine.
3. Add a start state `p1` and a normal state `p2` to the Controller.
4. Add a action transition from `p1` to `p2`. Annotate it with a delay of 1+ days.
5. Add an event action transition from `p2` to `p1`. Annotate it with the Initiator Event `I_ExchangeBooks` and a time constraint that makes it finish in 1 hour or less.

6. Add an Event Diagram to Borrower called `I_ExchangeBooks` and navigate to it.
7. Apply the `Initiator` stereotype to the pre-defined state machine and the `Main Machine` stereotype to the region inside the state machine.
8. Set `BorrowLink` as the Link of the Event through the properties on the Main Machine.
9. Create a start and end state with an invocation transition between them.
10. Add an input variable `current_books` to the start state (ignore that we haven't added input and output types to the initiator yet).
11. On the transition, write that `current_books` is given as input to `Library.GetNewBooks`, and the output is stored in the variable `new_books`.
12. Add the output variable `new_books` to the end state.

13. Validate the model.

Data:

1. Add a Type Diagram called `Global Types` to the SoI component.
2. Add a Type Declaration called `Book`.
3. Define book as a whole number (i.e. a number with no decimals).
4. Go to the `I_ExchangeBooks` Event Diagram and set "list of `Book`" as the input and output type of the initiator.

4.4.2.3 Phase three - Q&A

We first inquire the participant about the following questions. Afterwards, we follow up on any comments the participant made during the usability test. Finally, we ask if there is anything left the participant would like to say before ending the session.

1. Did you *like* anything about the IDE?
2. Did you *dislike* anything about the IDE?
3. Did you find anything particularly easy or difficult?
4. Were there any misleading icons or logos?
5. Did the design appear cohesive and consistent?
6. On a scale from 0 to 10, how pleasing was the IDE to look at as a whole and what do you base your rating on?
7. On a scale from 0 to 10, how pleasing was the IDE to navigate as a whole and what do you base your rating on?
8. On a scale from 0 to 10, how usable did you find the Data Type creation UI and what do you base your rating on?
9. What are the most important things to change in the IDE?
10. On a scale from 0 to 10 wrt. usability, how would you rate the ACS IDE and what do you base this rating on?
11. Did we miss any important questions and what would you answer?

4.4.3 Test Results

This section summarizes the results of each phase of the three test sessions, where Section 5.1 evaluates upon the results. We summarize question answers for phase one and three, and task performance for phase two.

Participants 2 and 3 have been exposed to the previous iterations of ACS 6+ months prior to the usability test, whereas participant 1 saw ACS for the first time in the introduction during phase one. No participant had seen the newest version of ACS before the test.

4.4.3.1 Phase One

Participant 1 is a contributor to the Arrowhead project who has participated in modeling a number of Arrowhead use-cases for educational purposes. They mainly use Papyrus for SoS design, but also use Studio4Education (no ratings were provided). They believe a SoS modeling tool should be able to guarantee consistency, and liked the suggestion in question 4, stating that all kinds of verification is good. They are experienced with Papyrus as a user and as a plugin developer.

Participant 2 works theoretically with SoSs and knows of their function and capabilities, but has no recent practical experience. If they were to use a SoS development tool, they'd like safety and privacy guarantees (the facilitator mentioned safety as an example beforehand). They described the suggestion from question 4 as "essential" from a theoretical perspective and found it odd that they had not yet seen a solution to that problem. They have limited experience with Papyrus; only theoretical knowledge.

Participant 3 is an experienced SoS developer who primarily use Papyrus and various DSMLs that are extensions of UML (including SysML) for SoS development. They rate SysML a "7" and Papyrus a "6" wrt. usability. In their opinion, a tool should provide static verification (e.g., of composition and communication) and verify the correctness, safety, security, and timing properties of a SoS model. They find the suggestion in question 4 very important. They are very experienced with Papyrus and use it almost every day.

4.4.3.2 Phase Two

Participant 1 tested the ACS IDE locally, where the rest took control of the facilitator's computer instead. The experienced Papyrus users (participants 1 and 3) had an easier time with Papyrus features and workflows, where Participant 2 (perhaps with their theoretical knowledge) had an easier time with grasping the ACS features and concepts.

Generally, it was straight forward to create the systems in the structure layer, where Links posed a much bigger challenge (especially the Link Holder). Performance wrt. the behavioral components was generally the same for all participants, where the biggest challenge was to create the annotations on transitions. Finally, modeling Data Types did not seem like a big challenge once the participants became familiar with the correct keywords to look for, however, no one liked the Data Type creation UI.

In case time was running out, we skipped some Phase 2 steps to not go over time. Participant 1 did not make it to "Data, Step 4". Participant 3 did not make it to "Behavior, Step 6-13" and "Data, Step 4". Participant 2 agreed to go 10 minutes over time and thus managed to finish all Phase 2 tasks.

Startup

`Step 1-2:` All participants opened the ACS IDE (Windows) with no problems, although participant 1 initially downloaded the source code due to overlooking the compiled binary on GitHub. All participants quickly managed to enter the ACS Project creation menu and create an ACS project, and most were audibly confused with the long wait between pressing "accept" and the visual feedback from the IDE.

`Step 3:` The experienced users (Participants 1 and 3) immediately navigated to the Structure Diagram, even before the task had been mentioned to them, where participant 2 spent relatively long looking in the wrong menus.

Structure

`Step 1-5:` Modeling the system structure of the library SoS and configuring names and System Cardinalities was quick and easy for all participants. All were bothered by the position of names and did not think they could be dragged into place.

`Step 6:` All participants were quick to make the interface connection after being reminded about Ports.

`Step 7-9:` Participant 2 immediately knew to use a Link Hub for Links between systems and only needed a quick reminder about the Link Holder to move on. For participants 1 and 3, the first time modeling a Link took relatively longer due to the uncommon use of Link Holders and Link Hubs, however, configuring the Link Connections was as quick as with participant 2.

`Step 10-12:` After modeling the first Link, all participants were equally quick and proficient in modeling the second Link.

`Step 13:` Only participant 3 ended up in a situation where they got to use error messages. This helped them remember that Link Connections should go between a Port and a Link Hub, and not two Ports.

