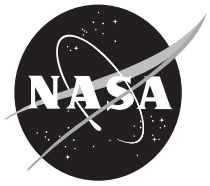
NASA/TM–2007–214155



Mise en Scene: Conversion of Scenarios to CSP Traces for the Requirements-to-Design-to-Code Project

John D. Carter, William B. Gardner, James L. Rash, and Michael G. Hinchey

September 2007

The NASA STI Program Office … in Profile

Since its founding, NASA has been dedicated to the advancement of aeronautics and space science. The NASA Scientific and Technical Information (STI) Program Office plays a key part in helping NASA maintain this important role.

The NASA STI Program Office is operated by Langley Research Center, the lead center for NASA's scientific and technical information. The NASA STI Program Office provides access to the NASA STI Database, the largest collection of aeronautical and space science STI in the world. The Program Office is also NASA's institutional mechanism for disseminating the results of its research and development activities. These results are published by NASA in the NASA STI Report Series, which includes the following report types:

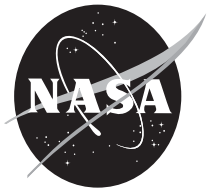
- TECHNICAL PUBLICATION. Reports of completed research or a major significant phase of research that present the results of NASA programs and include extensive data or theoretical analysis. Includes compilations of significant scientific and technical data and information deemed to be of continuing reference value. NASA's counterpart of peer-reviewed formal professional papers but has less stringent limitations on manuscript length and extent of graphic presentations.
- TECHNICAL MEMORANDUM. Scientific and technical findings that are preliminary or of specialized interest, e.g., quick release reports, working papers, and bibliographies that contain minimal annotation. Does not contain extensive analysis.
- CONTRACTOR REPORT. Scientific and technical findings by NASA-sponsored contractors and grantees.
- CONFERENCE PUBLICATION. Collected papers from scientific and technical conferences, symposia, seminars, or other meetings sponsored or cosponsored by NASA.
- SPECIAL PUBLICATION. Scientific, technical, or historical information from NASA programs, projects, and mission, often concerned with subjects having substantial public interest.
- TECHNICAL TRANSLATION. English-language translations of foreign scientific and technical material pertinent to NASA's mission.

Specialized services that complement the STI Program Office's diverse offerings include creating custom thesauri, building customized databases, organizing and publishing research results . . . even providing videos.

For more information about the NASA STI Program Office, see the following:

- Access the NASA STI Program Home Page at http://www.sti.nasa.gov/STI-homepage.html
- E-mail your question via the Internet to help@sti.nasa.gov
- Fax your question to the NASA Access Help Desk at (301) 621-0134
- Telephone the NASA Access Help Desk at (301) 621-0390
- Write to:
  NASA Access Help Desk
  NASA Center for AeroSpace Information
  7115 Standard Drive
  Hanover, MD 21076–1320

NASA/TM-2007-214155



Mise en Scene: Conversion of Scenarios to CSP Traces for the Requirements-to-Design-to-Code Project

John D. Carter and William B. Gardner
University of Guelph, Ontario, CANADA

James L. Rash and Michael G. Hinchey
NASA Goddard Space Flight Center, Greenbelt, Maryland

National Aeronautics and
Space Administration

Goddard Space Flight Center
Greenbelt, Maryland 20771

September 2007

Available from:

NASA Center for AeroSpace Information
7115 Standard Drive
Hanover, MD 21076-1320

National Technical Information Service
5285 Port Royal Road
Springfield, VA 22161

ABSTRACT

The "Requirements-to-Design-to-Code" (R2D2C) project at NASA's Goddard Space Flight Center is based on deriving a formal specification expressed in Communicating Sequential Processes (CSP) notation from system requirements supplied in the form of CSP traces. The traces, in turn, are to be extracted from scenarios, a user-friendly medium often used to describe the required behavior of computer systems under development. This work, called Mise en Scene, defines a new scenario medium (Scenario Notation Language, SNL) suitable for control-dominated systems, coupled with a two-stage process for automatic translation of scenarios to a new trace medium (Trace Notation Language, TNL) that encompasses CSP traces. Mise en Scene is offered as an initial solution to the problem of the scenarios-to-traces "D2" phase of R2D2C. A survey of the "scenario" concept and some case studies are also provided.

1. INTRODUCTION

Requirements to Design to Code (R2D2C), originated by NASA's Software Engineering Laboratory (Goddard Space Flight Center), is a requirements-based approach to system engineering. It can also be considered a "model-driven" or "model-based" methodology, but with the distinction that the model is derived automatically from the requirements instead of being created by a human designer. The goal is to make requirements the chief documentary artifact, and to derive a suitable implementation via several automated and semi-automated phases. Another distinctive feature is that the derived model is defined using a formal notation. This is useful for proving properties about the system and for guaranteeing that the derived implementation is functionally equivalent to the original requirements, i.e., the implementation is "correct by construction." In order that users having no special training in formal methods may freely use R2D2C, the formal model is intentionally kept "under the hood" of this methodology.

R2D2C's concepts have been described previously [HRR05b, HRR05a, HRR05c], and work is underway to specify and prototype the various phases of the design flow. An overview of the five-phase process is pictured in Figure 1. The work described in this Technical Memorandum, called **Mise en Scene** [Car06], is a prototype for phase D2, Traces Generator. To place this work in context, we first give an overview of R2D2C in Section 2 followed by a problem statement in Section 3. Section 4 is a survey of related work on scenarios. Section 5 describes Mise en Scene in detail, with case studies following in Sections 6 and 7. The last two sections contain future work and conclusions.

2. OVERVIEW OF R2D2C

In Figure 1, the R2D2C design flow begins with a set of requirements documents expressed in some manner of textual medium. From this set of requirements documents, the D1 phase, Scenarios Capture—in practice, a project-specific translator—creates a set of *scenarios* in a generic form not specific to any project. With the requirements having been massaged into a machine-readable, canonical form, the purpose of the next two phases is to convert them into an equivalent formal model.

The formal language used is Communicating Sequential Processes (CSP), a process algebra often used for modeling interprocess synchronization and communication for concurrent systems [Hoa78, Hoa83, Sch00]. The conversion of scenarios to a CSP specification is done in two phases: First, the D2 phase (the focus of this memorandum) creates a set of CSP *traces* that express the scenarios in a form compatible

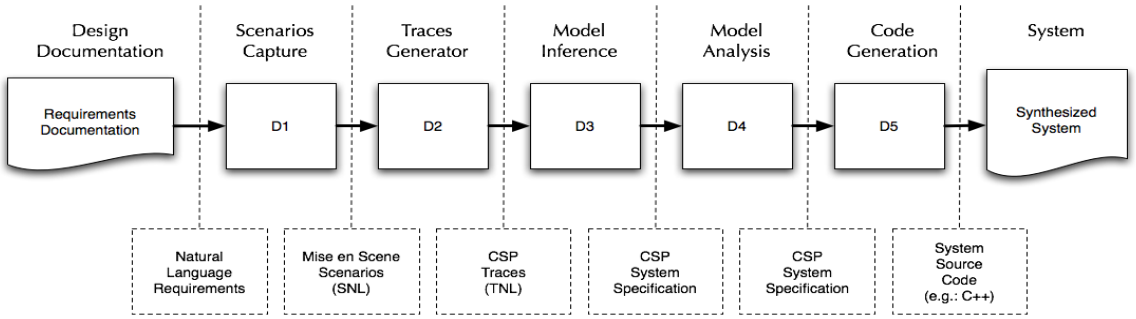


Fig. 1. Overview of the Requirements-to-Design-to-Code Design Flow

with CSP. In normal practice, traces would be derived from a CSP specification. However, in this case the D3 phase carries out the reverse derivation—inferring a specification from the given set of traces. This specification is the intermediate goal of the R2D2C design flow.

The D4 Model Analysis phase is provided to allow the specification to be formally analyzed, transformed, and/or optimized under a user's guidance. Finally, the D5 Code Generation phase synthesizes compilable source code from the CSP specification, or produces an implementation in some other suitable project-specific form.

The scope of this approach is limited to systems whose requirements are expressible as "scenarios," because the transformation path from scenarios via traces to a CSP specification is theoretically feasible. Before the work described herein was commenced, no precise definition of scenarios had been set down, nor any specific scenario medium defined for R2D2C. The contribution of this work is to present a narrative scenario medium inspired by existing scenario-based approaches that is suitable for bridging the gap between phases D1 and D3. The range of interpretations and approaches surveyed in the course of creating this solution is summarized in Section 4.

This new scenario-based approach, dubbed Mise en Scene (name explained below), specifically answers two questions:

- *What is the working definition of "scenario" for the R2D2C project?*
- *How shall such scenarios be mechanically converted to a set of CSP traces?*

The specifics of Mise en Scene's scenario medium, and the mechanics of its scenario-to-traces conversion method, will be described after a more precise explanation of the problem is given in the next section.

3. PROBLEM STATEMENT

The essential problem of designing the D2 phase is that of converting scenarios, which admit of a flexible definition, into CSP traces, which, in contrast, have a well-established definition. Thus, the input and output of the D2 phase can preliminarily be characterized as follows:

- A *scenario* is a sequence of steps that a system is required to carry out.
- A *trace* is a sequence of events that would be performed by a CSP specification during system execution.

In CSP, an *event* is an abstract name that can stand for any action the system takes or any stimulus that occurs in its environment. Data may optionally be communicated in conjunction with the event, in which case it is called a *channel.*

Superficially, the conversion of "steps" to "events" should be feasible, depending on what the steps encompass and how they are expressed. As for the input, R2D2C literature claims that the approach is applicable to systems specifiable by scenarios, yet is silent as to a preferred definition of scenarios. An important aspect of this work is to provide one.

Considering the target output, traces in CSP are represented as a list of events separated by commas, enclosed within a set of angle brackets (< >). A single trace, say <a, b, c>, meaning the sequence of the three named events, defines *one* permitted execution of a specification; *all* permitted executions are given as a set of traces. That set is represented as one or more comma-separated traces, enclosed within braces, e.g., { <>, <a>, <a,b> }. If a specification is viewed as a state machine, its set of traces represents all possible state transitions. Depending on the specification, an individual trace can be an infinite sequence, and the full set of traces may be infinite in size.

