Mise en Scene: Converting Scenarios to CSP Traces in Support of Requirements-Based Programming

J. Carter, W.B. Gardner
Department of Computing and Information Science, University of Guelph, ON, Canada
{jcarter,gardnerw}@uoguelph.ca

Abstract

The "Requirements to Design to Code" (R2D2C) project of NASA's Software Engineering Laboratory is based on inferring a formal specification, currently using Communicating Sequential Processes (CSP), from system requirements supplied in the form of scenarios, a user-friendly medium often used to describe the behavior of computer systems under development. The scenarios are first converted into an intermediate form, CSP traces, from which are derived CSP specifications. 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) which encompasses CSP traces. A survey of the "scenario" concept and a small case study are also presented.

1. Introduction

An approach to requirements-based programming called R2D2C, that utilizes a formal method internally, has been described in previous literature [1, 2]. The approach takes as its primary input "requirements as scenarios," as shown in the first phase of its development ("D") flow, D1 Scenarios Capture, illustrated in Figure 1. The subsequent phases, D2 Traces Generation and D3 Model Inference, convert the requirements into a formal model in algebraic CSP notation (Communicating Sequential Processes) [3, 4], which, in turn, is subjected to various kinds of analysis and verification in phase D4 Model Analysis, and used by D5 Code Generation to synthesize an implementation.

While the R2D2C approach is targeted at systems whose requirements can be expressed as "scenarios," that term was not defined. The term is common in software engineering, but somewhat ambiguous. In this context, whatever definition of scenario is intended or chosen must also be amenable to conversion into CSP traces, which have their own special properties.

This work provides a way to fill in the D2 block by answering two questions: (1) What shall be the working definition of "scenario" for R2D2C? and (2) How shall scenarios be converted to CSP traces? The paper proceeds by giving a more detailed problem statement that highlights specific challenges, and then surveys the literature on "scenarios" and "scenario-based approaches" for guidance before settling on a working definition. Next, our D2 candidate tool called **Mise en Scene** is described, which supports authoring of scenarios and automatic conversion to CSP traces. This is illustrated with a case study. Finally, future work in the context of R2D2C is proposed.

2. Problem statement

First, we give preliminary definitions of the D2 phase's input and output, respectively:

- A *scenario* is a sequence of steps that a system is required to carry out. (References and discussion are given below.)

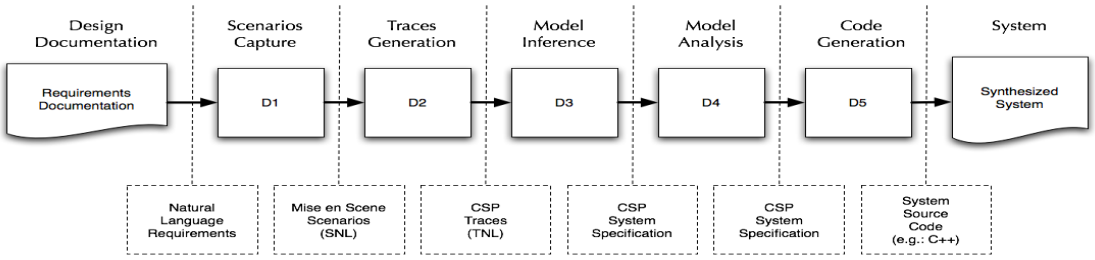

Figure 1. Phases of R2D2C development process

- A *trace* is a sequence of events that a CSP system specification executes.