Behavior

`Step 1:` Whereas participants 1 and 2 needed a hint that the Controller Diagram should be added to Borrower through the "Model Explorer" and found out the rest from there, participant 3 immediately knew what to do at the notion of a "Diagram".

`Step 2-3:` All participants were quick to apply the stereotypes, as well as to add the start and intermediate states with correct names.

`Step 4-5:` All participants were quick to add the Action Transitions, but took some time to get used to writing the behavior annotations in the transitions' name property. They all needed help remembering the correct syntax for Actions (which was expected).

`Step 6-9:` After the Controller Diagram, participants 1 and 2 were quick to add a new Event Diagram, apply the correct stereotypes, and create the states and transition. For participant 1, we missed Step 8, but participant 2 was quick to figure it out.

`Step 10-12:` Participants 1 and 2 were quick to add the input and output variables to the start and end state. After figuring out that the name property is used to hold Actions in the Controller Diagram, they quickly found out to use it for Invocations in the Event Diagram. With a refresher on the syntax for Invocations, participant 2 quickly managed to write it out. For participant 1, more direct directions for writing the Invocation was needed.

`Step 13:` *Due to a missed patch in the public version of the IDE, the validation plugin crashed when validating behavior diagrams.*

Data

`Step 1-2:` All participants were quick to add a Type Diagram to the SoI based on the experience from adding the previous diagrams and subsequently added the empty “Book” Type Declaration with ease.

`Step 3:` All participants needed either direct (press “that” button) or indirect (you need a “number” type) guidance for each click through the Data Type UI. However, participant 2 was quick to grasp the logic and terminology of the type system which notably increased their proficiency.

`Step 4:` Participant 2 took some time due some confusion about referencing the Book Type, since the “Type Reference” menu first showed up after reopening the Data Type UI. After seeing the “Browse” button under “Type Reference”, they instantly figured out how to make the type reference.

4.4.3.3 Phase Three

Finally, we present the participants’ answers to questions 1-10 from phase 3. Question 11 is omitted since it is a meta-question that affected the final list of questions. Prior participants were asked to answer the updated questions to align all answers.

First, Table 4.1 shows the answers to questions 1 and 2, where the columns represent questions and the rows represent participants (by number). Table 4.2 shows what each participant liked/disliked about the ACS IDE in relation to question 3.

As for questions 4 and 5, participants 1 and 2 found the UI non-misleading, and all participants found it cohesive and consistent. Participant 3 pointed out that the separate “UML” and “ACS” property tabs should be merged[1] into one “ACS” menu, and the colors for ACS components seemed arbitrary and could give wrong ideas about the model (e.g., Atomic Systems are red, but red usually means “error”).

[1]This is possible in Papyrus, but the authors did not know at the time.

#	Liked (Q1)	Disliked (Q2)
1	The simple and accessible component palette. The validation feature.	Positioning of structural component names. The Data Type UI.
2	Loved it a lot in general and found it an essential addition to the tool landscape.	Silly bugs. Lack of shortcuts/point-and-click for recurring actions and easy access to validation features. Nested menus. Hard-to-find menus. Too many menus.
3	How to model structure and state machines.	How to model Data Types and communication.

Table 4.1: Answers to questions 1 and 2. "#" refers to the participant number.

#	Easy	Difficult
1	How the simple palette makes it easy to find and add model components to diagrams (since some frameworks have a lot of components).	Nothing in particular.
2	Found the UI fairly convenient in general.	Nothing in particular.
3	Modeling system structure and state machines. Creating new diagrams.	Modeling Links. Modeling Data Types. Writing Actions/Invocations (*Suggests a specialized editor to improve this*).

Table 4.2: Answers to question 3. "#" refers to the participant number.

Table 4.3 presents each participant's rating of the ACS IDE wrt. questions 6 (Appearance), 7 (Navigation), and 8 (Data Type UI), where Table 4.4 shows what each participant finds the most important to change in the ACS IDE to make it better.

Participant 1 rated "7" for Appearance since it was generally easy to use, but the Data Type UI was bad to look at. They rated "9" for Navigation based on their familiarity with Papyrus, which made navigation easy. They rated "2" for the Data Type UI since it was confusing and unclear.

Participant 2 rated the IDE as a research prototype that is based on an existing platform with a context that cannot be changed, rather than a finished and custom built tool. The rating of "7" for Navigation and Data Type UI are based (mostly) on the Data Type UI, but the rating is slightly difficult to make since they never got into depth with the Data Type UI. If there were no nested menus (especially the Data Type UI), the rating for Navigability and Data Type UI would have been a "10".

Participant 3 rated "6" for Appearance based on their aforementioned dislikes about the UI, where a "10" would have been a web-based UI with improved icons. They rated "8" for Navigation based on their familiarity with Papyrus, where a "10" would require component-based navigation (e.g., double click an AS to enter its Controller). They rated "2" for the Data Type UI based on the nested menus, where a "10" would require having all information on the same page, no need to re-open it, and with a simpler editor.

#	**Rating**									
	1	2	3	4	5	6	7	8	9	10
1				D			A		N	
2							N,D			A
3		D				A		N		

Table 4.3: The answers to questions 6, 7, and 8. "#" refers to the participant number. The letters represent **A**ppearance (Q6), **N**avigation (Q7), and **D**ata Type UI (Q8).

#	**Things to change**
1	A simpler Data Type UI. The "Declared Type" stereotype tag is not visible in the Type Diagram, which is confusing.
2	The things mentioned in in question 2. Shortcuts, point-and-click, and fewer, simpler menus (all equally important).
3	A simpler Data Type UI. Make diagrams more attractive with better icons and colors.

Table 4.4: Answers to question 9. "#" refers to the participant number.

Lastly, Table 4.5 shows the three Participants' overall usability rating of the ACS IDE in relation to question 10.

Participant 1 rated "7" based on the overall modeling and verification process being easy, apart from the Data Type UI. A "10" would require all menus being clear and a good reference sheet. Participant 2 rated "10" for the overall experience of the IDE as a research prototype. Participant 3 rated "7" based on the ease of modeling the Structure Diagram and state machine based diagrams, and that validation is interesting and useful.

#	Rating									
	1	2	3	4	5	6	7	8	9	10
1							✓			
2										✓
3							✓			

Table 4.5: The answers to question 10. "#" refers to the participant number.

