Kenneth J. Turner. Formalising Graphical Behaviour Descriptions, Proc. 10th. International Conference on Algebraic Methodology and Software Technology, 537-552, Copyright Elsevier Science Publishers, Amsterdam, March 2004.

Formalising Graphical Behaviour Descriptions

Kenneth J. Turner

Computing Science and Mathematics, University of Stirling, Stirling FK9 4LA, UK

Abstract. CRESS (Chisel Representation Employing Systematic Specification) is used for graphical behaviour description, underpinned by formal and implementation languages. Plug-in frameworks adapt it for particular application domains such as Intelligent Networks, Internet Telephony and Interactive Voice Response. The CRESS notation and its syntax are explained. The semantics of CRESS is discussed with reference to its interpretation in LOTOS.

Keywords: Graphical Specification, LOTOS (Language Of Temporal Ordering Specification), SDL (Specification and Description Language), Voice Service

1 Introduction

1.1 Background

Diagrammatic representations abound in science and engineering. For example, software engineering uses flowcharts, entity-relationship diagrams, data-flow diagrams, state diagrams, and various UML diagrams. In general, diagrams are valued because they give a clear overview of a system. In industry, graphical representations are regarded as more accessible than textual ones (especially for more formal specifications).

This paper describes the basis of CRESS (Chisel Representation Employing Systematic Specification). Although CRESS was inspired by the need to represent voice services, it is a general-purpose way to represent behaviour graphically. CRESS can therefore be used for a variety of other applications. Unlike many diagrammatic forms, CRESS is precise in that diagrams are interpreted by an underlying formal model. CRESS diagrams can also be translated to implementation languages. CRESS supports:

- graphical behaviour description, open to non-specialists and industrial engineers
- a precise interpretation that allows rigorous analysis and development
- a portable toolset that facilitates specification, implementation, analysis and testing.

The same service diagrams can be used for multiple purposes. CRESS is neutral with respect to the target language. For formal analysis, CRESS diagrams are automatically translated to LOTOS (Language Of Temporal Ordering Specification [4]) and to SDL (Specification and Description Language [6]). For implementation, CRESS diagrams are automatically translated to Perl (for Internet Telephony services) or to VoiceXML (for Interactive Voice Response services).

CRESS was initially based on the industrial notation Chisel developed by BellCore [1]. However, CRESS has been considerably extended since its beginnings. For example, it now supports the notion of plug-in domains: the vocabulary and concepts required for each application area are defined separately. CRESS has been used in the domains of Intelligent Networks, Internet Telephony and Interactive Voice Response.

Although other papers by the author have described the *applications* of CRESS, the present paper considers the *foundations* of CRESS, namely the composition, syntax and semantics of CRESS diagrams. The CRESS philosophy focuses on two aspects:

– The graphical notation is of most interest and value to domain experts such as communications engineers. Such users require a convenient and pragmatic way of describing services.
– The rigorous analysis of formal specifications derived from diagrams is of most interest and value to formalists. Such users require precise expression with the ability to reason formally about the specifications.

The translation from diagrams to specifications is of limited interest to both categories of user. Although the paper gives some indication of how translation is achieved, it is a secondary issue. In particular, the translation procedure is not formalised though the results of the translation are formal. The important point is that specifications generated from CRESS are precise and can be reasoned about.

1.2 Relationship to Other Work

Diagrammatic notations, e.g. visual programming languages, are common in software engineering. However few graphical approaches have a formal basis. Statecharts [3], LSCs (Live Sequence Charts [7]), and UML all have graphical representations with a formal basis. The following techniques are perhaps closest to CRESS:

– SDL has a graphical syntax and a formal interpretation. However SDL lacks composition mechanisms that are appropriate for building services. It is a specialised notation that needs expert knowledge. SDL also does not have support for specific application domains.
– MSCs (Message Sequence Charts [5]) are higher-level and more straightforward. Several authors have given formal meaning to MSCs. Like SDL, MSCs are also rather generic and lack composition mechanisms suitable for services.
– UCMs (Use Case Maps [2]) are useful for high-level descriptions of requirements. UCMs have been represented using LOTOS and MSCs. However the formalisation of UCMs is not complete, and they lack support for specific application domains.