In CSP, traces are represented as a list of events separated by commas, enclosed within a set of angle brackets (< >). A single trace, say <a, b, c>, which means the sequence of three named events, defines *one* permitted execution of the specification; *all* permitted executions are given as a set of traces. The set is represented as one or more comma-separated traces, enclosed within braces, e.g., { <>, <a>, <a,b>, <a,b,c> }. If a specification is viewed as a state machine, its set of traces embodies 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, while CSP process specifications may include channel I/O events such as `sensor?degrees` (meaning, read the sensor channel into a degrees variable) and `sensor!32` (meaning, output 32 on the sensor channel), the trace event that records the sensor communication event where 32 is stored into degrees would be simply *channel.data:* `sensor.32`. That is, the direction of communication is not recorded. 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 level of traces; all one sees is the record of executed events. Stated another way, process specifications may provide clues about a possible implementation's architecture, but traces contain no such clues. For R2D2C's "D" 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 intentionally kept "under the hood." 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, we state our assumptions about the D2 phase's context: The D1 phase, Scenarios Capture, is likely to be application specific. We only assume that it will be capable of outputting "scenarios" using the syntax we specify. For prototyping work, it may be sufficient to represent D1 as a simple text editor.

As for D3 Model Inference, this is the heart of the theoretical work, which 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 [5]. We assume that D3 will accept input in the form of conventional CSP trace notation, but we also anticipate that there will be room for negotiation with an eventual D3 implementation 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 normally be thrown away during their conversion to traces. Therefore, we are not too concerned to produce a definitive and final form of D2 output at this time, since we expect that adjustments will be necessary as the requirements of D3 processing are firmed up.

The essential problem of designing the D2 phase— converting scenarios to traces—is comparable to digging a tunnel from both ends and arranging to meet in the middle. 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. As it is useless to prescribe scenario constructs that cannot be converted to traces, we have focused solely on constructs that have a conceptual analog in trace notation**.** We have adopted a subset of those constructs commonly appearing in scenario-based approaches, described next.

3. Related work on scenarios

In software and requirements engineering disciplines, as well as human-computer interaction (HCI), the term "scenario" is used frequently, in a variety of contexts. Its exact meaning has been the focus of many debates [6, 7, 8, 9, 10] and usage surveys [11, 12, 13, 14, 15, 16]. Most authors 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 commonly employed without any preamble specifying its meaning. It is fair to call the term "vague in definition and scope" [15].

Allenby and Kelly [17], for example, define scenarios as a "sequence of actions used to illustrate system behavior." The CREWS [18] survey of fifteen "current practices" of European companies using scenarios for system engineering found that many approaches to scenarios were based on UML use cases,

with added custom extensions to satisfy their needs. That survey identifies four criteria for characterizing the usage of scenarios:

- *Purpose*—Describes how the scenario is being used, and in what context.
- *Contents*—The knowledge of the problem domain as contained in the scenario.
- *Form*—The method or medium by which the scenario is expressed.
- *Lifecycle*—How scenarios change and evolve through the development process.

The CREWS project resulted in the creation of a scenario authoring tool "L'ECRITOIRE" [19].

Aside from their role in requirements gathering, scenarios are useful in producing test cases [20]. Similarly, anti-scenarios (and misuse cases) can be created to describe ways in which malicious actors strive to undermine the system, and demonstrate that the system is resilient to such attacks [21].

Notable drawbacks of scenarios have been cited. Scenarios are episodic in nature [22], and rely on the existence of well-defined start and end points, with their associated conditions. Many systems, particularly real-time systems, once started, ideally never terminate. This makes scenarios impractical for describing such systems, or requires careful thought by scenario authors. In the context of CSP traces, this lack of termination gives rise to infinite traces. In his work, Harel [23, 24] asserts that scenarios are more applicable to reactive systems (often referred to as "control dominated systems"), than to systems designed to handle complex calculations and data processing ("data dominated systems"). Scenarios are seen as more appropriate for describing interactions than for calculations. Scenarios may also be impractical for expressing non-functional requirements, and can become unwieldy as system size grows.

Scenarios (as variously defined) have been incorporated in software engineering approaches, such as Harel's "Play" [23], where scenarios are recorded as Live Sequence Charts (similar to Message Sequence Charts such as used by MESA, Message Sequence Charts Editor, Simulator, Analyzer [25]). The CREWS project created the SAVRE tool, Scenarios for Acquiring and Validating Requirements [12, 26], for generating scenarios from previously authored use cases. SAVRE represents a software-based approach that bridges automated system analysis techniques and human-based activities like stakeholder meeting/consultation sessions, with the purpose of identifying deficiencies or omissions in the proposed system.