Additional comments

Participant 2 mentions that only few would be familiar with the "SoI" term – so "SoS" would be a less confusing name for the SoI model component – but appreciates the effort to use a standardized term and separate concepts. Also, although they had little practical experience with Papyrus, their theoretical knowledge helped them navigate the UI.

5 Results

We evaluate on the usability results and the project process which reveals how the current version of the ACS IDE was executed and how the project was planned and handled. The usability tests can help evaluate ACS itself to some extent since certain IDE features that the test participants liked are direct results of certain ACS features.

We conclude based on the final state of the artifacts produced during the project and the evaluations described above, followed by various future works for the ACS framework and IDE.

5.1 Usability Evaluation

We evaluate on the "test premise" – since there are unusual circumstances surrounding the usability test – on "observed patterns" in the test results – since these indicate what is good/bad about the IDE and how different people react to it – and on "overall usability".

5.1.1 The test premise

Normally, usability tests are made with field experts who evaluate how a tool affects their usual workflow(s), however, the most fundamental challenge for us was that there exist no other "ACS experts" apart from the authors themselves. It is thus a challenge for any external party to decide whether the ACS IDE improves any workflow(s) or not. However, based on the answers from Phase One (see Section 4.4.3.1), we know that the participants liked the basic idea of verification of communication outlined in question 4.

In hindsight, 1 hour tests were too short. We only introduced a limited version of ACS and only managed to implement half of the intended features for the IDE, and even then, the participants barely had time to fully experience ACS and the IDE. Thus, if a participant was getting stuck with a task, instead of letting them figure out the solution, we had to help them rather quickly to not go over time. Future tests should allocate more time and include experienced ACS users or be preceded by an ACS course.

Furthermore, to our knowledge, there are no frameworks similar to ACS (wrt. its modeling and verification approach), which only makes it harder to evaluate ACS and the IDE, since there is not much to compare these with. While we do realize a test on 3 people might be too little to produce any hard facts, the tests still yielded a good amount of results about the IDE's user experience.

5.1.2 Observed patterns

Proficiency

Whereas the practical SoS practitioners (participants 1 and 3) handled the Papyrus and UML-specific features well, the theoretical SoS practitioner (participant 2) was quick to get used to the ACS-specific extensions and conventions. Conversely, the practical party took relatively longer to get used to ACS, whereas the theoretical party needed more guidance with Papyrus. This was a general tendency throughout all test sessions.

The practical party being the most proficient with Papyrus was expected, but we were surprised to see how much certain ACS additions (e.g., Links) seemed to hinder them for some time until it "clicked" for them. Conversely, the theoretical party quickly accepted Links (and Actions/Invocations in the name property) for what they were. Since ACS is implemented as an extension/Profile of UML, we suspect that the practical party's initial confusion was due to a betrayal of expectations to the UML norm and their experiences with Papyrus.

UML Ports can only be placed on UML Component boundary boxes, whereas ACS Link Hubs can both be placed on System Boundaries and freely inside a system. Since ACS Link Hubs are extensions of UML Ports, practical practitioners might not initially consider "free Links" a possibility. Participant 3 asked "why not just use the Link Holder as the Link Hub?", but this is impossible since UML Connectors can only connect UML Ports, whereas the Link Holder must be a UML Component to be placed freely.

We expected difficulty explaining this to people *without* UML experience, not the other way around. This is a peculiar result since the UML Profile for ACS was made in part to ease the introduction of ACS to UML users. However, this is just a small matter of understanding why models are as they are; not a matter of using the tool. The majority of ACS additions and concepts were quickly understood; especially systems and behavior diagrams which resemble regular components and state machines.

Unclear component selection

A recurring pattern was that participants sometimes selected and edited the wrong component for a while before eventually realizing what was happening. For some reason that we had no time to pursue, it is possible to sometimes misunderstand which component is being edited.

5.1.3 Overall usability

As with the pattern of proficiency, there is a subtle but clear separation between the practical and theoretical parties' experience of the ACS IDE. The theoretical party generally rated the ACS IDE more favorably than the practical party, where the two practical practitioners gave almost identical ratings (see Tables 4.3 and 4.5). This could be because the practical party focuses more on the tool, whereas the theoretical party focuses more on possible applications of the ACS framework (and was more lenient with the ratings).

The practical party especially liked the small number and easy retrieval/composition of diagrammatic components (goes for all layers), based on a tendency in other frameworks to have an abundance of components that can be hard to manage. This property of the IDE is directly attributable to the design of the ACS framework. Participant 2 (the theoretical party) liked the IDE a lot in general, which does not say much.

Participant 2 pointed out a lack of shortcuts and point-and-click solutions for simple and recurring actions, as well as more direct access to the validation feature, which is important for both the framework and IDE. Participant 3 pointed out a lack of shortcuts as well, but in terms of navigation using the model components themselves. We agree that such features would greatly improve the IDE.

All test participants disliked the Data Type UI, which is as expected, but to varying degrees (see Table 4.3). Only participant 3 mentioned the Data Type UI as particularly difficult to use (see Table 4.2), however, all participants still stated that it is important to change the Data Type UI (or nested menus) to improve the IDE (see table 4.4).

Even though the ACS IDE still needs many small and a few big improvements (apart from implementing the missing features), it still got an average overall rating of "8" (individual ratings: 7, 10, 7). This is a good result given that Participant 3 rated SysML "7" and Papyrus "6" (we did not acquire such ratings from the other participants).

5.2 Project Process Evaluation

This project was made by three students and consequently there was a need for coordination and knowledge sharing. This was structured around a short daily meeting every morning where project status and other information were communicated. A weekly meeting with the supervisor was used as a retrospective opportunity where we talked about what we achieved the previous week and how to better achieve our goals for the following week.

5.2.1 Work distribution

We distributed tasks to individuals with the goal of achieving parallel work. This had the risk of introducing divergent understandings of ACS between the tasks (e.g. implementation wouldn't match theory), however, due to the sheer number of meetings this risk was mitigated. We dedicated more time to a task when its deadline approached. This ensured that all three students understood and approved of the direction even if they hadn't previously been active in that task. Table C.1 (appendix C) visualizes how much time was dedicated to each task in a given week. We notice that in week 20 (deadline for article) and week 23 (deadline for report), all three students were dedicated to that one task, otherwise the distribution of work was fairly even between the different tasks. Overall the distribution worked well and the meetings full filled their intended purpose.