CRESS aims to circumvent these problems:

– CRESS accepts plug-in domains that allow it to be readily adapted for use in a variety of applications.
– CRESS supports the needs of voice services, though it is more widely applicable. Specific mechanisms are provided in CRESS for service composition.
– CRESS is intended as a front-end for formal representations. That is, CRESS is translated into a formal language into order to give it precise meaning. The implied semantic model of CRESS is reasonably straightforward, so there can be confidence in the equivalence of different formal models.

Formally-based language translation has been studied for many decades. [9] is an example of the state-of-the-art. Although such approaches might in principle be used for CRESS, they would be a side interest. In addition, the scale and challenges of translating CRESS to very different target languages cast doubt on the practicability of formally-based translation. CRESS includes concurrent, nondeterministic and event-driven behaviour. CRESS descriptions may be cyclic and may contain context-sensitive expressions. All these aspects would be challenging for a formal approach to translator design.

Even if the problems could be overcome, proving the translation correct would be very difficult and time-consuming. The outcome would also be of little benefit for the intended purposes of CRESS (graphical description and formal analysis).

1.3 Overview of The Paper

As motivation for CRESS, section 2 briefly overviews the role of CRESS and how it has been applied. Section 3 gives a condensed summary of the CRESS notation. An overview is given in section 4 of the syntax and static semantics of diagrams. Section 5 deals with the semantics of diagrams by considering their interpretation in LOTOS.

2 CRESS Application

2.1 The Role of CRESS

Why not define system behaviour using some implementation technique used for program development? Such a definition would naturally be rather low-level, language-dependent, and possibly platform-dependent. It would be hard to analyse the definition rigorously. Historically, this is how communications services were defined. The approach suffers from major problems such as incompatibility among vendors, inconsistency among features, lack of portability, and cost of testing.

A formally-based approach is therefore an obvious choice. Why not then just specify behaviour using a selected formal method? There are a number of problems that CRESS aims to circumvent:

Acceptability: Formal methods have achieved only limited penetration into industry. In general, engineers are not trained in formal methods and are hesitant to use them. CRESS exploits the benefits of formal methods 'behind the scenes' without forcing them on the user. As a graphical notation, CRESS is more accessible to the non-specialist. Indeed, CRESS benefits from its origins in Chisel as a notation that can be used by all stakeholders in system development.

Architecture: CRESS provides a framework and vocabulary for specifying applications in various domains. CRESS is therefore close to the architectural level at which a domain specialist would normally think. If a plain formal language is used, it is necessary to describe behaviour in terms of *language* concepts rather than in terms of *architectural* concepts. This leads to specifications that are low-level (in architectural terms) and verbose.

Neutrality: CRESS is not oriented towards any particular domain, target language or platform. It can therefore act as a front-end for creating formal specifications. CRESS diagrams are currently translated into LOTOS and SDL, and could be translated into many constructive formal languages.

Implementation: Formal specifications are usually rather distant from implementation. As a result, the effort put into specification is often not exploited when implementation takes place – there is a discontinuity in development that risks introducing inconsistencies. CRESS is unusual in that the *same* diagrams can be translated into both formal specifications and into implementations (in selected languages).

So, when is CRESS applicable? In general it can be used to give a constructive description of a system that has inputs, outputs, actions and events. That is, CRESS is useful for reactive systems. Although CRESS can be used to describe the behaviour of an entire system, its specialised capabilities come into their own when the system can be considered as a base behaviour (service) modified by optional additional capabilities (features). Such a situation is common in voice applications such as telephony, but is also common in many other cases. For example, software applications frequently have plug-in modules that are used to extend their abilities. CRESS is therefore most appropriate for modular systems.