Other scenario-based approaches include SCENario Environment (SCENE) [27], a software development tool for creating scenario diagrams from object-oriented source code, and Scenario Plus [28], a loosely-defined methodology providing a set of forms for eliciting scenarios from stakeholders, and guidelines for using such forms. The Scenario Plus use case is built upon the foundation laid by Cockburn [29]. Cockburn's use case comprises two types of scenarios, success and failure scenarios (occasionally referred to as "positive" or "negative" scenarios). A success scenario is a sequence that results in satisfactory behavior, while a failure scenario is used to demonstrate that the system does not fail in a risky or hazardous manner. It is important to note that Cockburn's success scenarios do not describe "required" behavior; they describe *an instance* where the desired (or safe) behavior was achieved. This is different from their proposed use in R2D2C, where scenarios do describe the required system behavior.

After studying the above definitions and uses of "scenarios," a set of attributes common to the majority of the approaches surveyed was identified. The result, listed in Table 1, is not dissimilar from Cockburn's description of scenario [29]. Attributes that are present in our approach (defined below) are marked with *, while attributes that are not explicitly used but partially present in our approach are marked with **.

Table 1. Common scenario attributes

Common Name of Attribute	Description
Name*	Title used to identify the scenario.
Description*	A textual description of the actions contained in, and the actors involved in, the scenario.
Author*	Who created the scenario.
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.
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.

Table 1. Common scenario attributes (Cont.)

Common Name of Attribute	Description
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.
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.
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.
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.
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.
Non-Functional Requirements**	Textual documentation containing requirements information that cannot be readily expressed in a functional form.
Minimal Guarantee	Conditions that are guaranteed to be true, in even the most disastrous failure.
Success Guarantee	Conditions that are true upon successful termination of a scenario.

Utilizing the selection of attributes above qualifies our approach as recognizably "scenario based." Still required was to customize our scenarios to suit translation to CSP traces, and provide a mechanical way of doing the translation. The full approach is described in the next section.

4. Mise en Scene

The name comes from the field of film studies, where it is regarded as a frequently overused term that is infrequently defined. Literally translating *mise-en-scène* from French gives "putting in the scene." It is often used to describe everything visible to the camera within a shot, especially elements relevant to the narrative of the work. Mise en Scene was chosen as a title partly as a nod to ambiguity in terminology, noting that the term "scenario" is equally ambiguous and prone to many interpretations.

Mise en Scene has four components: (1) the scenario medium, Scenario Notation Language (SNL); (2) a means for connecting scenarios, Scenario Glossary (SNLGlue); (3) the mechanical transformation process, SCN2T ("seen-2-it"); and (4) the output trace notation (TNL). Each is described in the following subsections.

4.1. Representing scenarios

The medium created for Mise en Scene is a textual, form-based notation loosely based on Cockburn's guidelines for effective use cases [29]. Whereas Cockburn states that a use case collects a number of scenarios describing related behavior, Mise en Scene does not follow his conceptual grouping. An SNL scenario is used to describe a single pattern of required execution in the proposed system from the standpoint of an actor.

The description below uses three special terms: **Component** is currently synonymous with actor, but is reserved for potential generalization to the notion that other entities in addition to actors might be involved in the scenario. The smallest unit of execution that components carry out is called a **task**, which is intended to correspond to a CSP event. The **channel** is imported directly from its CSP sense for unidirectional, nonbuffered communication between a "producer" compo-

nent and a "consumer" component. SNL has statements for defining the *schemas* of components and tasks by unique ID and textual description. A task schema may optionally list the components that are allowed to perform it. A channel schema lists the data types that it communicates, and its related producer and consumer components; its ID is 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.