Trace events provide a kind of "black box" view of system execution. For example, CSP processes may specify input or output on named *channels*. Channels are used for interprocess communication, and, in an implementation, interface the system with its environment. Specifications may include channel I/O events such as `sensor?degrees`, meaning "read the channel named 'sensor' into a variable named 'degrees'," and `sensor!32`, meaning "output 32 on the sensor channel." The corresponding trace event that records the sensor communication event where the 32 is stored into degrees would be written in the form *channel.data:* `sensor.32`.

A side effect of CSP trace notation is that the direction of communication is lost. This is natural, because, while abstract processes (which engage in communication and synchronization amongst themselves) are an essential ingredient of CSP specifications, process identity disappears at the "black box" observation level; all an observer sees is the record of executed events and not the underlying process architecture. Stated another way, process specifications may provide clues about a possible implementation's architecture, but traces contain no such clues.

For R2D2C's design flow, this means that any architectural clues inherent in the scenarios are, in principle, discarded in the phase of converting to traces. Furthermore, the process architecture inferred by phase D3, and in turn synthesized into an implementation by D5, may bear no relation to an architecture suggested by the input scenarios. On the one hand, this may be of no consequence to the R2D2C user, since the entire formal model—CSP traces and equivalent specifications—is, as mentioned earlier, intentionally kept for internal use. On the other hand, throwing away useful data may serve to make the job of inferring specifications that much harder, and may incline the model inference phase to create process architectures that lead to unintuitive implementations.

Next, the assumptions about the D2 phase's context are stated: The D1 phase, Scenarios Capture, is likely to be application specific. It is assumed that D1 will be capable of outputting "scenarios" using some specified syntax. For prototyping work, it is sufficient to implement D1 as a simple text editor.

The D3 Model Inference phase is the heart of the theoreti-

cal work, and may face challenges of computational complexity due to (a) the potentially enormous volume of traces needed to record a non-trivial system's behavioral requirements, and (b) the extensive processing required by D3's prospective theorem prover, ACL2 [KM07]. This work assumes that D3 will accept input in the form of conventional CSP trace notation, but anticipates there will be room for negotiation with an eventual D3 design so as to reduce the volume of trace input (i.e., notational shortcuts), and help D3 to infer features of the target system's requirements that would otherwise be thrown away during their conversion to traces. Therefore, this work is not overly concerned with producing a definitive and final form of D2 output at this time, since adjustments will likely be necessary as the requirements of D3 processing are firmed up in future work.

To summarize the problem, on the "left" input end, the output of the D1 phase wants to be in a form that software practitioners can recognize as "scenarios." On the "right" output end are CSP traces, a highly constrained medium. It would be useless to prescribe scenario constructs that cannot be converted to traces, therefore this work has focused solely on constructs that have a conceptual analog in trace notation**.** A subset of those constructs commonly appearing in scenario-based approaches, surveyed in the next section, have been adopted.

4. RELATED WORK ON SCENARIOS

The term *scenario*, as will be seen from the citations below, is generally used to mean an expressed, exemplary action of a proposed or existing system. Scenarios are expressed in a range of mediums, with textual and graphic being the most prevalent. Scenarios are narrative in form, describing a sequence of actions taking place between a system component and its environment. Actions or events in scenarios are fully or partially ordered. While these actions are performed sequentially, scenarios often contain "if-else"-style conditions and looping constructs. Scenarios have clearly defined start and end points, which serve to place them within the operational context of the entire system. Scenario end points are usually related to a single goal, which may or may not be achieved upon termination of the scenario. The scenario goal may be further decomposed into lesser "subgoals," achieved through intermediate actions contained within the scenario.

A *scenario-based approach* is any software engineering or requirements engineering methodology making use of scenarios as a process artifact. Within software engineering, scenarios arise as usage examples of a proposed system, as system test cases, or to describe a design context into which the final system must fit.

Scenarios are frequently used in requirements engineering [Car99, FKV91, KS98, Mai98, McG97] as a bridge between the realms of developer and customer. Scenarios provide a requirements capture medium, often serving as concrete examples to elicit further dialog about a proposed system, or as a type of "pseudocode" readily understandable by customers, written in their language, using customer terminology.

Different approaches use scenarios to varying degrees, ranging from relying on them as a primary design artifact, to using them as part of a larger set of practices and techniques. Within R2D2C, scenarios are used as a digestible intermediate form between diverse kinds of application-specific requirements documentation and the rigid structure of formal specifications.

For software and requirements engineering, as well as human-computer interaction (HCI), the term is used frequently, in a variety of contexts. Its exact meaning has been the focus of many debates [Cam92a, Cam92b, KK92, Wri92, YB92] and usage surveys [GC04, Mai98, RAC+98, Nar92, WPJH98, YB87]. Most practitioners do not claim that their meaning is "the one true meaning," but rather an individual interpretation that suits their task. Yet despite being a possible source of confusion, the term scenario is common, and generally employed without any preamble specifying its meaning, just as in the existing R2D2C literature. It is accurate to call the term "vague in definition and scope" [WPJH98].

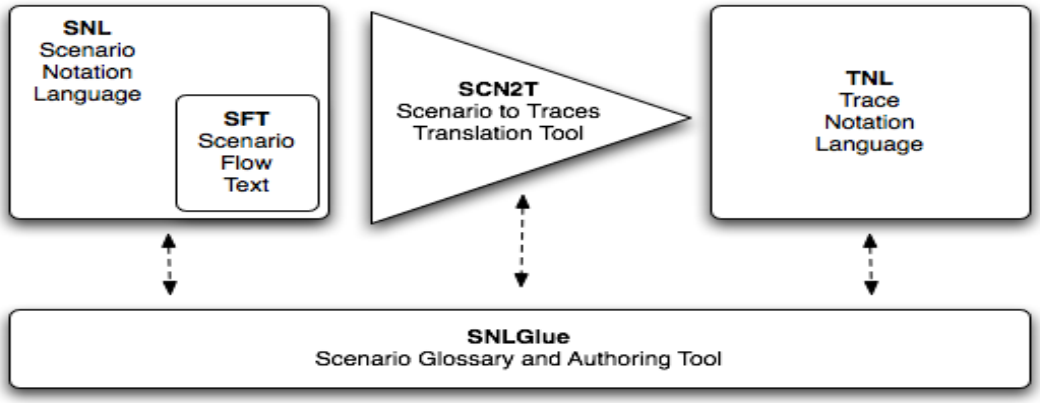


Fig. 2. General Overview of Mise en Scene and its Elements

5. MISE EN SCENE

As a result of studying the scenarios literature, a particular format was devised with the aim of being recognizable to scenario practitioners while having characteristics that can be translated to CSP traces. This design for R2D2C's D2 phase is called Mise en Scene, a term that comes from the field of film studies. Literally translating from French gives "putting in the scene," and the term is often used to describe everything visible to the camera within a shot, especially elements relevant to the narrative of the work. This title was chosen as a wry nod to ambiguity in terminology, since "scenario" is similarly ambiguous and prone to many interpretations.

Mise en Scene has four elements, illustrated in block diagram form in Figure 2, which together accomplish the work of the D2 phase:

1. Scenario Medium—Scenario Notation Language (SNL)

The medium created for Mise en Scene is a textual, form-based notation loosely based on Cockburn's guidelines for effective use cases [Coc01]. Whereas Cockburn states that a use case collects a number of scenarios describing related behavior, Mise en Scene does not follow this conceptual grouping. An SNL scenario is used to describe a single pattern of required execution in the proposed system. The system description may consist of multiple scenarios.

2. System for Connecting Scenarios—Scenario Glossary (SNLGlue)

A system represented as a collection of scenarios has the potential to be extremely disjoint, so as a means of connecting common elements in scenarios, a system-wide glossary is proposed, allowing a Mise en Scene user to intelligently view scenarios interactively, thus showing their interconnections with other scenarios. SNLGlue relates to the "data dictionary" concept employed in relational database management systems and in other system engineering approaches.

3. Trace Medium—Trace Notation Language (TNL)

Traces are syntactically simple, and require no specialized medium for the Mise en Scene approach. The conventional form of traces as used with the Formal Systems CSP tools [For] is applied in Mise en Scene, with the addition of an XML wrapper that provides the ability to store additional optional information used for Mise en Scene and R2D2C integration.

4. Mechanical Transformation Process—SCN2T

The cornerstone of Mise en Scene is the process by which scenarios represented in SNL can be converted to TNL. Section 5.4 lists rules for translating from the SNL scenario medium to a set of traces.

5.1 Scenario Schema

Following a study of the definitions and uses of "scenarios" in the literature, a set of attributes common to the majority of the approaches surveyed was identified. The result, listed in Table 1, is similar to Cockburn's description of a scenario [Coc01]. The attributes are sorted into two broad categories: *informational,* which likely serve a primarily documentary purpose, and *functional,* which ought to be reflected in any synthesized implementation and must therefore be translated to traces. Not all possible attributes were selected for Mise en Scene. Those marked with * are present in the Mise en Scene scenario schema (defined below) under those or similar names, while attributes marked with ** can optionally be entered in the Description for documentary purposes. They were not given dedicated fields for the sake of simplicity. Functional attributes that were not amenable to translation were not incorporated in the Mise en Scene scenario schema, but might be added in future work.

Table 1. Common scenario attributes

Common Name of Attribute	Description
Informational Attributes	
Name*	Title used to identify the scenario.
Description*	A textual overview of the actions contained in, and the actors involved in, the scenario.
Author*	Who created the scenario.
Actors*	Persons or systems involved in carrying out the scenario. Actors can be "Primary"—the actor who carries out the actions of the scenario—or "Secondary"—actors who aid the primary actor.
Revision History**	A method of tracking changes between scenario edits.
Stakeholders**	A description or list of persons affected by the design of or changes to the scenario.
Goals**	A scenario has one or more goals, which are carried out through a set of actions.
Subgoals**	Goals may be further decomposed into subgoals, which contain their own sets of actions.
Non-Functional Requirements**	Textual documentation containing requirements information that cannot be readily expressed in a functional form.

Table 1. Common scenario attributes (Cont.)