CRESS cannot be used for non-constructive (e.g. declarative, logical) descriptions. Although CRESS descriptions are hierarchical in the sense of diagrams invoking other diagrams, CRESS is not able to describe a system at multiple levels of abstraction.

CRESS scales reasonably well. Since systems are usually described by multiple diagrams, a complex system can be handled as long as its behaviour can be broken down into manageable diagrams. When used for verification (proof), CRESS has the same characteristics as the formal language in which CRESS is interpreted. In practice, this usually places severe limits on what can be verified. When used for validation (testing), CRESS is much more practical. As long as tests can be defined fairly independently, a complex system can be validated without much regard to scale.

2.2 Domain Frameworks

In themselves, CRESS diagrams have no meaning. In fact, CRESS is deliberately open-ended as far as applications are concerned. Although CRESS defines a structure for diagrams, diagram content is governed by plug-in domains that define the syntax and semantics of diagram contents. As will be seen, CRESS has been used in three domains with four target languages.

Some aspects are domain-specific but language-independent. For example, a domain defines the names and parameter types of input signals, output signals, actions and event conditions. The sources and destinations are given for signals. To resolve certain kinds of composition problems, diagrams may be given priorities that control the order in which they are applied.

Some aspects are both domain-specific and language-specific. A specification architecture gives an outline specification in the chosen target language. This identifies the communicating subsystems and the domain-independent data types. When CRESS diagrams in this domain are translated to this language, the generated code is embedded in the specification architecture. Most of the generated code deals with behaviour, but some of it defines domain-specific data types.

2.3 Tool Support

The CRESS toolset is written largely in Perl for portability. About 14,000 lines of code in numerous modules are used to support seven main tools. The major CRESS tool resembles a conventional compiler. However CRESS is unusual in interpreting a graphical description. In addition, the possibly cyclic nature of graphs requires special treatment.

The CRESS lexer reads each diagram and turns it into a directed graph. The CRESS parser combines all the graphs into one, and checks the syntax and static semantics of the resulting graph. A CRESS code generator for each target language translates the graph into this language. These tools are invoked by a preprocessor that is used with a specification architecture. This determines the configuration and deployment of diagrams, and embeds their translation into the specification framework.

When CRESS is interpreted in a formal language, verification and validation are obvious choices. [15] describes how these can be performed. In general, CRESS is open to any verification technique that would normally be used with the formal language. However, CRESS is accompanied by its culinary counterpart MUSTARD (Multiple-Use Scenario Test And Refusal Description) as a practical means of validating behaviour. The MUSTARD tool translates validation scenarios into the target language and automatically checks them against the system specification. This is used to establish confidence in the CRESS description, and also to check for incompatibilities among features.

As far as possible, the toolset is automated so that the user is unaware of it. For example, a LOTOS user issues a TOPO toolset command to process a given specification. This invokes the CRESS toolset, creates a new specification from the CRESS diagrams, and uses MUSTARD to validate it. Similarly, an SDL user clicks on a button in the Telelogic TAU toolset to perform comparable actions.

2.4 Applications of CRESS

The following is a brief overview of CRESS in selected domains. Refer to the cited papers for more details.

IN: The Intelligent Network is an architecture for telephony networks that separates call routing (service switching) from call processing (service control). In particular, dedicated network elements handle complex services. The IN supports a flexible range of services such as Call Forwarding, Call Screening, Credit Card Calling, and FreePhone. IN services are complemented by ones that reside in switches (exchanges), e.g. Call Waiting or Conference Calling. The major issues in defining IN services are vendor-independence and mutual compatibility.

CRESS has been used to model a range of typical services from the IN [10, 12]. Such descriptions are independent of vendor, platform and language. The CRESS descriptions are interpreted in LOTOS or SDL, allowing rigorous analysis in isolation (checking correctness) and in combination (checking mutual compatibility).

Internet Telephony: Voice calls can be supported over an Internet-like network. This may be the Internet proper, though voice traffic is increasingly being carried over private networks that follow Internet standards. H.323 is a well-established set of standards for digital telephony. However SIP (Session Initiation Protocol [8]) is a simpler and more flexible alternative that is rapidly gaining in popularity. For example, SIP is being widely used to replace conventional telephony, and has been adopted for the new generation (3G) of mobile telephones.