5.2.2 Tools and methods

We used standard communication tools to share artifacts and messages outside office hours, however, given the daily meetings there was little use for these, and their choice was not that relevant. We used eclipse and papyrus to implement ACS. We used more time than we wanted to get a working development environment up and running. This is partly because the tools and methods for development were new to all group members. The tools had a significant learning curve and the documentation was either outdated or missing. We greatly benefited from contacting some of the developers behind the tools as they provided invaluable feedback on code snippets. We all agreed that eclipse and papyrus certainly had a lot of room for improvement in this regard, but we also weren't able to find an alternative that could replace them for this use case. The only other solution we could imagine was a fully custom tool, however, that would introduce a different set of uncertainties and risks.

5.3 Conclusion

In this work, we continued the development of the "Abstract Communicating Systems" modeling framework (ACS), where the main contribution consists of a new iteration of ACS with many corrections and new features, as well as a half-finished IDE for ACS. Most of the groundwork had already been laid during two prior semester projects, so no additional preliminary work was needed. The first project [12] provided the foundation for ACS and the second [11] provided the insights that put ACS into context by surveying existing tools and methods.

In a nutshell, if the project started with a hundred goals, it ended with a thousand remaining. The deeper we got into the development of ACS and the IDE, the more we realized how much we could achieve, as well as how much had to be postponed (see Section 5.4) due to the scale of the project as a whole. While we made much progress on ACS and the IDE, more research and work are needed to get both industry-ready.

Our initial efforts on ACS started with the future works from the previous ACS version [12] and the research problems and directions identified by the survey from last semester [11]. The former lead to the unified “Cardinality” and the “reflexive” Link Connection type, where the latter lead to an overhaul of ACS’ communication mechanisms (see Section 2.3.3.4) and entire type system (see Section 2.4.5). From here, the new communication mechanisms spawned the Boundary Link and the Publish/Subscribe Event Types, as well as Tokens used to remember connections and filter Subscriber inputs.

Other ACS improvements such as Locking, clearer syntax and semantics in general, and field selection on variables in Communicators gradually appeared in the examples we used for discussion.

We based the ACS IDE on Eclipse Papyrus due to it being the current best on the market that was open-source and since we had a connection to one of the developers (who we had to lean on a few times). The learning curve was high and more than a month was spent going from scratch to getting a simple plugin to compile. Due to a lack of (or outdated) documentation, much inspiration was sought from other public Papyrus plugins with a similar focus. With time, the ACS IDE started working and the current version supports all ACS syntax and most static verification. Communication verification is left as future work.

Finally, we sought the help of three SoS-related industry experts to perform usability tests on the ACS IDE. Usually, such tests are performed with field experts, however, since ACS is brand new and one-of-a-kind, there are no ACS experts (apart from the authors) and only a few frameworks to compare it to. Thus, the tests generally consisted of two questionnaires, a limited introduction to ACS, and a list of very direct usability tasks.

Overall, the ACS IDE received positive responses – some of which are directly attributable to the design of ACS itself – but a few big problems stood out (see Sections 4.4.3 and 5.1 about usability results).

As an additional effort to disseminate ACS into the academia, we have published the ACS IDE source code [5] to GitHub and submitted a 10-page article about the ACS framework, the UML Profile, and the IDE to the Technical Track of the “MODELS 2022[1]” conference [39]. As of writing, the notification to authors about acceptance is expected on the 12th of July.

[1]ACM / IEEE 25th International Conference on Model Driven Engineering Languages and Systems

5.4 Future Works

A lot of features and improvement ideas for the ACS framework had to be postponed due to the scale of the project as a whole. The following sections describe what lies in store for Structure, Behavior, Data, High-level verification, and other extended features. The last section describes which features did not make it into the ACS IDE, both planned and from the usability tests.

5.4.1 Structure

Global References and Circular Dependencies

Global References have been removed for reasons described in Section 2.2.5, but these could interplay well with "Public ACS Models" (see Section 5.4.5), so we plan to re-introduce them in the future when we have a better idea of their semantics/implications.

Furthermore, it is possible for two separate SoSs to become dependent on each other over time, which may require both global references and circular dependencies between systems/models (none of which are currently supported). Interdependent SoSs could even merge, in which case support for merging (or similar) ACS models would be an interesting feature. Since communication and data would remain the same, we believe only the Structure layer would require non-trivial merging.

Improved N-Cardinality

The **N**-Cardinality describes a number of systems that will be known at runtime. However, it might be possible at design time to know the range in which this number will lie. Thus, we plan to use inequalities (or similar) to describe this range (which may not exceed any Z-System-Cardinalities). Some ideas are:

$$\begin{array}{cccc} 1 <= N & 1 <= N <= 5 & N <= 5 & N \\ [1; inf] & [1;5] & [0;5] & [0; inf] \end{array}$$

Link Aliasing and Port Aliasing

Link Connections cannot cross System Boundaries, and while a Boundary Link supports communicating across a single Boundary, it is impossible to cross two or more Boundaries.

However, this can be circumvented by creating a “dummy interface AS” with exactly the same behavior as an outside system, and this is repeated for every additional boundary that needs crossing. These dummy ASs then proxy the communication out until communication reaches the desired system. This could resemble a Proxy or VPN.

Since circumventing this is possible using one of the most essential features in ACS, and since this kind of communication might be plausible anyway, it could be a good idea to add language support for it. Similar to an Interface Connection (see Section 2.2.2.2), a “Link Alias” (or similar) could set a parent Boundary Link as an alias of a child Boundary Link. We’re unsure about the specifics of Link Aliasing; especially wrt. free Links.

Whereas Link Aliasing would allow “going up” through the nesting layers, Port Aliasing could allow “going down” through the nesting layers; especially by allowing non-AS systems to be interfaces. Thus, the two mechanisms would mirror each other.

Also, some non-interface systems might want to communicate out, without any outside systems being able to communicate in. This could be allowed using (modified) Boundary Links or by annotating Ports with, e.g., “OUT”.

Better Control of Link Promises

The Link promise refers to how all Events that communicate over a Link must have a communication path that makes use of all system instances that the Event could possibly communicate with. However, imagine fetching data and depending on some choice, the data either goes to one or another system. In that case, one Link must have two (reactive) Link Connections, where it only ever makes sense to use one of them.