Common Name of Attribute	Description
Functional Attributes	
Precondition*	A description of the state the system must be in for a scenario to be eligible for execution.
Triggers*	The event or events that cause a scenario to be invoked.
Flow / Path*	The sequence of actions that constitutes successful execution.
Alternate Flows / Paths*	Flows of executions invoked by conditional statements within a scenario.
Extensions / Exceptions*	Flows of execution used to handle failures or other exceptional circumstances.
Priority	A classification attribute used to resolve non-determinism or scheduling when multiple scenarios can be invoked.
Constraints	A set of requirements for data or operations contained in the scenario; may be expressed in a formal syntax or natural language.
Success Guarantee	Conditions that are true upon successful termination of a scenario.
Minimal Guarantee	Conditions that are guaranteed to be true, in even the most disastrous failure.
Safety Properties	A description or list of things that cannot happen within the scenario. In performing any of this list, the scenario violates its safety properties, and is considered unsafe.

In the bullets below, each field in the Mise en Scene scenario schema is defined. Required fields are in **bold**, the others being optional. Along the way, concepts linking scenario attributes and CSP are interspersed. Readers who prefer to see concrete examples may wish to scan the case studies in Sections 6 and 7 in parallel with this section.

- **ID**—A name that uniquely identifies the scenario. Same as "Name" in Table 1.
- Description—A free-form textual description of the scenario and what it does. This serves as a comment for readers in addition to providing text that can be used for searching purposes.
- Author—The names of the author(s), and any other details.

The following two fields fulfill the purpose of the "Actors" attribute in Table 1. The Mise en Scene "component" is similar to the actor/role concept within Cockburn's treatment of use cases [Coc01], as well as to the use-case concept included in UML [BRJ99, Fow03]. A similar concept is seen in other approaches incorporating use cases [SDV96, KA06, KKAM06]. The chief difference is that a component is allowed to be internal or external to the system, whereas the actor/role concept places the actor outside the system. This nuance was introduced in an effort to create a concept similar to a use case's actors that better matches the notion of "processes" in CSP. As with use-case actors, components may also represent other systems or users of the system.

In scenarios, components are classified as primary or supplemental. (The term "secondary" was avoided to lessen the possible confusion with use-case secondary actors.)

- **Primary Component**—The identifier of the component that carries out all actions in this scenario. A scenario must only have a single component as its primary component, which is considered to be the initiator of the scenario.
- Supplemental Components—Identifiers of components referenced within the scenario, with which the primary component communicates and synchronizes. A scenario may have any number (including 0) of supplemental components.

In the two following attributes, Precondition and Trigger, the concept of a "task" is referenced. The smallest unit of execution that components carry out, intended to correspond to a CSP event, is called a *task*, defined by one step in the scenario's flow text. A component can only perform a single task at a time, though two components may cooperate in performing a single task. When tasks in other scenarios are referenced for triggers and preconditions, they are qualified with the relevant component in the form *componentID*::*taskID*. While components and tasks are defined implicitly by mentioning them in scenarios, SNL also has statements for explicitly adding documentation to component names and task IDs. A task schema may optionally list the components that are allowed to perform it, and this will be enforced by Mise en Scene when processing scenario flow text.

In one sense, the scenarios cause the system to transition from state to state by performing individual tasks. Nonetheless, it is worth observing that CSP traces provide an event-based formalism, so the notion of "system state" is only implied and not explicit, as it would be in a state-based formalism (e.g., Statecharts). The underlying trace semantics impose an inherent limitation on the specification of preconditions: They can only refer to tasks, not to "states." For example, one cannot specify a condition that "The door is open"; instead, one specifies that "The OpenDoor task has been performed." If the action is reversible, the door's state could be specified by requiring that the number of OpenDoor events exceeds the number of CloseDoor events.

- **Precondition**—A partial description of the system state required before the scenario can be executed, represented

as a set of task identifiers. These tasks must have occurred before the scenario can be triggered. The preconditions of the system can be empty (specified as "none"), making the scenario always ready to be triggered (see next). This kind of precondition is fairly simple, and would not allow for the situation where a subsequent task neutralizes or reverses the referenced task. In future work, it is planned to expand preconditions to accept a logical predicate on multiple tasks.

- **Trigger**—The task ID of the system event that causes the scenario flow text to be "executed." The trigger, which is mandatory, in combination with the scenario's preconditions (if any) provide a guard: the behavior contained in a scenario may only occur after its trigger has passed. The most permissive trigger is the task ID System::start, which enables a scenario to be executed at any time during system execution. Once a scenario completes, it must be retriggered in order to execute again.
- **Scenario Flow**—An ordered set of steps expressed in a restricted syntax that specifies the behavior of the scenario as actions undertaken by the primary component, and communication/synchronization with its supplemental components. Cockburn calls this as the "main success scenario," or the case in which "nothing goes wrong" [Coc01]. As with use case authoring, the scenario flow should provide behavior for the nominal flow of execution, and exceptional circumstances are to be handled by scenario extensions. The syntax of the scenario flow text, SFT, is detailed in Section 5.2.
- Extensions—Analogous to subroutines or functions in a conventional programming language. Scenario extensions are written using SFT syntax, and do not contain preconditions or triggers. Extensions are executed by the SFT "extension" directive. Upon completion of an extension, control returns to the calling flow, allowing for extensions to call other extensions. A scenario may invoke only its own extensions, not extensions contained in other scenarios. A scenario extension is identified by the ScenarioID followed by the scope resolution operator (::) and the extension identifier.

Besides the nine attributes listed above, scenarios may contain an optional "preamble" field. As with a strongly-typed programming language, the purpose is to declare any variables that will be referenced by the scenario flow text. See Section 5.2.3 for more on variables.

5.2 Scenario Flow Text (SFT)

Scenario Flow Text, or SFT (pronounced "soft"), is a series of lines, each containing syntax known as a *step*. A step contains a task to perform and/or other measures such as conditionals or communication directives.

The SFT syntax is based upon simple natural language sentences. The SVDPI (Subject Verb Direct-object Prepositional-phrase Indirect-object) pattern, as used in much of the use case literature [Coc01, RA97, KKAM06], is employed with minor additions. SVDPI was chosen as a compromise between overly formal syntax and unrestricted natural language. An example is shown in Figure 3.

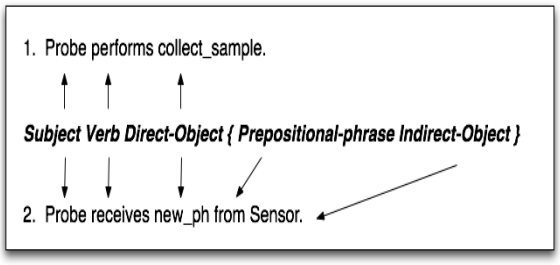


Fig. 3. Example of SFT Syntax using SVDPI Pattern

SFT steps may refer to three types of entities—components, tasks, and channels. Components and tasks were introduced in the previous section. As for channels, just as in CSP, they are one-way (simplex) channels with two fixed endpoints: the producer, who outputs to the channel, and the consumer who receives input from the channel. Channels have an associated set of data types that can be communicated along the channel. Operations on channels are considered non-buffered: a value may only be sent/received if the other endpoint is ready to receive/send the message.

A channel schema lists the data types that it communicates, and identifies its producer and consumer components; its ID is automatically derived from those three attributes. Since it is possible to define multiple channels between a pair of components, each channel can be augmented with a unique alias.

5.2.1 SFT Step Statements

The flow text of an SNL scenario is built using a series of SFT step statements. Every SFT statement is numbered, to provide a unique identifier for each step in the scenario. Specific steps of a scenario may be referenced using the following syntax:

```
ScenarioID::Step#N
```

Where:

- ScenarioID is a valid scenario identifier.
- N is valid step number within the scenario specified.

Within an extension (discussed below), steps are identified as follows:

```
ScenarioID::Extension::ExtensionID#N
```

In practice, the current version of SNL does not use step

references. This construct is reserved for future use with possible goto and looping steps (see Section 5.2.2).

The following paragraphs define the various kinds of step statements. Literal keywords and symbols are shown in bold font. The notation {X|Y|Z} indicates a choice among X, Y, or Z. Brackets [X] indicate that X is optional. Note that each step is terminated with a period.

Executional Step. The basic building block of SFT is the executional step. It is used to specify that the primary component of a scenario carries out a specific task. An executional step is performed in isolation—no communication with another component takes place. The syntax of the executional step is as follows:

```
ComponentID {performs|does|executes} TaskID.
```

Where:

- ComponentID is the identifier of the primary component of the scenario containing this step.
- The verb is one of the three specified synonyms.
- TaskID is the identifier of a task that is capable of being performed by component specified.

In order to generate an "empty step," TaskID may be given as **`System::nothing`**. The empty step is intended for use with conditionals to indicate that in some circumstance, nothing should be done.

Communication Step—Sending on an unspecified channel. There are two possible communication steps, one for sending and one for receiving. In this one, the primary component sends one or more values, in a single transfer event, to a supplemental component via a channel. The syntax of the send step is as follows:

```
ComponentID {sends|transmits|writes|outputs|
posts|puts} (Value1,Value2,..,ValueN-1,ValueN)
to SupplementalComponent.
```

Where:

- ComponentID is the identifier of the primary component of the scenario containing this step.
- The verb is any of the six specified synonyms.
- Data values are separated by commas. The enclosing parentheses are optional if only a single data value is sent.
- SupplementalComponent is the identifier of a supplemental component with whom the primary component can communicate.

In this version of the sending step, the channel is unspecified. If a channel exists that matches the endpoints and specified data types, that channel ID is automatically selected. If multiple candidates exist, then the channel must be specified using the next construct.

Communication Step—Sending on a specified channel. If multiple channels of the same type(s) exist between the same endpoints, then the channel alias must be specified. The syntax is as follows:

```
ComponentID {sends|transmits|writes|outputs|
posts|puts} (Value1,Value2,..,ValueN-1,ValueN)
to SupplementalComponent via ChannelAlias
```

Where:

- ChannelAlias is the alias of the channel to send on.

The other syntax elements are identical to the unspecified channel send.