CRESS has been used to model SIP and its associated services. In fact, SIP is sufficiently new that there is not yet a consensus on what a SIP service is. An important aspect of the CRESS work has therefore been to define a SIP service architecture

[11–13]. For formal analysis, a range of SIP services has been described in CRESS and interpreted in LOTOS and SDL. This part of the work has had similar goals to the IN study. To gain practical benefits, the same service diagrams are also translated into Perl for use with SIP CGI (Common Gateway Interface).

IVR: Interactive Voice Response systems are used to provide callers with a voice interface. Speech recognition allows natural language enquiries, and speech synthesis gives fixed or generated voice responses. IVR systems are a major growth area, largely because of customer dissatisfaction with touch-tone enquiry systems. Among competing standards, VoiceXML [16] has gained a dominant position.
CRESS has been used to model VoiceXML and its associated services [12–15]. In fact, VoiceXML does not recognise the concepts of service or feature. Part of the CRESS work has therefore been to investigate the value of these ideas in an IVR setting. For formal analysis, a range of IVR services has been described in CRESS and interpreted in LOTOS and SDL. This part of the work has had similar goals to the IN study. For practical deployment, the same service diagrams are also translated into VoiceXML for deployment in an IVR server.

3 CRESS Notation

CRESS may appear to define state diagrams. However, state is intentionally implicit in CRESS because this allows more abstract descriptions to be given. CRESS has explicit support for defining and composing features. Plug-in domains adapt the notation for selected application areas. Sample diagrams from the domains mentioned earlier are used to illustrate the notation.

3.1 Diagram Expressions

CRESS expressions offer the usual logical, arithmetic and comparison operators. String and set operators are also supported. Assignment is indicated by '$<-$'. Since CRESS acts as a front-end for a variety of target languages, it does not itself define operator precedences. Complex expressions are therefore parenthesised as necessary.

3.2 Diagram Types

A CRESS diagram is a directed, possibly cyclic, graph. Ultimately, CRESS deals with a single diagram. However it is convenient to construct diagrams from smaller pieces. A multi-page diagram, for example, is linked through connectors. More usefully, CRESS supports the notion of feature diagrams that extend or modify other diagrams. This is especially useful for systems that have a base functionality plus additional capabilities. CRESS supports three similar kinds of diagrams:

Root Diagrams describe the fundamental behaviour of a system. An extract from a sample root diagram appears in figure 1. This describes POTS (Plain Old Telephone Service, i.e. an ordinary telephone call). For example the subscriber may go off-hook and dial a number, resulting in both telephones ringing.