Required fields in a scenario schema are in bold, the others being optional:

- **ID**—A name that uniquely identifies the scenario.
- 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.
- **Primary Actor**—The identifier of the component that carries out all actions in this scenario. A scenario may only have a single component as its primary actor, which is considered to be the initiator of the scenario.
- Secondary Actors—Identifiers of components referenced within the scenario, with which the primary actor communicates and synchronizes.
- **Scenario Flow**—Cockburn refers to this as the "main success scenario," or the case in which "nothing goes wrong" [29]. The flow of a scenario is an ordered set of steps expressed in a restricted syntax that specifies the behavior of the scenario as actions undertaken by the primary actor, and as communication and synchronization with the secondary actors. 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 medium, SFT, is described in Section 4.2.
- **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, making the scenario always ready to be triggered (see next).
- **Trigger**—The trigger of a scenario is 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 present) provide a guard. The most permissive trigger is the task ID System::start, which enables a scenario to be executed upon system startup. Once a scenario completes, it must be retriggered in order to execute again.
- Extensions—Extensions of a scenario are 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 `invokes` 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.

4.2. Scenario Flow Text (SFT)

Scenario Flow Text, or SFT, is a number 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 [29, 30, 31], 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 2.

Variables are used for channel operations sending or receiving data, and are necessary for arithmetic and conditional syntax. SFT provides two levels of scope, system and scenario levels, where variables are defined in `Preamble` fields. Some can be set to constant values to provide system-wide configuration information. SFT supports the following built-in variable types:

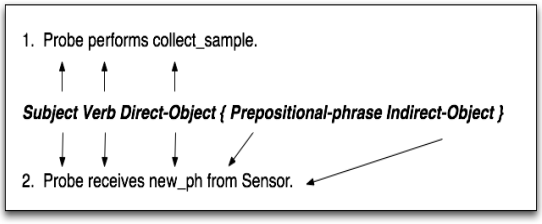

Figure 2. SFT syntax with SVDPI pattern

integer, character, float, string, bit, boolean. Additionally, SFT provides the ability to add custom variable types to a specification. Type information is contained in the outgoing trace representation, discussed next.

4.3. Representing traces

The common form of traces as used with the Formal Systems CSP tools [32] is also utilized in Mise en Scene, with the addition of an XML wrapper that associates a set of traces with the scenarios from which they were derived and enables additional information to be passed to the next stage 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 scenarios from which the set of traces was derived.
- **Actor Identifier**—The identifiers of the system components 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 trace output, we distinguish *terminal traces,* those that contain a "full thread" of events recording a system execution from start to finish, from non-terminal traces, those prefixes of a terminal trace that record any execution short of the end. (Infinite traces 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 < > which 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 D2. Therefore, the SCN2T conversion process generates only the terminal traces from a system represented as a collection of scenarios. Non-terminal traces can be easily derived by the D3 phase, should it require them.

TNL diverges from traditional CSP traces in one other manner: variables. Channel events in traces would normally contain literal data values, not variables. But this can result in a state space explosion, as all combinations of valid data for variables must be generated in a process's traces. To simplify the TNL medium, variable placeholders have been introduced, thus deferring the complete variable expansion to the D3 phase, if required at all. These placeholders have types and value ranges associated with them.

Since conditional expressions and arithmetic operations cannot be directly expressed in CSP trace notation, our approach introduces a method for encoding this information as trace events. Without the ability to pass this information to D3 and later stages, calculations contained in the scenario would be lost and not able to be synthesized into an executable implementation.

4.4. Process level and system traces

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

In the scenario-based approaches surveyed, a major difficulty was connecting, collecting, and categorizing scenarios. To assist a scenario's authors, we envision a scenario editor with automated actor 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 component that unites the other three (SNL, TNL, SCN2T). SNLGlue is a software tool, rather than another language, likely to be added during integration with R2D2C.

4.5. Conversion algorithm

The cornerstone of Mise en Scene is the process by which scenarios represented in SNL are converted to TNL. This section starts by describing the rules that are used to convert from a single scenario's SFT to TNL, and then explains how to generate traces from scenarios in combination.

Table 2 lists the main elements of SFT syntax by step type, with examples, and describes the corresponding trace output.