Communication Step—Receiving on an unspecified channel. This step is the counterpart of the sending step. Its syntax is similar to the sending step, the change being the verb and preposition:

```
ComponentID {receives|reads|inputs|
obtains|gets} (Value1,Value2,..,ValueN-1,
ValueN) from SupplementalComponent.
```

If the channel alias must be specified to remove ambiguity, the following construct should be used.

Communication Step—Receiving on a specified channel. As with sending, an author can specify the channel for transmission as follows:

```
ComponentID {receives|reads|inputs|
obtains|gets} (Value1,Value2,..,ValueN-1,
ValueN) from SupplementalComponent
via ChannelAlias.
```

Conditional Step. The conditional step allows for one of two or more steps to be selected given a condition that is evaluated.

```
if Condition1 then STEP1.
[else if Condition2 then STEP2.]
[else if CondtionN-1 then STEPN-1.]
else STEPN.
```

Where:

- Condition1 through ConditionN-1 are conditional expressions.
- STEP1 through STEPN are valid steps, as specified using SFT syntax. These steps can also contain conditional steps, thus allowing for nesting of conditionals.
- Any number of else-if steps may be written.

- The last line must contain an else step, which is executed if all previous conditions have been evaluated as false.

In its simplest form, the conditional contains only an "if" step and an "else" step. Each line is evaluated in sequence, and the first true statement is executed. If all statements are false, the else is executed.

At present, Mise en Scene supports the following conditional expressions:

- Relational: **<, <=, ==, >, >=**
- Logical: **NOT, AND, OR, XOR**
- Event: `TaskID` **occurred** (true if TaskID has occurred in the environment of the scenario containing this conditional)

Extension Step. This step is used to invoke a scenario extension. Scenario extensions are written as separate scenario flow fragments, and used to handle exceptional cases. They are generally used with conditional steps, or used for organizational purposes when writing particularly long scenario flows. The syntax used to invoke a scenario extension is:

```
ComponentID invokes ExtensionID.
```

Where:

- Scenario is the scenario identifier.
- ExtensionID is an identifier of one of this scenario's extensions.

Upon completion of the extension, execution returns to the scenario flow, continuing with the next step in the flow.

Arithmetic Step. The arithmetic step is used to manipulate scenario variables (described in Section 5.2.3). Here, SFT syntax uses the familiar form of the C programming language. Although an arithmetic step may be composed of a number of statements, it is still considered a single task, making it indivisible. The syntax of the arithmetic step is as follows (in this case, the braces, shown in bold, are literal symbols):

```
ComponentID performs { assignments }.
```

Where:

- ComponentID is the primary component of the scenario.
- assignments are one or more assignment statements of the form *variable* **=** *expression,* written using the operators given below. The semicolon-separated statements must be enclosed within a set of braces.

At present, Mise en Scene provides the following arithmetic operators: **+** (addition), **-** (subtraction), **/** (division), ***** (multiplication), and **%** (modulo).

Rendezvous Step. The rendezvous step is a modified executional step that performs a task synchronized with another component. The synchronizing component must appear in the scenario's list of supplemental components. An executional step is made to be a rendezvous step by using the "with" keyword as follows:

```
Step with Component.
```

Where:

- Step is an executional step, written using valid SFT syntax.
- Component is a component appearing in the list of supplemental components for the scenario.

Note that every communication step is implicitly a rendezvous, so no "with" clause is required.

An author must be careful in specifying behavior using the rendezvous and communication steps, as they may introduce deadlock (mutual dependency) into a system. Within R2D2C, the D3 and D4 phases will recognize this and alert the author, but at present, Mise en Scene (the D2 step) does not diagnose this flaw.

5.2.2 Constructs omitted from SFT

Originally, a "goto" construct was introduced to allow the flow of execution to skip to another step in the scenario. This would be used for scenario "looping" (allowing a scenario to repeat portions of its behavior indefinitely or until a condition was satisfied). However, the construct has been removed from this initial version of SFT to avoid the possibility of infinite traces. In future work (see Section 8), infinite traces are cited as a topic for further exploration, and upon developing a trace medium suitable for infinite traces, this construct is likely to be reintroduced.

5.2.3 SFT and Variables

Mise en Scene allows for variables to exist at two scopes: scenario and system. Scenario variables may only be used within the scenario that contains them, whereas System Variables can be accessed (read-only) by any scenarios within the system. The latter allows for parameters to be defined to provide system-wide configuration information. Just as CSP has no "global variables," system variables cannot be used to communicate between components; channels must be used for that purpose. Variables are declared in the system preamble and scenario preambles. See the Section 6 and Section 7 case studies for examples of both kinds of preambles.

Variables are used for channel operations to send or receive data, and are necessary for arithmetic and conditional syntax. SFT supports the following built-in variable types: integer, character, float, string, bit, boolean. Additionally, SFT

provides the ability to add custom variable types to a specification in the system preamble. Type information is contained in the outgoing trace representation, discussed next.

5.3 Representing Traces

The common form of traces as used with the Formal Systems CSP tools, FDR and Probe [For], is also utilized in Mise en Scene, with the addition of an XML wrapper that associates a set of traces with the scenario(s) from which they were derived and enables additional information to be passed to the next phase of R2D2C alongside the trace set. This is called Trace Notation Language (TNL). Additional information included in this wrapper are:

- **Scenario Identifier**—The ID of the scenario(s) from which the set of traces was derived.
- **Component Identifier**—The identifier(s) of the system component to which this set of traces belongs.
- **Trace Set**—The set of traces for the system component, represented using the aforementioned trace notation.
- **Types**—Definition of types and ranges of variables used within this set of traces.

In order to reduce the volume of traces output, Mise en Scene distinguishes between "terminal" traces—those that contain all the events recording a system execution from start to finish—and "non-terminal" traces. The latter are prefixes of a terminal trace, and record any execution short of the end. (These definitions rule out infinite traces, which are the subject of future work.) From the CSP standpoint, a system's traces must include all possible terminal and non-terminal traces, as well as the empty trace < > that represents the system before it does anything. But from the computing standpoint, the non-terminal traces are purely redundant and would needlessly bulk up the output of the D2 phase. Therefore, the SCN2T conversion process generates only the terminal traces from a system's scenarios. Non-terminal traces can be easily derived by the D3 phase, should it require them.

TNL differs from traditional CSP traces in another manner: variables. As properly defined, channel communication events appearing in traces do not contain variables, but only containing actual literal values. This may lead to a state space explosion, as all combinations of valid data for variables must be contained in a process's traces. To simplify the TNL medium, variable place holders have been introduced. These place holders have types and value ranges associated with them, allowing the complete variable expansion to take place at the D3 phase if required.

Since conditional expressions and arithmetic operations cannot be directly expressed in conventional CSP trace notation, Mise en Scene encodes this information in the guise of trace events. Without the ability to pass these operations to D3 and later phases, calculations contained in the scenario would be lost and not able to be synthesized into an executable implementation.

5.4 Translation Algorithms

The cornerstone of Mise en Scene is the process by which scenarios represented in SNL are translated to TNL. This process is composed of two sub-problems. The first is converting individual scenarios into sets of equivalent traces. This is handled by mapping each line in a scenario to one or more events in the generated traces. The second, and larger, problem is composing the multiple sets of individual component traces into a set of system traces. This is more difficult than single scenario to trace translation due to the need to satisfy constraints of multiple scenarios, the possibility of combinatorial explosion of traces, and the resulting large data sets.

This section starts by describing the rules used to translate from a single scenario's SFT to TNL, and then explains how to generate traces from scenarios in combination.

5.4.1 Individual Scenario Translation

Table 2 lists the elements of SFT syntax by step type, and describes the corresponding trace output. Each of these rules is applied to every step in a scenario to generate all possible traces for that scenario.

Table 2. Translating steps of SFT

Step Type	Effect on Trace Output
Executional	States that a system component performs an action, thus a single event is appended to the trace output.
Communication	Generates a *channel.data* event appended to the trace output. The channel event contains the sender and receiver of the channel as well as the data type.
Conditional	Creates a set of events equal to the number of branches in the conditional statement. Each set of events resulting from the conditional syntax is appended to every trace preceding the conditional statement. The branch can be of any step type.
Extensions	The traces for an extension invocation are generated (according to the steps they contain) and are appended to the trace output.
Arithmetic	Creates an event that denotes the arithmetic operation, and the values (variables or constants) referenced by the operation.
Rendezvous	At the scenario level, rendezvous syntax has no additional effect on the trace output.

5.4.2 Parallel Composition of Scenarios

After the translation of individual scenarios, and appending sequentially composed scenarios, comes the task of combining scenarios in parallel. The trigger attribute determines how scenarios combine. If two scenarios share a trigger, they are eligible to be executed in parallel. If a scenario is triggered by another scenario's event, the former scenario is composed sequentially with the event contained in the latter scenario.

Parallel execution implies that the order of tasks within each individual scenario must be preserved, but the relative order between scenarios is not important, except where synchronization or communication occur. That is, if scenario P generates a trace <a,b,c> and scenario Q generates <d,e,f>, then when P and Q execute in parallel, any of ten possible permutations of <a,b,c,d,e,f> may arise such that a, b, and c, and d, e, and f, still occur in their original orders. This is what is meant by *interleaving* traces. If the scenarios share events in common for communication or synchronization purposes, then the combined trace contains only a single instance of the shared events. That is, if P generates <a,x,b> and Q generates <d,e,x>, where x is a shared event, the combined traces are the four permutations of <a,d,e,x,b> where a, x, and b, and d, e, and x, occur in their original orders.

Precondition and trigger attributes play an important role in the creation of system traces. Preconditions filter the set of terminal traces to remove all traces that do not contain the events specified by the precondition of the scenario. Triggers are used to determine the order of scenario execution.