We could solve this by using “weak” Link Connections to denote that it is not required to use all/any instances of the connected system.

System Polymorphism

Multiple systems might share a common interface in terms of Handlers and/or behavior, and possibly more. It would seem realistic that instances from two different systems could be used for the same communication in certain cases, i.e., an invocation in a Communicator could either pick `SharedHandler` on `System1` or `System2`.

This resembles the notion of C# or Java interfaces. Perhaps similar interfaces could be applied to systems/ASs? However, this could require a single Link Connection to connect to two different systems, or perhaps a shared “Interface component”, which is then connected to two (or more) systems. We are unsure of how to best achieve this.

5.4.2 Behavior

Multi-Threading and Improved Tokens

Section 2.3.3.3 explains "control flow" and how each AS only has one. Control flow resembles threads from programming, and a single one might not be enough to properly model the behavior of an AS. Specialized transitions in the Controller and/or Communicator could be used to spawn and destroy control flows (threads).

Multiple control flows would affect Tokens, Locking, and possibly how communication works. Tokens should remember which flow(s) they point to, Locking should affect individual flows, and communication with multiple flows at once might be possible. A system instance could thus also communicate with itself.

Furthermore, sending Tokens between systems could be interesting, which would require Tokens on Invocation Transitions (like Variables) to use them as input/output. Links could affect allowed Token exchanges, but sending Tokens to systems with no compatible Link would also be interesting since this could resemble how a VPN works.

Instance and Circumstance Specific Timing Constraints

Time constraints might have to vary depending on specific circumstances on each instance of some system. Thus, time variables could be used to configure and/or compute delays at run-time/verification-time. While more granular timing constraints could provide valuable information and analyses, it might not be necessary since the non-deterministic choices during verification will select the delays that result in correct behavior.

Improved Locking Semantics

It might benefit ACS to support group-based Locking of system instances since such behavior is within the range of possibility, but we currently do not have any use cases for this nor a good idea of the semantics and wider implications thereof.

Additionally, since the `"lockable"` attribute is currently placed on states, it is not possible to differentiate between lockable and non-lockable paths/transitions leaving that state. This can be circumvented by having a zero-delay Action leading to the lockable state, but this feels like a sub-optimal solution. The `"lockable"` annotation could instead be placed on Action transitions where communication is used to lock/unlock, but then locking/unlocking cannot happen without communication (which it can currently).

Finally, lockable elements (states or transitions) and Tokens could be annotated with a

Time To Live (TTL) that states “how many transitions they’ll last” before self-destructing.

Improved Data Flow

Since this version of ACS adds support for range-constraints on the size of arrays, it is now also possible to add array indexing to Variables in Communicators (which currently only supports field selection for Record Types). If the supplied index is guaranteed to be valid, we’re guaranteed to get an output value. If not guaranteed, the element type could become nullable or it could be considered an error.

Another possibility to improve data flow is to introduce a “bag” to ASs, which is a predefined set of AS-scope-global Variables that can be read/overwritten at any time, possibly using transitions in Communicators. Thus, data could implicitly be carried between/across any number of Communicator invocations. The number of bag-variables never changes, although the bag-Variables can have values of any size.

Variable and Token Operations

A Variable’s type is composed of a limited set of fundamental/elemental types (and constraints) that have well-defined properties and capabilities. Thus, it could be possible to model arithmetic operations between numbers, logical operations between booleans, equality/inequality comparisons, array concatenation, array append, array size, nullable checks, and more.

Even though ACS never deals with concrete values, this makes data flow more explicit, allows for better code generation (see Section 5.4.5), and could be used to enhance the capabilities for more explicit decision-making based on data values (see next point).

Furthermore, Group Tokens are essentially sets, so set operations between Tokens could perhaps be a useful addition, however, we do not have any concrete use cases for this.

More Explicit Decision Making

States and/or transitions in Controllers and Communicators could be annotated with invariants or guards based on the hypothetical values of Variables (even from the “bag” suggested above). This could improve code generation (see Section 5.4.5) and verification capabilities by making more accurate non-deterministic choices that have less risk of accidentally making a faulty model appear correct. Also, priorities on transitions could be used to make consistent decisions when multiple choices are available.

Furthermore, since Communicators can have multiple end-states, it would make sense for the specific end-state of an Invocation to affect how the Controller will behave afterward. Currently, we have no ideas as to what the syntax for this feature should look like.

Lastly, “Error handlers” have been removed for reasons explained in Section 2.3.6, but these provide more realistic behavior modeling and should be reintroduced.

5.4.3 Data

Streamlining the Type System

Due to pre-defined types (see Section 2.4.1), global types had to be compared on structure instead of names since the user would otherwise be unable to properly extend predefined types. For example, extending `int32` to make an `int20` would not allow `int16` to be used as an `int20`, since they would not be on the same branch of the specialization tree.

We are thinking of swapping pre-defined types for type libraries, which can be edited for specific needs. This would restore “name comparison” to global types. Alternatively, instead of “pre-defined” types, we could switch to “auto-generated” types. Need an `int42` and `int64`? Need a `string_UTF8` and `string_ASCII`? These are simply generated by ACS with the correct structure and inheritance trees based on a naming scheme.

Furthermore, if a global type propagates out of a SoI/Reference, we are still unsure whether it should become a scoped or global type. Also, we’re unsure whether a SoI should or should not automatically import the global types of its dependencies.

More Flexible Types

Arrays currently add a single dimension, whereas multi-dimensional arrays are closer to “lists of lists” and thus have no guarantee on structure, such as being rectangular (2D), cubic (3D), or whatever it is called for higher dimensions. I.e., the array type `TYP[][]` could be an array with two sub-arrays, where the first sub-array has 3 elements and the second has 4 elements.

Instead, the array type `TYP[,]` would be forced to have the shape of an $X \times Y$ matrix. Likewise, `TYP[„]` would be an $X \times Y \times Z$ cube, and so on. Furthermore, by annotating types in the square brackets, the dimensions could be given types, thus turning the array into an associative array. Some examples (with very explicit type names) are `Person[string]` and `Product[Size, Type]`.