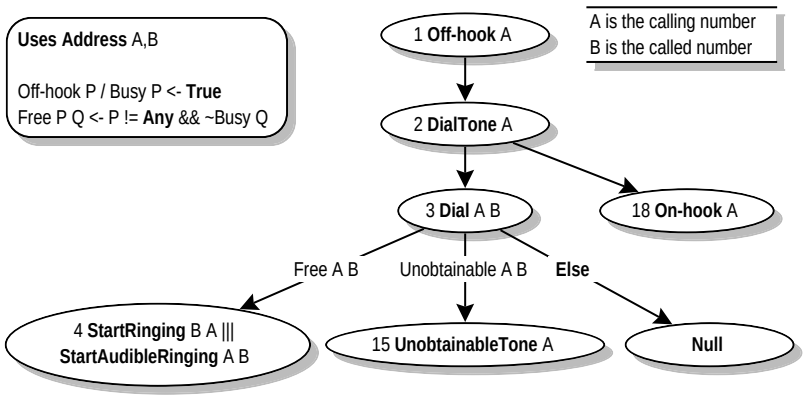

Fig. 1. CRESS Root Diagram (*Incomplete*) for the Plain Old Telephone Service

Spliced (Cut-and-Paste) Feature Diagrams are applied to matching behaviour in another diagram. The feature is copied and used to replace the matching behaviour. Note that this is a static, syntactic operation. A modified diagram is created by the process of composition. A spliced feature has a unique entry node. It has one or more exit nodes that indicate how it flows back to the diagram being modified. A spliced feature can be used to replace, extend or modify behaviour in the original diagram. A sample spliced feature appears in figure 2 that describes the Calling Number Delivery feature in Intelligent Networks. This matches node POTS 4 that follows node POTS 3 via the *Free A B* arc. If the called number *B* has subscribed to the *CallingNumber* feature, the caller's number *A* is displayed. The feature continues from nodes POTS 5 or POTS 13.

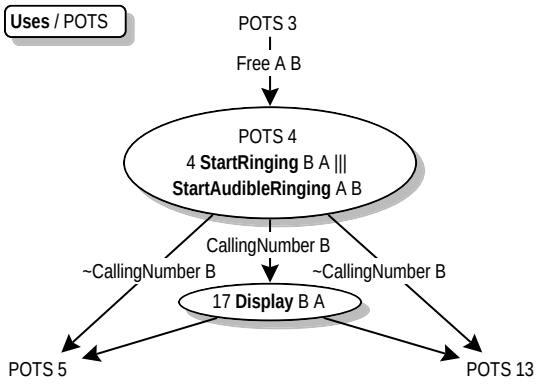

Fig. 2. CRESS Spliced Feature Diagram for Calling Number Delivery

Template (Macro) Feature Diagrams are similar but are parameterised. The actual parameters are determined by pattern matching the triggering node to the template. A template feature may have several exit nodes, but only one of these (**Finish**) continues with the original behaviour. A template is instantiated and applied statically when diagrams are composed. Sample template features appear in figures 3, 4 and 5. Figure 3 describes an Intelligent Networks feature to check if the called number *Q* is busy; if so and the callee has subscribed to call forwarding, the call is diverted to the selected *ForwardBusy* number. Figure 4 describes an Internet Telephony feature that blocks a caller *P* who appears in the screening list *ScreenIn* for callee *Q*. Figure 5 describes an Interactive Voice Response feature for charity donations. It asks the user whether to start again after a request to clear all inputs. Unless the user re-confirms clearing, the donation details are submitted to a server.

It is normally preferable to use template features rather than spliced features. This is because the latter often have to repeat large pieces of the matching diagram. If a feature loops back to its initial node, this is interpreted as meaning a loop back to the triggering node. This allows a chain of features to be triggered by the same node; the instantiated features are combined in sequence.

3.3 Diagram Nodes

Diagrams contain several kinds of nodes, distinguished by shapes that are arbitrary but known: comments, behaviour nodes, labels and rule boxes.

Comments (parallel lines, figure 1) contain explanatory text. Some diagram editors used with CRESS allow hyperlinks to be added as comments, e.g. links to an audio commentary, document or figure.

Behaviour Nodes (ovals) contain behaviours and their parameters. A node is identified by a number, optionally followed by one or more symbols to indicate its kind:

- '<', '>' (figure 4 nodes 1 and 2) means the node contains input, output signals (used if a signal can be sent in both directions)
- '+', '–', '=' (figure 3 node 1, figure 5 node 1) means a template is appended, prefixed, substituted (relative to the triggering node)
- '!' (figure 5 node 3) means a node will not be matched by a template (used to prevent unintended or recursive substitution).

A behaviour node contains signals (inputs or outputs, figure 1 nodes 1 and 2) or actions (like programming language statements, figure 5 node 3). Behaviours may carry variables or expressions as parameters (figure 1 node 1, figure 5 node 2) Several behaviours may occur in parallel ('|||', figure 1 node 4). Each behaviour may be followed by variable assignments separated by '/'.

Labels (plain text or ovals) are used as connectors to join parts of a diagram. A source label cites the diagram name (optional, meaning the same diagram) and node number (figure 2 node POTS 3). A destination label (figure 2 node POTS 5) may define variable assignments much as a behaviour node does.