The generation of system traces (representing the parallel composition of scenarios) is carried out using an eight-step algorithm:

- **Step 1**—The set of traces for each scenario should be generated using the previously discussed rules for each scenario, and then all eligible scenarios are combined sequentially.
- **Step 2**—Create a set of tuples S, whose members have the following composition:

```
SCN = <ScenarioIDs, ComponentID, Trigger,
        Precondition, Traces, Syncs>
```

  Populate this set by creating a tuple from each one of the scenarios and its derived traces. Each field is explained next:

  - ScenarioIDs: The IDs of all scenarios the set of traces was derived from. Initially this starts as a single scenario ID.
  - ComponentID: The component that executes the traces contained in the tuple.
  - Trigger: The trigger of the scenario this tuple was created from.
  - Precondition: The precondition of the scenario this tuple was created from.
  - Traces: The set of traces derived from the scenario.
  - Syncs: The list of events that this component rendezvous with other components to perform.

  Another tuple is introduced:

```
SYSTEMTRACEEVENT =
            <taskID,ScenarioIDsTriggered>
```

  This tuple is used to represent events in system traces. ScenarioIDsTriggered is a list used to store the IDs of scenarios that this taskID has already triggered, to prevent multiple triggering. These tuples are stored in a set, SYSTEM, which is initially empty.

- **Step 3—**Concatenate any scenarios triggered by another scenario's *Scenario*::success event. Mise en Scene allows especially long scenarios to be broken up into several smaller scenarios, composed sequentially using *PreviousScenario*::success as the trigger of subsequent scenarios.

  This is done as follows:

  1. Iterate over the set of tuples S, searching for a tuple T that contains a trigger of X::success where X is a scenario.
  2. Locate scenario X's tuple in S.
  3. Append all the traces of T to all of the traces for X. Append the ScenarioIDs of T to the scenarios IDs of X. Make the Syncs of X equal to the union of the Syncs for X and T.
  4. Remove T from S.
  5. Repeat steps 1-4 until no more success triggers are found.

- **Step 4—**Select a scenario that is eligible to run at system start. Locate a scenario R such that its trigger is System::start. Make the traces contained in SYSTEM equal to the traces of R. Remove this tuple R from S.
- **Step 5**—If possible, select another tuple R such that its trigger is System::start. Interleave R's traces with SYSTEM, taking into account any common SYNCs. Remove tuple R. Repeat this step until no more System::start triggers are found.
- **Step 6**—Find the first tuple Q in the set of tuples S with a trigger event that has occurred in SYSTEM. Verify that the trigger event selected has not already triggered this scenario. Each individual event in a trace in SYSTEM keeps a list of the ScenarioIDs it has triggered. Then do as follows:

  1. Interleave the traces of Q with the traces of SYSTEM, between the occurrence of Q's trigger, and the end of SYSTEM or Q being retriggered, whichever occurs first. Rendezvous with any SYNCS present in the system.
  2. Store the scenario ID of Q in the event inside the traces of SYSTEM that triggered Q.

Repeat this step until no more scenarios can be triggered.

- **Step 7**—Using the preconditions of scenarios contained in S, and the traces in SYSTEM, prune any traces with events occurring before a precondition has been met. By having each event in a trace in SYSTEM keep track of scenarios it has triggered, the system is able to perform this reduction after building the set of system traces, thus simplifying step 6.
- **Step 8**—The above steps generate the set of all terminal traces, containing all possible interleavings. If non-terminal traces are needed, calculate all the prefixes of all traces contained in SYSTEM, and union this set with SYSTEM.

A final, optional step is variable expansion. True CSP traces do not contain "variable placeholders" like TNL traces do; instead, the trace set contains all traces for all possible combinations of variables' values. This expansion is easily performed, but omitted from Mise en Scene as it may increase the size of the trace set by several orders of magnitude, and is likely to be more hindrance than help to the D3 phase.

As the current conception of Mise en Scene does not allow for infinite traces, one is not able to specify two scenarios, A and B, that re-trigger each other, creating a ping-pong effect. Infinite traces and their effect on system specifications are a focus of future work.

5.5 SNLGlue

In the scenario-based approaches surveyed, a major difficulty was connecting, collecting, and categorizing scenarios. To assist a scenario's authors, a scenario editor is envisioned possessing automated component and task highlighting, and the ability to query the system and obtain a clear view of interaction between components. This is the fourth ingredient of Mise en Scene, the element that unites the other three (SNL, TNL, and SCN2T). SNLGlue is a software tool, rather than another language, likely to be added during integration with R2D2C. It is seen as being particularly helpful for users who wish to dispense with the D1 phase and directly author scenarios using SNL.

6. PREVIOUS CASE STUDIES

In order to confirm that the expressive capabilities of Mise en Scene's scenario medium are at least sufficient to handle the few published R2D2C examples, these were represented using SNL. Below is the "Page Analyst" scenario taken from the LOGOS/ANTS system [HRR05c], reworked in terms of SNL:

```
Scenario ID: RequestPagerInfo
Scenario Description: Requests the pager
information for an analyst and sends the request
to the DatabaseAgent. Presented on pg. 11 of
NASA/TM--2005--212774
Author ID: GSFC
Primary Component: PagerAgent
Supplemental Components: UIAgent,DatabaseAgent
Preconditions: None
Trigger:
UIAgent_PagerAgent_PAGERINFOTYPE_request.
requestinfo
Preamble:
{
  PAGERINFOTYPE requestinfo;
  ANALYSTINFOTYPE analystinfo;
}
Scenario Flow:
1. PagerAgent sends requestinfo to DatabaseAgent
via QUERY.
2. PagerAgent receives analystinfo from
DatabaseAgent via RESULT.
3. PagerAgent performs createandStoreMessage.
Scenario Extensions: None
```

The RequestPagerInfo scenario defines the tasks carried out by the PagerAgent to retrieve an analyst's contact information and page them using said information. The scenario is triggered by a request from the UIAgent.

The traces resulting from the SCN2T translation algorithm are:

```
<PagerAgent_DatabaseAgent_PAGERINFOTYPE_QUERY.r
equestinfo,
DatabaseAgent_PagerAgent_ANALYSTINFOTYPE_RESULT
.analystinfo, createandStoreMessage>
```

SNL versions of the other case studies from existing R2D2C publications may be found in Appendix A of [Car06].

Since the existing examples were brief and did not exercise the richness of SNL's syntactic constructs, a somewhat larger control-dominated system was created as a case study.

7. ROBOT PROBE CASE STUDY

This example presents a set of scenarios for a robotic probe designed to conduct soil surveys. Only the scenarios for

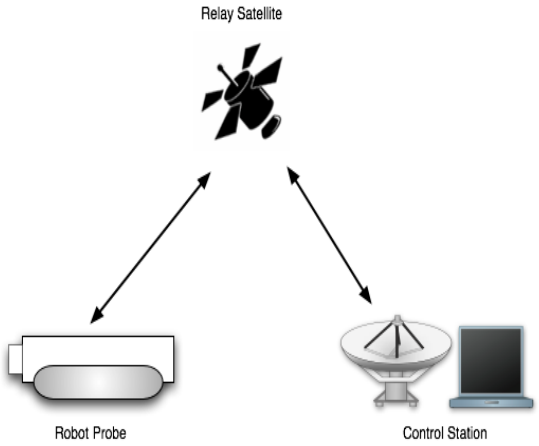


Fig. 4. System Architecture of Robot Probe System.

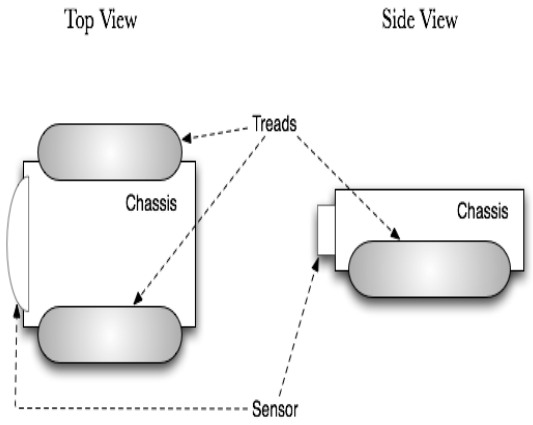


Fig. 5. Robot Probe Top and Side Views.

the probe itself are presented. The probe is controlled by a command station component ("Station"). Figure 4 shows an overview of the entire system, and Figure 5 is a sketch of the probe itself.

The probe contains two treads both capable of forward and backward movement, allowing the probe to turn clockwise, turn counter-clockwise, move forward, and move backward, as shown in Figure 6. The front of the probe contains a sensor that reads the pH and water content of the soil at the probe's current location. The sensor and the two treads are controlled by three controllers, shown in Figure 7. The probe receives commands from and relays data to the system component Station, not shown. This example presents all permitted interactions with Station, which is considered to be in the environment of Probe.

The scenarios in this example rely heavily on the extension/invoke construct, and also demonstrate the use of unnamed channels.

In the following sections the set of traces for each possible Probe instruction is derived. Due to the size of the trace set, it is abbreviated.

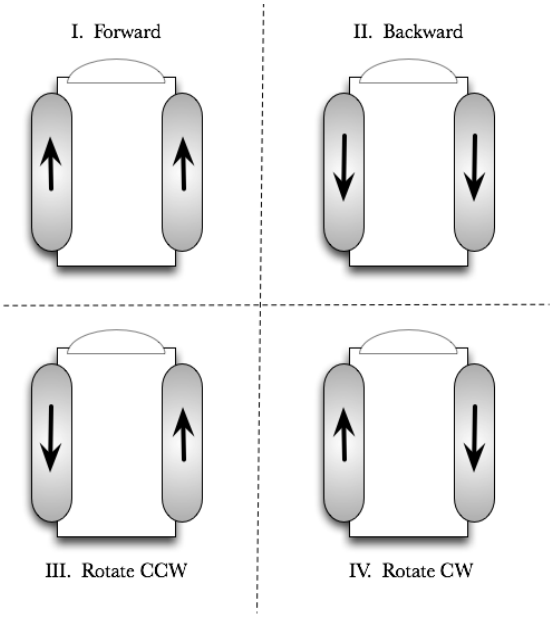


Fig. 6. Robot Probe Movement Commands showing tread directions.