5.4.4 High-level Verification and Analysis

Loop Detection

It is possible for a single system or loop of systems to make a circular communication chain that could exhaust all available instances (resulting in an error) or would require infinitely many instances due to a lack of behavioral options. In the future, we plan to add semantics to detect and thwart such loops.

Verification of Dynamic Topologies

ACS has trouble verifying SoSs where the number of system instances changes at run-time since the current verification approach assumes a constant number of non-terminating system instances. In the future, ACS should also be able to account for the appearance and disappearance of system instances. This is particularly relevant with respect to the '**N**' and '*' Cardinalities.

SoI Validation Assumptions

If a SoI has an interface, external systems need to communicate with that interface to make the SoI exhibit its modeled behavior. However, ACS does not model anything outside the SoI, but the timings of outside communication might be as important for verification as on the inside. Since we cannot assume anything about outside communication patterns, we currently do not know how to verify a SoI with an interface.

We could make a new SoI/Model that models some sample communication with a reference to the SoI we want to test, but this might not be representative of practical issues. We need a proper way to perform this kind of verification.

5.4.5 Extended Features

Public ACS Models and Obfuscation

To allow SoS designers to discover other models and to trace how various ACS models depend on each other, it would be interesting to make a public ACS library. It could further list global instances of these models which could then be referenced by Global References (see Section 5.4.1), and thus also show how global/public instances inter-depend.

Furthermore, since some designers would like to hide model details from other people, but still allow others to interface with said model, there could be a need for "model obfuscation". This would swap the internals of a SoI for a minimal model with the same external behavior and that does not hint to the original structure of the SoI.

Model and Code Generation

The separation of Controllers and Communicators and the interplay between these across systems may not be the easiest to handle if long communication chains across various systems need to be modeled. For an overview, it could be interesting to allow a mapping to and from the Controller/Communicator model and a central "sequence diagram"-based (or similar) representation of one (or more) communication chains. Thus, the designer would simply have to modify the variant they are most comfortable with.

Additionally, ACS intends to support code framework generation for software-based ASs. These frameworks implement all the communication logic needed by each system implementation, where internal behavior is programmed and coupled with the framework separately. Old frameworks should be easily swapped out with newly generated ones, as the SoS gradually evolves. Other artifacts such as documentation can also be generated.

Deployment and Update Planning

After developing a SoS, it must also be deployed. The deployment could happen in different stages (containing different systems), depending on how each system depends on other systems. ACS could produce a plan based on the model that describes how a SoS can (or cannot) be deployed. The same goes for updates to the SoS, where old components must be swapped out for new ones at different times to not crash the SoS. Such planning could take concepts such as down-time, distribution, and throughput into consideration, and maybe even generate a script that could automatically handle the issue.

5.4.6 IDE

Future work on the ACS IDE must first focus on getting the remaining verification features implemented (some static and all high-level verification). Afterward, the focus should be on the quality of life features mentioned by the usability test participants, such as improving color schemes and icons and making shortcuts to and point-and-click editing for basic/recurring actions (see Sections 4.4.3.3 and 5.1). Especially the Data Type UI. Later works can focus on the features mentioned in Section 5.4.5.

Alternatively, create an ACS IDE from scratch, which can be tailored to optimize the ACS workflow. Thus, we would not have to deal with the many features in Papyrus we do not need, thus also reducing confusion by too many choices. However, making software as good as Papyrus from scratch would be a huge undertaking.

Bibliography

[1] Forbes; IHS.
Internet of Things (IoT) connected devices installed base worldwide from 2015 to 2025.
url: https://www.statista.com/statistics/471264/iot-number-of-connected-devices-worldwide/.
Last retrieved 3-06-2022.

[2] SysML.org.
SysML Open Source Project - What is SysML? Who created SysML?
url: https://sysml.org/.
Last retrieved 3/06/2022.

[3] robmosys.
Robmosys - Composeable Models and Software.
url: https://robmosys.eu/.
Last retrieved 3/06/2022.

[4] Eclipse Contributors.
Papyrus/Papyrus Developer Guide.
url: https://wiki.eclipse.org/Papyrus/Papyrus_Developer_Guide#Papyrus_Plug-ins_and_Features.
Last retrieved 27-05-2022.

[5] ACS developers.
The ACS Modeling Tool.
2022.
url: https://github.com/acs-modeling-tool/acs-modeling-tool.

[6] Object Management Group.
Unified Modeling Language.
Version 2.5.1.
Dec. 2017.
url: https://www.omg.org/spec/UML/.

[7] UPPAAL developers.
Features.
url: https://uppaal.org/features/.
Last retrieved 06/06/2022.

[8] Wikipedia contributors.
List of model checking tools.

url: `https://en.wikipedia.org/wiki/List_of_model_checking_tools`.
Last retrieved 06/06/2022.

[9] The Community Research and Development Information Service.
Designing for Adaptability and evolutioN in System of systems Engineering.
url: `https://cordis.europa.eu/project/id/287716`.
Last retrieved 08-06-2022.

[10] The Community Research and Development Information Service.
Comprehensive Modelling for Advanced Systems of Systems.
url: `https://cordis.europa.eu/project/id/287829`.
Last retrieved 08-06-2022.

[11] Emil Palmelund, Jonas Madsen, and Sean Kristian Remond Harbo.
Methodologies and Tools for System of Systems Engineering.
Our 9th semester project report on AAU.

[12] Emil Palmelund Voldby et al.
Communication-Oriented Modelling of Systems-Of-Systems.
The 7th semester project report on AAU of two of the current project's authors.

[13] Sean Kristian Remond Harbo et al.
"Communication Oriented Modeling of Evolving Systems of Systems".
In: *2021 16th International Conference of System of Systems Engineering (SoSE)*.
2021,
Pp. 88–94.
doi: `10.1109/SOSE52739.2021.9497495`.

[14] OMG Consortium.
What is UML.
July 2005.
url: `https://www.uml.org/what-is-uml.htm`.
Last retrieved 08-06-2022.

[15] Kirill Fakhroutdinov.
UML, Meta Meta Models and Profiles.
url: `https://www.uml-diagrams.org/uml-meta-models.html`.
Last retrieved 08-06-2022.

[16] Kirill Fakhroutdinov.
UML Profile.
url: `https://www.uml-diagrams.org/profile.html`.
Last retrieved 08-06-2022.