After the conversion of individual scenarios, and appending sequentially composed scenarios (i.e., where one scenario's termination triggers another scenario), comes the task of combining scenarios in parallel. Concurrency is inferred from rendezvous and communication steps. The trigger attribute also determines how scenarios combine: If two scenarios share a trigger, they are eligible to be placed in parallel.

Table 2. Translating steps of SFT

Step Type	Step Example and Trace Output
Executional	Robot performs systemCheck. States that a system component performs an action, thus a single event is appended to the trace output.
Communication	Robot receives loc from PositioningSat. 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	if (ph > 4) then Robot performs report_sample. else Robot performs discard_sample. 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	Robot invokes Init::Extension::HandleFailedStartup. The traces for an extension are calculated according to the steps they contain, and are appended to the outputted set of traces.
Arithmetic	Robot performs { x = x + 2 }. Creates an event that denotes the arithmetic operation, and the values (variables or constants) contained in the operation.
Rendezvous	Robot performs synchronizeClocks with PositioningSat. At the scenario level, rendezvous syntax has no additional effect on the trace output.

Preconditions play an important role in the creation of system traces. Preconditions filter the set of terminal traces to remove all those that do not contain the events specified by the precondition.

The generation of system traces is carried out using an eight-step algorithm for determining when scenarios are eligible to execute, and calculating all the possible ways that events can interleave in the set of system traces. This algorithm is repeated until the set of system traces becomes stable, i.e., all terminal traces have been calculated. See [33] for details.

5. Case studies

As a means of confirming the expressive capabilities of Mise en Scene's scenario medium, we set out to rewrite existing R2D2C examples in SNL. Below is the "Page Analyst" scenario from the LOGOS/ANTS system [34], reworked here using 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 Actor: PagerAgent
Secondary Actors: 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 specifies the tasks carried out by the PagerAgent to retrieve an analyst's contact information and page the analyst. The scenario is triggered by a request from the UIAgent.

The three-event terminal trace generated from this scenario by SCN2T matches the scenario's three steps:

```
<PagerAgent_DatabaseAgent_PAGERINFOTYPE_QU
ERY.requestinfo,
DatabaseAgent_PagerAgent_ANALYSTINFOTYPE_R
ESULT.analystinfo, createandStoreMessage>
```

Since the examples contained in existing R2D2C publications were brief and did not exercise the richness of SNL's syntactic constructs, we constructed a somewhat larger control-dominated system. Below is an excerpt from a remotely-controlled robot probe system, given in full in [33]. 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 Actor: Probe
Secondary Actors: 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.
```

The five traces generated from this scenario by SCN2T are listed in Figure 3. The complete robot probe contains an additional five scenarios: RobotStart, Sensor_Collect, Move_Probe, LeftTread_Movement, and RightTread_Movement [33].

6. Future work

The largest open problem with respect to SCN2T