Special labels are also used. A **Start** node (figure 3) is the initial one of a graph. It is required when a graph is cyclic so the initial node is ambiguous; it is implicit

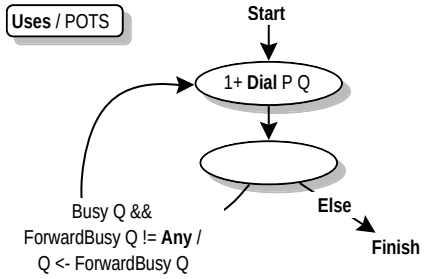

Fig. 3. CRESS Template Feature Diagram for Call Forward on Busy Line

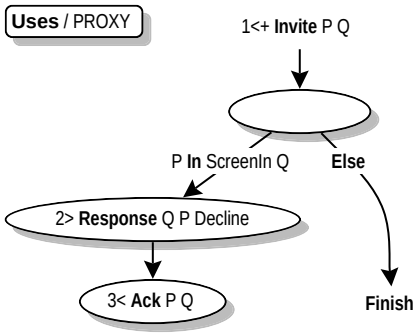

Fig. 4. CRESS Template Feature Diagram for Call Screening

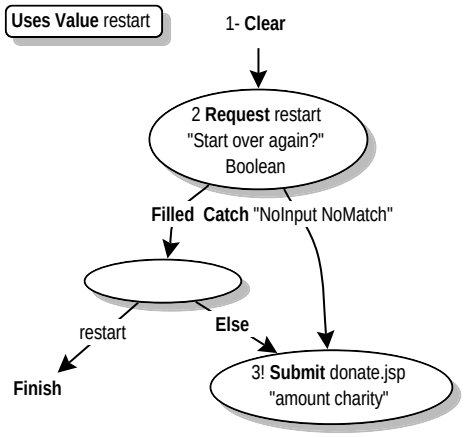

Fig. 5. CRESS Template Feature Diagram for Restarting A Charity Donation

when the initial node is well-defined (e.g. figure 1). A **Null** node (figure 1) does nothing. It may be used as a shorthand to join a number of nodes to a number of other nodes. A **Finish** node (figure 3) is used to indicate the exit of a template. In fact, labels can be left empty for these special nodes (e.g. a **Null** label is normally omitted, figures 3 and 4).

Rule Boxes (rounded rectangles) have multiple purposes. They give a variety of rules to define variables, diagram interdependencies, assignments, macros, signal transformations, and configurations. A **Uses** clause may begin by optionally declaring the variables local to a diagram (figure 1). This is optionally followed by '/' and a list of other diagrams that the diagram depends on (figure 3). For example, a feature diagram lists the root (or other) diagrams that it may modify. The explicit dependencies among diagrams are used to determine the full set of diagrams needed.

A variable initialisation (not illustrated) assigns a value at the start of behaviour. Variables may also be assigned when a certain behaviour occurs. For example in the first rule of figure 1, the occurrence of *Off-hook* for any telephone *P* causes that telephone to be marked as busy. Parameterised macros and their expansion may be defined. In the second rule of figure 1, *Free P Q* expands to a check that *P* is a defined telephone number and *Q* is not busy. Signal transformations (not illustrated) are macros that allow one signal and its parameters to be replaced by another.

Rule boxes may also be used to define a system configuration (not illustrated). A **Deploys** clause lists the behaviour diagrams that apply. In addition, feature parameters may be defined (e.g. the forwarding number for a telephone).

3.4 Diagram Arcs

The arcs between nodes indicate the flow of control. Arcs may be unlabelled (figure 1 node 1 to 2) or may be labelled with guards. These may define value conditions (imposing a restriction on progress, figure 1 node 3 to 4). **Else** means the complement of other value conditions (figure 1 node 3 to **Null**). The use of **Else** is not obligatory, but if it is omitted then a dynamic check can be performed for all guards being unfulfilled.