7.1 System Preamble

Shown below is the system preamble for the robot probe system. Several custom data types have been added for the representation of x,y coordinates, probe commands, pH and water sample values, and electromechanical instructions to the probe's treads.

```
System::preamble
{
nametype integer = {-1024..1024}
nametype string = {0..512}
nametype character = {0..255}
nametype float = {-1000000.00..1000000.00}
nametype bit = {0..1}
nametype boolean = {0..1}
nametype COORDINATE = {-1000..1000)
nametype COMMANDTYPE = {0..4}
nametype DIRECTIONCOMMAND = {0..3}
nametype PHSAMPLE = {0..8}
nametype WATERSAMPLE = {0..100}
nametype TREADDIRECTION = {0..1}
}
```

Fig. 8. System Preamble for Robot Probe.

Next, the schemas for all components of the Robot Probe are given:

Component ID: Station
Description: The command station relaying directional commands to the probe, and the recipient of data collected from the sensor.

Component ID: Probe

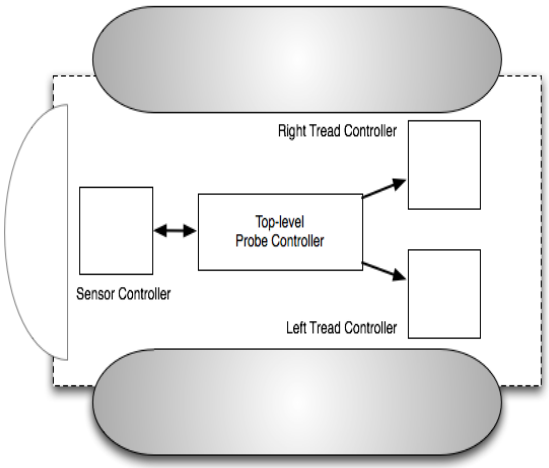


Fig. 7. Robot Probe Internal View of Major System Components.

```
Description: A two tread, 'tank-style' robotic
probe that is controlled remotely by a contol
station. The probe collects soil samples and
transmits this data to the control station.

Component ID: MotorControl
Description: An integrated controller for the
probe's two independent treads. Receives
commands from the probe and send electro-
mechanical instructions and synchronization
instructions to each of the probe's treads.
Component ID: LeftTread
Description: The tread on the left side of the
probe, controlled by MotorControl.

Component ID: RightTread
Description: The tread on the left side of the
probe, controlled by MotorControl.

Component ID: Sensor
Description: The sensor on the front of the
probe that collects pH and water data from the
terrain. Sensor is controlled by the probe, and
relays data back to the control station.
```

Fig. 9. Schemas for Robot Probe components

Next are a number of scenarios that describe the behavior of the robot probe, beginning with probe initialization:

```
Scenario ID: RobotStart
Description: Tasks performed by the robotic
probe upon system start.
Author: John Carter
Primary Component: Probe
Supplemental Component: Station
Precondition: None
Trigger: System::start
Preamble:
{
  COORDINATE xloc;
  COORDINATE yloc;
}
Scenario Flow:
1.  Probe performs robot_initialize.
2.  Probe sends xloc to Station.
3.  Probe sends yloc to Station.
4.  Probe performs robot_ready.

Extensions: None
```

Fig. 10. Scenario describing Probe initialization

The following scenario outlines the flow of execution for receiving and processing a command from Station:

```
Scenario ID: RobotCommand
Description: The probe receives a command to
turn, move, or collect a sample from Station and
executes it.
Author: John Carter
Primary Component: Probe
Supplemental Components: Station, MotorControl,
Sensor
Precondition: Probe::robot_ready
Trigger: Station::robot_command
Preamble: {
  COMMANDTYPE cmd;
  PHSAMPLE new_ph;
  WATERSAMPLE new_water;
}
Scenario Flow:
1.  Probe receives cmd from Station via command.
2.  if (cmd == 0) Probe invokes Forward.
    else if (cmd == 1) Probe invokes TurnRight.
    else if (cmd == 2) Probe invokes Backward.
    else if (cmd == 3) Probe invokes TurnLeft.
    else Probe invokes CollectData.
3.  Probe performs acknowledged.

Extension ID: RobotCommand::Extension::Forward
Description: Commands MotorControl to move robot
forward.
1.  Probe performs ready_move.
2.  Probe sends 0 to MotorControl.
3.  Probe performs move_complete with
MotorControl.

Extension ID:
RobotCommand::Extension::TurnRight
Description: Commands MotorControl to turn robot
CW.
1.  Probe performs ready_move.
2.  Probe sends 1 to MotorControl.
3.  Probe performs move_complete with
MotorControl..

Extension ID: RobotCommand::Extension::Backward
Description: Commands MotorControl to move robot
backward.
1.  Probe performs ready_move.
2.  Probe sends 2 to MotorControl.
3.  Probe performs move_complete with
MotorControl.

Extension ID: RobotCommand::Extension::TurnLeft
Description: Commands MotorControl to turn robot
CCW.
1.  Probe performs ready_move.
2.  Probe sends 3 to MotorControl.
3.  Probe performs move_complete with
MotorControl.

Extension ID:
RobotCommand::Extension::CollectData
Description: Collects a set of samples (pH and
water) from the sensor.
1. Probe performs collect_sample.
2. Probe receives new_ph from Sensor.
3. Probe receives new_water from Sensor.
4. Probe sends new_ph to Station.
5. Probe sends new_water to Station.
6. Probe performs sample_done with Sensor.
```

Fig. 11. Scenario showing Probe processing a command from Station

The following scenario collects and forwards data:

```
Scenario ID: Sensor_Collect
Description: Collects a pH and water sample from
the soil at current location.
```

```
Primary Component: Sensor
Supplemental Components: Probe
Precondition: Probe::robot_initialize
Trigger: Probe::collect_sample
Preamble: {
  PHSAMPLE ph;
  WATERSAMPLE water;
}
Scenario Flow:
1.  Sensor performs collect_ph.
2.  Sensor performs collect_water.
3.  Sensor sends ph to Probe.
4.  Sensor sends water to Probe.
3.  Sensor performs sample_done with Probe.

Extensions: None
```

Fig. 12. Scenario describing the Sensor component collecting a sample

The Move_Probe scenario relays commands from the Probe to the independent treads:

```
Scenario ID: Move_Probe
Description: Processes movement commands from
the probe.
Primary Component: MotorControl
Supplemental Component: Probe
Preconditions: None
Trigger: Probe::ready_move
Preamble:
{
  DIRECTIONCOMMAND direction_cmd;
}
Scenario Flow:
1. MotorControl performs move_command.
2. MotorControl receives direction_cmd from
Probe.
3. if (direction_cmd == 0) MotorControl invokes
move_ahead.
    else if (direction_cmd == 1) MotorControl
invokes turn_right.
    else if (direction_cmd == 2) MotorControl
invokes move_back.
    else MotorControl invokes turn_left.
4. MotorControl performs disengage with
LeftTread.
5. MotorControl performs disengage with
RightTread.
6. MotorControl performs move_complete with
Probe.

Extensions:

Extension ID: Move_Probe::Extension::move_ahead
Description: Moves robot probe forward.
1. MotorControl sends 1 to LeftTread via LTDIR.
2. MotorControl sends 1 to RightTread via RTDIR.

Extension ID: Move_Probe::Extension::turn_right
Description: Turns robot probe clockwise.
1. MotorControl sends 1 to LeftTread via LTDIR.
2. MotorControl sends 0 to RightTread via RTDIR.

Extension ID: Move_Probe::Extension::move_back
Description: Moves robot probe backward.
1. MotorControl sends 0 to LeftTread via LTDIR.
2. MotorControl sends 0 to RightTread via RTDIR.

Extension ID: Move_Probe::Extension::turn_left
Description: Moves robot probe counter-
clockwise.
1. MotorControl sends 0 to LeftTread via LTDIR.
2. MotorControl sends 1 to RightTread via RTDIR.
Scenario description of the MotorControl.
```

Fig. 13. Scenario for the MotorControl

Finally, the scenarios describing the behavior for one of the independent treads are presented. The behavior between the two scenarios is similar, hence only of the two treads is included:

```
Scenario ID: LeftTread_Movement
Description: Controls the left tread on a tank-
style robot.
Primary Component: LeftTread
Supplemental Components: MotorControl,
RightTread
Precondition: Probe::robot_initialize
Trigger: MotorControl:move_command
Preamble:
{
  TREADDIRECTION lt_direction;
}

Scenario Flow:
Trigger: MotorControl:move_command
1. LeftTread receives lt_direction from
MotorControl via LTDIR.
2. if (lt_direction == 1) LeftTread performs
lt_setforward.
    else LeftTread performs lt_setbackward.
3. LeftTread performs direction_engage with
RightTread.
4. LeftTread performs move with RightTread.
5. LeftTread performs disengage with
MotorControl.

Extensions: None
Scenario for the behavior of the probe's left
tread.
```

Fig. 14. Scenario for the Probe's left tread

Next we perform the translation of all individual traces.