[17] The Eclipse Foundation.
About the Eclipse Foundation.
url: `https://www.eclipse.org/org/`.
Last retrieved 08-06-2022.

[18] The Eclipse Foundation.
Notes on the Eclipse Plug-in Architecture.

url: https://www.eclipse.org/articles/Article-Plug-in-architecture/plugin_architecture.html.
Last retrieved 08-06-2022.

[19] The Eclipse Foundation.
Eclipse Modeling Tools.
url: https://www.eclipse.org/downloads/packages/release/2022-03/r/eclipse-modeling-tools.
Last retrieved 10-05-2022.

[20] The Eclipse Foundation.
Eclipse Papyrus™ Modeling environment.
url: https://www.eclipse.org/papyrus/.
Last retrieved 10-05-2022.

[21] DANSE Consortium.
DANSE - Designing for Adaptability and evolutioN in System of systems Engineering.
url: https://web.archive.org/web/20210424073231/http://www.danse-ip.eu/home/index.html.
Last retrieved 08-06-2022.

[22] COMPASS Consortium.
Comprehensive Modelling for Advanced Systems of Systems.
url: https://web.archive.org/web/20210424034357/http://www.compass-research.eu/.
Last retrieved 08-06-2022.

[23] IEEE Consortium.
Systems and software engineering — System of systems (SoS) considerations in life cycle stages of a system.
url: https://www.iso.org/standard/71955.html.
Last retrieved 06-06-2022.

[24] Wikipedia Contributors.
Routing.
url: https://en.wikipedia.org/wiki/Routing#Delivery_schemes.
Last retrieved 06-06-2022.

[25] Gorry Fairhurst.
Unicast, Broadcast, and Multicast.
url: https://erg.abdn.ac.uk/users/gorry/course/intro-pages/uni-b-mcast.html.
Last retrieved 06-06-2022.

[26] C. Metz.
“IP anycast point-to-(any) point communication”.
In: *IEEE Internet Computing* 6.2 (2002), pp. 94–98.
doi: 10.1109/4236.991450.

[27] Gorry Fairhurst.

Transmission Control Protocol (TCP).
url: https://erg.abdn.ac.uk/users/gorry/course/inet-pages/tcp.html.
Last retrieved 06-06-2022.

[28] Gorry Fairhurst.
The User Datagram Protocol (UDP).
url: https://erg.abdn.ac.uk/users/gorry/course/inet-pages/udp.html.
Last retrieved 06-06-2022.

[29] Catchpoint.
MQTT Broker.
url: https://www.catchpoint.com/network-admin-guide/mqtt-broker.
Last retrieved 06-06-2022.

[30] Mathias Knøsgaard Kristensen; Felix Cho Petersen; Emil Palmelund Voldby; Simon Vinberg Andersen; Sean Kristian Remond Harbo.
Communication-Oriented Modelling of Systems-Of-Systems: A Modelling Framework Concerning Verification and Code Generation for Systems-of-Systems.
Last Retrieved 24-05-2022. Requires login.
Dec. 2020.
url: https://projekter.aau.dk/projekter/da/studentthesis/communicationoriented-modelling-of-systemsofsystems(7d1e31c1-09ee-47a6-b116-7ca794b1dcd6).html.

[31] Robert W. Sebesta.
Concepts of programming languages.
11th edition.
Pearson Education, 2016.
isbn: 978-0-13-394302-3,

[32] Timothy C. Lethbridge et al.
“Umple: Model-driven development for open source and education”.
In: *Science of Computer Programming* 208 (2021), p. 102665.
issn: 0167-6423.
doi: https://doi.org/10.1016/j.scico.2021.102665.
url: https://www.sciencedirect.com/science/article/pii/S0167642321000587.

[33] Various Authors.
ModelioOpenSource-Modelio.
url: https://github.com/ModelioOpenSource/Modelio.
Last retrieved 30-05-2021.

[34] bmaggi.
bmaggi-Papyrus-ListPublic.
url: https://github.com/bmaggi/Papyrus-List.
Last retrieved 30-05-2021.

[35] Various Authors.
index-org.eclipse.papyrus.git.

url: https://git.eclipse.org/c/papyrus/org.eclipse.papyrus.git/.
Last retrieved 31-05-2021.

[36] Eclipse.
M2Eclipse.
url: https://www.eclipse.org/m2e/.
Last retrieved 27-05-2022.

[37] The Eclipse Foundation.
Eclipse IDE for Eclipse Committers.
url: https://www.eclipse.org/downloads/packages/release/2022-03/r/eclipse-ide-eclipse-committers.
Last retrieved 10-05-2022.

[38] Eclipse.
Oomph.
url: https://eclipsesource.com/blogs/tutorials/oomph-basic-tutorial/.
Last retrieved 1-06-2022.

[39] University of Montreal.
Technical Track.
url: https://conf.researchr.org/track/models-2022/models-2022-technical-track#Foundations-Track.
Last retrieved 08-06-2022.

A Journal

In total there were 19 weeks in the semester (Week 5 - week 23). These are notes that describe main achievements from each week. Some activities that appeared often were:

- A Supervisor meeting.
- General ACS discussion.
- General Code/Report/Article improvements.
- Daily group meeting to coordinate project tasks and status.
- Project administration and coordination for various related tasks.

These are not included in the journal below as they would appear in almost every week.

Week 5, 2022: 31th jan – 6th feb

Planned tasks and agreed on a scope for the semester. Initiated contact with supervisor and discussed the best way to precede with the semester.

Week 6, 2022: 7th feb – 13th feb

Examined the requirements for the semester in terms of exam. Read some old exam reports to get a feeling for the semester. Read some documentation on Papyrus and Eclipse plugins. Discussed with supervisor report structure and relevant conferences for a related article.

Week 7, 2022: 14th feb – 20th feb

Finished the type system's abstract syntax, fundamental types (incl./excl. enums), and type-substitution/sub-typing rules. Developed a "hello world" eclipse plugin. Made artifacts for a related article (Process description document).

Week 8, 2022: 21th feb – 27th feb

Mostly finished type system. Discussed ways to improve ACS deficiencies in terms of dataflow. Continued work on minial eclipse plugin. Drafted master thesis contract.

Week 9, 2022: 28th feb – 6th mar

Explore idea of Protocol Diagram. Started formalization of ACS (type propagation first). Rename some ACS components (e.g. directionalities). Implemented a wizard to create ACS projects and styled with a SysML profile.