```
<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, Sensor_Probe_WATERSAMPLE.new_water,
Probe_Station_PHSAMPLE.new_ph, Probe_Station_WATERSAMPLE.new_water, sample_done,
acknowledged>
```

Figure 3. Traces of RobotCommand scenario

is the area 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 are 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 starting point for developing such a system.

Another possible area of application for Mise en Scene is the generation of traces for use in formal verification. To check 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 System's 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.

7. Conclusion

The process of scenario-to-trace conversion is composed of two sub-problems. The first is converting individual scenarios into sets of equivalent traces. This was handled by a process of mapping each line in a scenario to one or more events in the generated traces. The second, and larger, problem is combining the set of individual component traces into a set of system traces. This is more difficult than single scenario conversion due to the need to satisfy constraints of multiple scenarios, the resulting large data sets, and the likelihood of combinatorial explosion of traces.

A challenge of all scenario-based approaches is managing ambiguity introduced by scenario-based techniques. In specifying a system as a set of loosely connected scenarios, it is difficult to compose them into a larger whole, and with Mise en Scene we have achieved this through the communication and synchronization paradigm of CSP, and by limiting the specification medium from natural language to a structured text representation.

Scenario-based approaches are not the best fit for every kind of possible system. However, within the planned scope of R2D2C, Mise en Scene provides a working definition of scenarios and a path to go forward into the phase of formal model extraction from CSP traces.

Acknowledgements

This work is supported by grants from Canada's Natural Science and Engineering Research Council.

References

[1] 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.

[2] James L. Rash, Michael G. Hinchey, Christopher A. Rouff, Denis Gracanin, and John Erickson. Experiences with a requirements-based programming approach to the development of a NASA autonomous ground control system. In *Proceedings of Engineering of Computer Based Systems (ECBS '05)*, pages 490–497, Los Alamitos, CA, USA, 2005. IEEE Computer Society.

[3] C.A.R. Hoare. *Communicating Sequential Processes*. Prentice Hall International, revised edition, July 2004.

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

[5] Matt Kaufmann and J Strother Moore. ACL2 homepage [online]. November 2006 [cited December 8, 2006]. Available from: http://www.cs.utexas.edu/users/moore/acl2/.

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

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

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

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

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

[11] 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.

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

[13] 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.

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

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

[16] 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.

[17] Karen Allenby and Tim Kelly. Deriving safety requirements using scenarios. In *Proceedings of the Fifth IEEE International Symposium on Requirements Engineering (RE '01)*, page 228, Washington, DC, USA, 2001. IEEE Computer Society.

[18] Camille Ben Achour. CREWS webpage [online]. February 2000 [cited June 1, 2006]. Available from: http://crinfo.univ-paris1.fr/CREWS/.

[19] M. Tawbi, C. Souveyet, and C. Rolland. L'ECRITOIRE: A tool to support a goal-scenario based approach to requirements engineering. Internal Report CREWS No. 21.903, ESPRIT project, 1998. Available from: ftp://sunsite.informatik.rwth-aachen.de/pub/CREWS/CREWS-98-23.pdf.

[20] Ian Alexander. *Scenarios, Stories, Use Cases*, chapter 14. Use Cases, Test Cases, pages 281–298. John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, England, 1st edition, 2004.

[21] Ian Alexander. *Scenarios, Stories, Use Cases*, chapter 7. Negative Scenarios and Misuse Cases, pages 119–139. John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, England, 1st edition, 2004.

[22] Matthias Jarke. Scenarios for modeling. *Communications of the ACM*, 42(1):47–48, 1999.

[23] David Harel and Rami Marelly. *Come, Let's Play: Scenario-Based Programming Using LSC's and the Play-Engine*. Springer-Verlag New York, Inc., Secaucus, NJ, 2003.

[24] David Harel. From play-in scenarios to code: An achievable dream. *Computer*, 34(1):53–60, 2001.

[25] Hanene Ben-Abdallah and Stefan Leue. MESA: Support for scenario-based design of concurrent systems. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 118–135, 1998. Available from: http://citeseer.ist.psu.edu/article/ben-abdallah97mesa.html.

[26] Perminder Sahota. *Scenarios, Stories, Use Cases*, chapter 18. Scenarios in Air Traffic Control (ATC), pages 363–377. John Wiley & Sons, Ltd, The Atrium, Southern Gate, Chichester, West Sussex, England, 1st edition, 2004.

[27] Kai Koskimies and Hanspeter Mössenböck. Scene: Using scenario diagrams and active text for illustrating object-oriented programs. In *Proceedings of the 18th international conference on Software engineering (ICSE '96)*, pages 366–375, Washington, DC, USA, 1996. IEEE Computer Society.

[28] Ian Alexander. Scenario plus website [online, cited June 21, 2006]. Available from: http://www.scenarioplus.org.uk/.

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

[30] 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.

[31] 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.

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

[33] J. Carter, W.B. Gardner, J.L. Rash, and M.G. Hinchey. Mise en Scene: Scenario to CSP trace conversion for the Requirements to Design to Code project. Technical Report (no. TBD), National Aeronautics and Space Administration, 2007.

[34] 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.