7.2 Individual Scenario Traces

The trace resulting from translating RobotStart is:

```
terminal traces (Probe) = {
<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready> }
```

The traces resulting from the RobotCommand scenario and its extensions are:

```
terminal traces (Probe) = {
<Station_Probe_COMMANDTYPE.cmd,op_equal.cmd.0.1
```

```
,ready_move,Probe_MotorControl_DIRECTIONCOMMAND
.0,move_complete,acknowledged>,
<Station_Probe_COMMANDTYPE.cmd,op_equal.cmd.1.1
,ready_move,Probe_MotorControl_DIRECTIONCOMMAND
.1,move_complete,acknowledged>,
<Station_Probe_COMMANDTYPE.cmd,op_equal.cmd.2.1
,ready_move,Probe_MotorControl_DIRECTIONCOMMAND
.2,move_complete,acknowledged>,
<Station_Probe_COMMANDTYPE.cmd,op_equal.cmd.3.1
,ready_move,Probe_MotorControl_DIRECTIONCOMMAND
.3,move_complete,acknowledged>,
<Station_Probe_COMMANDTYPE.cmd,op_equal.cmd.4.1
,collect_sample,Sensor_Probe_PHSAMPLE.new_ph,Se
nsor_Probe_WATERSAMPLE.new_water,
Probe_Station_PHSAMPLE.new_ph,Probe_Station_WAT
ERSAMPLE.new_water,sample_done,acknowledged>
}
```

The traces of the Sensor resulting from the Sensor_Collect scenario are:

```
terminal traces (SensorCollect) = {
<collect_ph,collect_water,Sensor_Probe_PHSAMPLE
.ph,Sensor_Probe_WATERSAMPLE.water,sample_done>
}
```

The traces of the MotorControl, created from the Move_Probe scenario:

```
terminal traces (MoveProbe) = {
<move_command,Probe_MotorControl_DIRECTIONCOMMA
ND.direction_cmd,op_equal.direction_cmd.0.1,Mot
orControl_LeftTread_TREADDIRECTION_LTDIR.1,Moto
rControl_RightTread_TREADDIRECTION_TRDIR.1,dise
ngage,disengage,move_complete>,

<move_command,Probe_MotorControl_DIRECTIONCOMMA
ND.direction_cmd,op_equal.direction_cmd.0.2,Mot
orControl_LeftTread_TREADDIRECTION_LTDIR.1,Moto
rControl_RightTread_TREADDIRECTION_TRDIR.0,dise
ngage,disengage,move_complete>,

<move_command,Probe_MotorControl_DIRECTIONCOMMA
ND.direction_cmd,op_equal.direction_cmd.0.3,Mot
orControl_LeftTread_TREADDIRECTION_LTDIR.0,Moto
rControl_RightTread_TREADDIRECTION_TRDIR.0,dise
ngage,disengage,move_complete>,

<move_command,Probe_MotorControl_DIRECTIONCOMMA
ND.direction_cmd,op_equal.direction_cmd.0.4,Mot
orControl_LeftTread_TREADDIRECTION_LTDIR.0,Moto
rControl_RightTread_TREADDIRECTION_TRDIR.1,dise
ngage,disengage,move_complete> }
```

The traces from MotorControl are:

```
terminal traces(LeftTread_Movement) = {
<MotorControl_LeftTread_TREADDIRECTION_LTDIR.lt
_direction,op_equal.lt_direction.1.1,lt_setforw
ard,direction_engage,move,disengage>
<MotorControl_LeftTread_TREADDIRECTION_LTDIR.lt
_direction,op_equal.lt_direction.0.1,lt_setback
ward,direction_engage,move,disengage> }

terminal traces(RightTread_Movement) = {
<MotorControl_RightTread_TREADDIRECTION_RTDIR.r
t_direction,op_equal.rt_direction.1.1,rt_setfor
ward,direction_engage,move,disengage>
<MotorControl_RightTread_TREADDIRECTION_RTDIR.r
t_direction,op_equal.rt_direction.0.1,rt_setbac
kward,direction_engage,move,disengage> }
```

7.3 System Traces

Next, individual traces of each Probe component are combined into a set of traces representing the behavior of the entire system. Recall that Station is considered part of the environment of the Probe, so all possible interactions with Station must be included.

After the system executes System::start, only one scenario is eligible to run, RobotStart, so begin with its trace:

```
traces (System) = {
<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready...>
}
```

After executing this series of events, no more scenarios can execute, due to the RobotCommand triggering on Station::robot_command. By decoupling Station from this system for the purposes of this example, it is assumed that the Station performs this immediately after robot_ready. This allows the example to incorporate the set of traces arising from RobotCommand, and these are appended to the previous trace, along the set of possible commands originating from Station:

```
traces (System) = {
<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.0.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.0,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.1.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.1,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.2.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.2,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.3.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.3,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,<Station_Probe_COMMANDTYPE.cmd,op_equ
```

```
al.cmd.4.1,collect_sample,Sensor_Probe_PHSAMPLE
.new_ph,Sensor_Probe_WATERSAMPLE.new_water,
Probe_Station_PHSAMPLE.new_ph,Probe_Station_WAT
ERSAMPLE.new_water,sample_done,acknowledged...>
}
```

Next, Sensor's traces are incorporated, which are only appended to the last traces in the previous set, as it shows the flow of execution when Station requests a sample. After inserting Sensor's events between Probe and Sensor's rendezvous, the trace set becomes:

```
traces (System) = {
<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.0.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.0,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.1.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.1,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.2.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.2,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.3.1,ready_move,Probe_MotorControl_DIRECTI
ONCOMMAND.3,move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command, <Station_Probe_COMMANDTYPE.cmd,
op_equal.cmd.4.1,collect_sample,
collect_ph,collect_water,Sensor_Probe_PHSAMPLE.
new_ph,
Sensor_Probe_WATERSAMPLE.new_water,Probe_Statio
n_PHSAMPLE.new_ph,Probe_Station_WATERSAMPLE.new
_water,sample_done,acknowledged...> }
```

Next, the traces of Motor_Control are incorporated into the first four traces of the previous set, which represent the move commands:

```
traces (System) = {
<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.0.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.0,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.1,
MotorControl_RightTread_TREADDIRECTION_TRDIR.1,
disengage, disengage, move_complete,
acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.1.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.1,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.1,
MotorControl_RightTread_TREADDIRECTION_TRDIR.0,
disengage, disengage, move_complete,
acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.2.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.2,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.0,
MotorControl_RightTread_TREADDIRECTION_TRDIR.0,
disengage, disengage,
move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.3.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.3,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.0,
MotorControl_RightTread_TREADDIRECTION_TRDIR.1,
disengage, disengage,
move_complete,acknowledged...>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command, <Station_Probe_COMMANDTYPE.cmd,
op_equal.cmd.4.1,collect_sample,
collect_ph,collect_water,Sensor_Probe_PHSAMPLE.
new_ph,
Sensor_Probe_WATERSAMPLE.new_water,Probe_Statio
n_PHSAMPLE.new_ph,Probe_Station_WATERSAMPLE.new
_water,sample_done,acknowledged...> }
```

Finally, the events of the left and right treads are added. First, the interleavings between the LeftTread and RightTread are performed. All of the events in their traces are rendezvous, except for lt_setforward, lt_setbackward, rt_setforward, rt_setbackward. Each of the movements (first four traces) in the above set, uses one of lt_ and rt_. The order in which these events occur is arbitrary, since the treads engage, move, and disengage together. From this, add the two possible interleavings for each, and arrive at:

```
traces (System) = {
<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.0.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.0,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.1,
MotorControl_RightTread_TREADDIRECTION_TRDIR.1,
lt_setforward, rt_setfoward, direction_engage,
move, disengage, disengage, move_complete,
acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.1.1,ready_move,move_command,Probe_MotorCo
```

```
ntrol_DIRECTIONCOMMAND.1,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.1,
MotorControl_RightTread_TREADDIRECTION_TRDIR.0,
lt_setforward, rt_setbackward,
direction_engage, move, disengage, disengage,
move_complete, acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.2.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.2,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.0,
MotorControl_RightTread_TREADDIRECTION_TRDIR.0,
lt_setbackward, rt_setbackward,
direction_engage, move, disengage, disengage,
move_complete,acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.3.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.3,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.0,
MotorControl_RightTread_TREADDIRECTION_TRDIR.1,
lt_setbackward, rt_setforward,
direction_engage, move, disengage, disengage,
move_complete,acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.0.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.0,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.1,
MotorControl_RightTread_TREADDIRECTION_TRDIR.1,
rt_setforward, lt_setfoward, direction_engage,
move, disengage, disengage, move_complete,
acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.1.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.1,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.1,
MotorControl_RightTread_TREADDIRECTION_TRDIR.0,
lt_setforward, rt_setbackward,
direction_engage, move, disengage, disengage,
move_complete, acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.2.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.2,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.0,
MotorControl_RightTread_TREADDIRECTION_TRDIR.0,
rt_setbackward, lt_setbackward,
direction_engage, move, disengage, disengage,
move_complete,acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command,Station_Probe_COMMANDTYPE.cmd,op_equa
l.cmd.3.1,ready_move,move_command,Probe_MotorCo
ntrol_DIRECTIONCOMMAND.3,
MotorControl_LeftTread_TREADDIRECTION_LTDIR.0,
MotorControl_RightTread_TREADDIRECTION_TRDIR.1,
rt_setbackward, lt_setfoward, direction_engage,
move, disengage, disengage,
move_complete,acknowledged>,

<robot_initialize,Probe_Station_COORDINATE.xloc
,Probe_Station_COORDINATE.yloc,robot_ready,robo
t_command, <Station_Probe_COMMANDTYPE.cmd,
op_equal.cmd.4.1,collect_sample,
collect_ph,collect_water,Sensor_Probe_PHSAMPLE.
new_ph,
Sensor_Probe_WATERSAMPLE.new_water,Probe_Statio
n_PHSAMPLE.new_ph,Probe_Station_WATERSAMPLE.new
_water,sample_done,acknowledged> }
```

The listing of the above traces is tedious, but serves to illustrate the translation algorithm at work on the case study's scenarios.

8. FUTURE WORK

The largest open problem with respect to SCN2T is the issue of infinite traces. There may be a need for "while"-style looping within the scenario medium, though the priority of this need has yet to be investigated, but infinite traces will be a by-product of a looping construct. A number of case studies by trial users of Mise en Scene would likely highlight constructs missing from and required in the SNL medium.

Another area for further development is the implementation of a software prototype of SCN2T. This depends largely on R2D2C integration. During the course of this work, a number of C++ classes and utilities were written to automate calculations involving traces. These classes were developed with an eventual prototype in mind, and should serve as a good starting point for building a full implementation.

Another possible area of application for Mise en Scene is the generation of traces for use in formal verification. In order to verify trace refinement, a "safety specification" in the form of a process, or set of traces that defines all of the permitted system behavior, must be created. A tool such as Formal Systems' FDR2 is able to prove that a candidate CSP implementation falls within the set of behavior prescribed in the safety specification. Traces generated from natural language scenarios may represent a user-friendly route to creating the needed safety specifications.

R2D2C also includes a shortcut "S" flow whereby scenarios are converted directly to a subset of CSP, bypassing the traces generation and model inference phases. SNL to CSP conversion has been attempted, and success has been achieved in the area of single scenarios; however, more work is needed toward composing CSP processes derived from scenarios to form the top-level system.