Week 10, 2022: 7th mar – 13th mar

Finished and submitted Master thesis contract. Add publish subscribe communication to ACS. Planning report structure. Made graphical assets for the plugin and made palette for plugin. Explored differences between SysML profile and UML profile(2.5).

Week 11, 2022: 14th mar – 20th mar

Made a fully reference-based mathematical definition of ACS. Finished the abstract syntax.

Week 12, 2022: 21th mar – 27th mar

Implemented some ACS stereotypes in Papyrus. Worked on semantics formalisation.

Week 13, 2022: 28th mar – 3th apr

Formalizing the semantics. Dealt with some profile implementation limitations. Worked on the profile and properties menu.

Week 14, 2022: 4th apr – 10th apr

Formalizing verification semantics. Completed the uml profile. Finished implementing properties menu and started verification.

Week 15, 2022: 11th apr – 17th apr

Started work on accompanying article about ACS.

Week 16, 2022: 18th apr – 24th apr

Continued work on formalizing the verification semantics. Propose new combination syntax for formalisation, including structural-operational semantics and custom list semantics. Finished complete report structure. Finished first few chapters of report.

Week 17, 2022: 25th apr – 1st may

Written report chapters 2. Discussed Type Diagrams and papyrus implementation relations.

Week 18, 2022: 2th may – 8th may

Finished basic validation in ACS tool. Written more report chapters. Code refactor and clean for public release.

Week 19, 2022: 9th may – 15th may

Written article chapters. Made public code release for double anonymous conference (attachment to article). Made example ACS models reference in the paper and elsewhere.

Week 20, 2022: 16th may – 22th may

Finished article for submission. Polished code base and compiled into a package with release.

Week 21, 2022: 23th may – 29th may

Conducted first Usability test. Written report chapter 3. Outlined and planned chapter 4.

Week 22, 2022: 30th may – 5th jun

Improved multiple report chapters. Written chapter 4. Improved usability test based on the gained experienced from previous test. Finished one more usability test.

Week 23, 2022: 6th jun – 12th jun

Finished one more usability test. Improved many chapters in report. Many finishing touches including: report formatting, summary, title, journal, appendix and bibliography.

B Full dependency graph

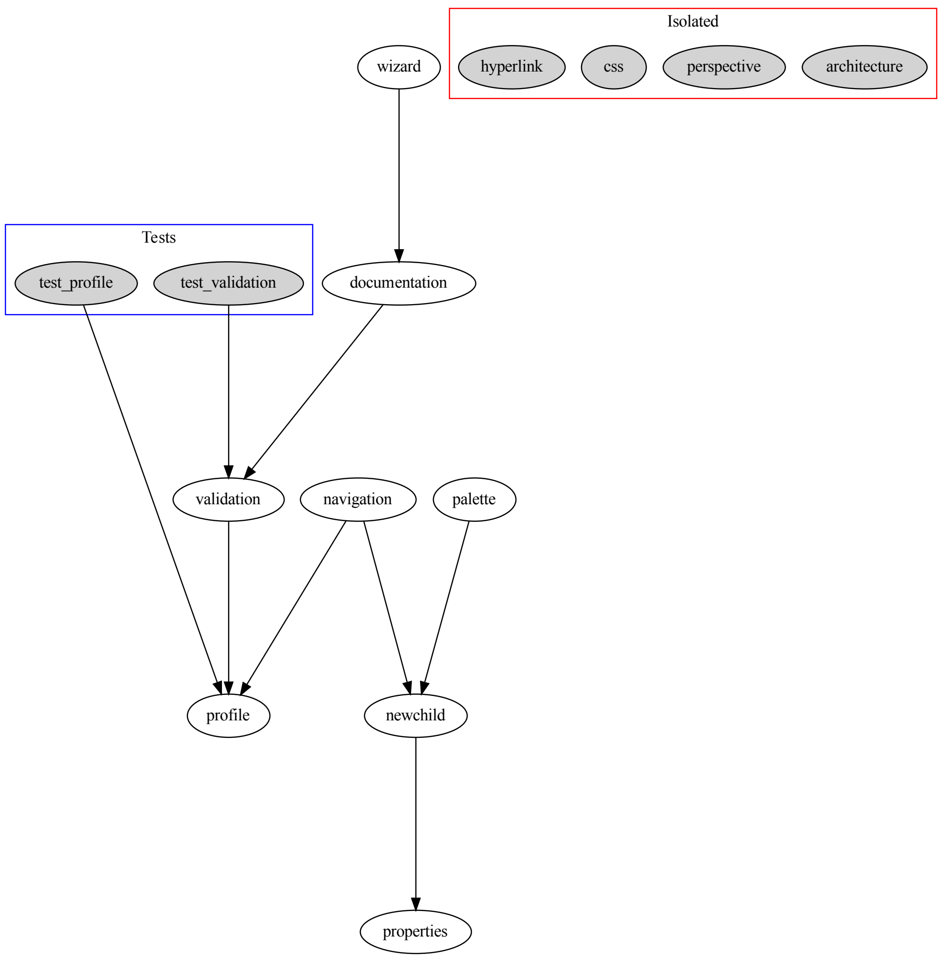


Figure B.1: Plugin dependency graph for all ACS plugins. An arrow denotes a dependency between two plugins by pointing towards the dependency.

C Process Table

		Task					
		Research	Formalization	Implementation	Article	User Test	Report
Week	5	✓✓✓					
	6	✓✓✓					
	7		✓	✓	✓		
	8	✓	✓	✓			
	9		✓✓	✓			
	10	✓	✓	✓			
	11	✓	✓✓				
	12		✓	✓✓			
	13		✓	✓✓			
	14		✓	✓✓			
	15			✓	✓✓		
	16		✓	✓			✓
	17		✓	✓			✓
	18			✓			✓✓
	19			✓	✓✓		
	20				✓✓✓		
	21					✓✓	✓
	22					✓	✓✓
	23						✓✓✓
Total		9	12	15	8	3	10

Table C.1: Summary of the overall activities and the work spend on them. A check-mark represents a person dedicated to that task that week. The total is a break down of total(approximate) work(person/week) spent on each activity. A detailed break down of each week is available in the Journal (Appendix A)