9. CONCLUSION

This research addressed the challenge of defining "scenario" in an appropriate manner for the R2D2C project. The resulting SNL meets the objectives of (1) being recognizable as a "scenario-based approach," since it is compatible with widely surveyed usage, and (2) being translatable to CSP traces. Therefore, SNL can play its desired role in the D2 phase of R2D2C. This was demonstrated by reworking existing R2D2C case studies in terms of SNL, and by developing a new case study of a control-dominated system (robot probe) that would be within the intended scope of R2D2C.

Furthermore, a challenge of all scenario-based approaches is managing the fragmentation resulting from specifying a system as a set of loosely connected scenarios. It becomes difficult to compose the scenarios into a larger whole. With Mise en Scene this has been achieved by limiting the specification medium from natural language to structured text, and by employing the communication and synchronization paradigm of CSP. The resulting scenario medium is suitable for directly authoring scenarios for R2D2C even without the aid of a D1 Scenarios Capture phase, as shown by the robot probe case study.

Finally, Mise en Scene provides the necessary automatic means of translating from SNL scenarios to CSP traces, output in the form of TNL. While the need for some adaptation of TNL to an eventual implementation of the D3 Model Inference phase must be anticipated, Mise en Scene provides a path to go forward into the research on formal model extraction from CSP traces, thereby considerably advancing the work on R2D2C.

10. ACKNOWLEDGMENTS

This work was supported by research grants from Canada's Natural Science and Engineering Research Council (NSERC).

REFERENCES

[BRJ99] Grady Booch, James Rumbaugh, and Ivar Jacobson. *The Unified Modeling Language user guide*. Addison Wesley Longman Publishing Co., Inc., Redwood City, CA, USA, 1999.

[Cam92a] Robert L. Campbell. Categorizing scenarios: a quixotic quest? *SIGCHI Bull.*, 24(4):16–17, 1992.

[Cam92b] Robert L. Campbell. Will the real scenario please stand up? *SIGCHI Bull.*, 24(2):6–8, 1992.

[Car99] Paul B. Carpenter. Verification of requirements for safety-critical software. In *Proceedings of the 1999 annual ACM SIGAda international conference on Ada (SIGAda '99)*, pages 23–29, New York, NY, USA, 1999. ACM Press.

[Car06] John Douglas Carter. Mise en Scene: A scenario-based approach to generating CSP system traces. Master's thesis, University of Guelph, Dept. of Computing and Information Science, Guelph, ON, October 2006.

[Coc01] Alistair Cockburn. *Writing Effective Use Cases*. The Agile Software Development Series. Addison Wesley, Indianapolis, IN, USA, 2001.

[FKV91] M. D. Fraser, K. Kumar, and V. K. Vaishnavi. Informal and formal requirements specification languages: Bridging the gap. *IEEE Trans. Softw. Eng.*, 17(5):454–466, 1991.

[For] Formal Systems (Europe) Ltd. Formal Systems website [online, cited Oct. 1, 2006]. Available from: http://www.fsel.com.

[Fow03] Martin Fowler. *UML Distilled: A Brief Guide to the Standard Object Modeling Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2003.

[GC04] Kentaro Go and John M. Carroll. The blind men and the elephant: Views of scenario-based system design. *Interactions*, 11(6):44–53, November-December 2004.

[Hoa78] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 21(8):666–677, 1978.

[Hoa83] C. A. R. Hoare. Communicating sequential processes. *Commun. ACM*, 26(1):100–106, 1983.

[HRR05a] Michael G. Hinchey, James L. Rash, and Christopher A. Rouff. Enabling requirements-based programming for highly-dependable complex parallel and distributed systems. In *Proceedings of the 11th International Conference on Parallel and Distributed Systems - Workshops (ICPADS'05)*, pages 570–574, Washington, DC, USA, 2005. IEEE Computer Society.

[HRR05b] Michael G. Hinchey, James L. Rash, and Christopher A. Rouff. A formal approach to requirements-based programming. In *Proceedings of 12th Annual IEEE International Conference and Workshop on the Engineering of Computer Based Systems (ECBS 2005)*, pages 339–345, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[HRR05c] Michael G. Hinchey, James L. Rash, and Christopher A. Rouff. Requirements to Design to Code: Towards a fully formal approach to automatic code generation. Technical Report 2005-212774, National Aeronautics and Space Administration, July 2005.

[KA06] Jason Kealey and Daniel Amyot. Towards the automated conversion of natural-language use cases to graphical use case maps. In *Proceedings of CCECE/CCGEI 2006*, pages 2342–2345. IEEE Canada, 2006.

[KK92] Clare-Marie Karat and John Karat. Some dialog on scenarios. *SIGCHI Bull.*, 24(4):7, 1992.

[KKAM06] Jason Kealey, Yongdae Kim, Daniel Amyot, and Gunter Mussbacher. Integrating an Eclipse-based scenario modeling environment with a requirements management system. In *Proceedings of CCECE/CCGEI 2006*, pages 2397–2400. IEEE Canada, 2006.

[KM07] Matt Kaufmann and J. Strother Moore. ACL2 webpage [online]. April 2007 [cited May 3, 2007]. Available from: http://www.cs.utexas.edu/users/moore/acl2/. University of Texas at Austin.

[KS98] Gerald Kotonya and Ian Sommerville. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Baffins Lane, Chichester, West Sussex, England, 1998.

[Mai98] N. A. M. Maiden. CREWS-SAVRE: Scenarios for acquiring and validating requirements. *Automated Software Engg.*, 5(4):419–446, 1998.

[McG97] Karen L. McGraw. *User-Centered Requirements: The Scenario-Based Engineering Process*. Lawrence Erlbaum Associates, 10 Industrial Avenue, Mahwah, NJ, USA, 1997.

[Nar92] Bonnie A. Nardi. The use of scenarios in design. *SIGCHI Bull.*, 24(4):13–14, 1992.

[RA97] C. Rolland and C. Ben Achour. Guiding the construction of textual use case specifications. *Data & Knowledge Engineering Journal*, 25(1):125–160, March 1997.

[RAC+98] C. Rolland, C. Ben Achour, C. Cauvet, J. Ralyté, A. Sutcliffe, N. Maiden, M. Jarke, P. Haumer, K. Pohl, E. Dubois, and P. Heymans. A proposal for a scenario classification framework. *Requirements Engineering*, 3(1):23–47, 1998.

[Sch00] Steve Schneider. *Concurrent and Real-time Systems: The CSP Approach*. John Wiley & Sons, Baffins Lane, Chichester, West Sussex England, 2000.

[SDV96] Stéphane Somé, Rachida Dssouli, and Jean Vaucher. Toward an automation of requirements engineering using scenarios. *Journal of Computing and Information*, pages 1110–1132, 1996. Special issue: ICCI'96, 8th International Conference of Computing and Information. Available from: citeseer.ist.psu.edu/218135.html.

[WPJH98] Klaus Weidenhaupt, Klaus Pohl, Matthias Jarke, and Peter Haumer. Scenarios in system development: Current practice. *IEEE Softw.*, 15(2):34–45, 1998.

[Wri92] Peter Wright. What's in a scenario? *SIGCHI Bull.*, 24(4):11–12, 1992.

[YB87] Richard M. Young and Phil Barnard. The use of scenarios in human-computer interaction research: Turbocharging the tortoise of cumulative science. In *Proceedings of the SIGCHI/GI conference on Human factors in computing systems and graphics interface (CHI '87)*, pages 291–296, New York, NY, USA, 1987. ACM Press.

[YB92] Richard M. Young and Philip J. Barnard. Multiple uses of scenarios: A reply to Campbell. *SIGCHI Bull.*, 24(4):10, 1992.

REPORT DOCUMENTATION PAGE

Form Approved
OMB No. 0704-0188

1. REPORT DATE *(DD-MM-YYYY)*	2. REPORT TYPE	3. DATES COVERED *(From - To)*
10-09-2007	Technical Memorandum	

4. TITLE AND SUBTITLE

Mise en Scene: Conversion of Scenarios to CSP Traces for the Requirements-to-Design-to-Code Project

5a. CONTRACT NUMBER

5b. GRANT NUMBER

5c. PROGRAM ELEMENT NUMBER

6. AUTHOR(S)

John D. Carter, William B. Gardner, James L. Rash, and Michael G. Hinchey

5d. PROJECT NUMBER

5e. TASK NUMBER

5f. WORK UNIT NUMBER

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Goddard Space Flight Center
Greenbelt, MD 20771

8. PERFORMING ORGANIZATION REPORT NUMBER

2007-02241

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

National Aeronautics and Space Administration
Washington, DC 20546-0001

10. SPONSORING/MONITOR'S ACRONYM(S)

11. SPONSORING/MONITORING REPORT NUMBER

TM-2007-214155

12. DISTRIBUTION/AVAILABILITY STATEMENT

Unclassified-Unlimited, Subject Category: 62
Report available from the NASA Center for Aerospace Information, 7115 Standard Drive, Hanover, MD 21076. (301)621-0390

13. SUPPLEMENTARY NOTES

John D. Carter and William B. Gardner: University of Guelph, Ontario, Canada

14. ABSTRACT

The "Requirements-to-Design-to-Code" (R2D2C) project at NASA's Goddard Space Flight Center is based on deriving a formal specification expressed in Communicating Sequential Processes (CSP) notation from system requirements supplied in the form of CSP traces. The traces, in turn, are to be extracted from scenarios, a user-friendly medium often used to describe the required behavior of computer systems under development. This work, called Mise en Scene, defines a new scenario medium (Scenario Notation Language, SNL) suitable for control-dominated systems, coupled with a two-stage process for automatic translation of scenarios to a new trace medium (Trace Notation Language, TNL) that encompasses CSP traces. Mise en Scene is offered as an initial solution to the problem of the scenarios-to-traces "D2" phase of R2D2C. A survey of the "scenario" concept and some case studies are also provided.

15. SUBJECT TERMS

Requirements Based Programming, Formal Methods, Formal Specification

16. SECURITY CLASSIFICATION OF:			17. LIMITATION OF ABSTRACT	18. NUMBER OF PAGES	19b. NAME OF RESPONSIBLE PERSON
a. REPORT	b. ABSTRACT	c. THIS PAGE			James L. Rash
Unclassified	Unclassified	Unclassified	Unclassified	19	19b. TELEPHONE NUMBER *(Include area code)* (301) 286-5246