Guards may also define event conditions that are activated by dynamic occurrence of an event (figure 5 node 2 to 3). Event conditions are distinguished by their names (e.g. **Filled**, **Catch**). Since events may be intercepted at several levels in a CRESS description, there is no equivalent of **Else** for event conditions.

A guard may be followed by assignments separated by '/' (figure 3 empty node to 1). This can be necessary to change system state without executing other behaviour. Sometimes it is convenient to have an empty guard condition but associated assignments, i.e. the progression to a new behaviour node changes the system state.

4 CRESS Syntax and Static Semantics

The syntax and static semantics of CRESS are handled by a preprocessor, a lexical analyser and a parser.

4.1 Specification Architecture Preprocessor

CRESS is driven by a specification architecture for a given domain and target language. This may contain preprocesssor calls that invoke the CRESS tools:

Cress(*Options,Diagrams***):** generates code for the diagrams to be included. Translator options may optionally be given. The diagrams may be listed explicitly, but normally the keyword **Features** is used to mean those diagrams defined by the system configuration. To deal with the situation that a domain may have multiple root diagrams (e.g. Internet Telephony), all diagrams with a certain prefix may be included. For example, **Cress(**PROXY**)** will include all SIP Proxy Server diagrams.

Cress(Profiles): generates code for domain-specific user profiles, e.g. the features and their parameters that subscribers have selected.

Cress(Types): generates code for domain-specific data types.

The preprocessor calls the CRESS lexical analyser and subsequent tools.

4.2 Diagram Lexical Analysis

A graph editor is the most obvious tool to create CRESS diagrams. In fact, the requirements of CRESS are modest. The graph editor must be able to create a small number of node shapes, and must be able to associate multi-line labels with nodes and arcs. For practical reasons, the graph editor must also be multi-platform. A surprising number of graph editors fail to meet these criteria (e.g. Graphlet, GraphMaker, JGraphPad, OpenJGraph, W3Pal). One of the few appropriate graph editors is yEd from yWorks GmbH; this is free and, being Java-based, is multi-platform. It is hoped to use the DiaGen tool in future to create a CRESS-specific editor. Graph editors usually provide analysis features that are irrelevant to CRESS (e.g. finding the minimum spanning tree of a graph). They also tend to draw rather visually plain graphs.

An obvious alternative is a drawing package, but again many of these do not meet the CRESS criteria (e.g. Corel Draw, Dia, jfig, xfig). One of the few suitable drawing packages is Diagram! from Lighthouse Design; the figures in this paper were drawn using this tool. Although Diagram! runs on four different platforms and is free, it requires NEXTSTEP/OPENSTEP which limits its portability.

Another issue is the file representation of graphs. There are many competing formats, a number of which fail to represent the basic graph topology required by CRESS. GML (Graph Modeling Language) and XGMML (eXtensible Graph Markup and Modeling Language) are both suitable. In principle, GraphML (Graph Markup Language) would also be suitable but in practice it is used with proprietary extensions. Support for standard formats among graph editors is rather patchy. CRESS therefore accepts graphs in GML or XGMML format as well as the format created by the Diagram! tool.

The CRESS lexical analyser parses the file representation of a diagram and reduces it to a common internal format. In fact this is tricky, partly because nodes and arcs may appear in any order in the file, and partly because the graph may be cyclic. The generated graph is handed off to the CRESS parser for syntax and static semantic checking.

4.3 Diagram Syntactic Analysis

A list of CRESS diagrams is considered at the same time. There must be at most one root diagram. The other diagrams constitute a hierarchy of features that modify each other or the root diagram. Feature interaction is a well-known problem whereby independently defined features may interfere with each other. A common solution is to prioritise features. In CRESS, feature diagrams have priorities that ensure a well-defined order of application. Each feature in turn is combined with another or with the root diagram. If it is a spliced feature, the feature nodes are cut and pasted into the root diagram. If it is a template feature, the current root diagram is scanned for matching nodes. Each of these is then modified by the instantiated template. The final result is a single diagram. Various manipulations are carried out as this diagram is created:

- Source and destination labels are paired up, joining subgraphs.
- Event names are normalised. CRESS allows some latitude in naming for readability (e.g. *Start Audible Ringing* vs. *StartAudibleRinging*) and for British vs. American English (e.g. *Dialling* vs. *Dialing*).
- '<' and '>' labels on nodes are used to disambiguate signal directions and are then removed. A '!' label preventing template matching is removed after all templates substitutions have been handled.
- The successor nodes of each node are ordered by signal name. This simplifies interpretation in some languages (e.g. SDL).
- **Null** nodes are removed where possible to reduce the size of the graph. This cannot always be done (e.g. in a recursive loop, figure 3).
- Nodes that are reached via an **Else** or an event condition are moved to the end of the node successor list. This simplifies error-checking.
- Since a graph may be cyclic, nodes that appear earlier or later in the graph are specially marked.

The consistency of the final diagram is then checked for syntactic and static semantic correctness. More than 50 checks are applied, including the following as examples:

- A graph must have at most one **Start** node and one **Finish** node.
- Behaviour node labels must be unique.
- A behaviour node must contain events of the same type (input, output, action).
- A branch cannot lead to multiple output behaviour nodes. Non-deterministic inputs are allowed, but not non-deterministic outputs.
- A **Null** node cannot cycle back to itself.
- Expressions must be well-formed and have the correct parameter types for events and operators.
- At most one **Else** may appear in a list of value conditions, and **Else** cannot appear as the only such condition.

The CRESS parser performs the diagram composition, manipulation and checking described above. The result is a graph stored in an internal format. This is handed off to a CRESS code generator for translation to some specification or implementation language.

5 CRESS Dynamic Semantics

For space reasons, the formal interpretation of CRESS is explained here with reference to just one language – LOTOS. The interpretation using SDL is broadly similar, though restrictions on SDL inputs and outputs considerably complicate translation. Code for SDL is generated in SDL/PR (program-like) format.

5.1 Specification Architecture using LOTOS

As an example of what a specification architecture looks like, the following is an outline for Intelligent Network services and LOTOS. There are comparable LOTOS specifications for Internet Telephony and Interactive Voice Response. Domain-defined specification architectures are also defined for SDL, Perl and VoiceXML as appropriate.

```
Specification INSystem [User] : NoExit                          (* Intelligent Network *)
  Library                                                       (* library types *)
  Type Address                                                  (* address operations *)
  Type Addresses                                                (* addresses *)
  Type BooleanOperations                                        (* boolean operations *)
  Type Digit                                                    (* digits *)
  Type Number                                                   (* numbers *)
  Type Statuses                                                 (* call statuses *)
  Type StatusResult                                             (* call status results *)
  Cress(Types)                                                  (* generate types *)
  Behaviour INStructure [User]                                  (* overall behaviour *)
Where                                                           (* local definition *)
  Process INStructure [User]                                    (* IN network structure *)
    Hide Bill,Stat,Scp In                                       (* hide internal signals *)
      (
        (
          (
            CallInstances [Bill,Scp,Stat,User]                  (* call instances *)
          |[User,Stat]|                                         (* synchronise user/status messages *)
            CallCoordinator [User,Stat] ({})                    (* call coordinator *)
          )
        |[Scp]|                                                 (* synchronise service control messages *)
          ServiceControl [Scp,Stat]                             (* service control point *)
        )
      |[Stat]|                                                  (* synchronise status messages *)
        StatusManager [Bill,Stat] (0, Cress(Profiles))          (* generate subscriber profiles *)
      )
    |[Bill]|                                                    (* synchronise on billing messages *)
      BillingSystem [Bill]                                      (* billing system *)
  Process CallInstances [Bill,Scp,Stat,User]                    (* call instances *)
    Cress(Features)                                             (* generate feature behaviour *)
  Process CallCoordinator [User,Stat] (Addresses)               (* call coordinator *)
  Process ServiceControl [Scp,Stat]                             (* service control point *)
  Process StatusManager [Bill,Stat] (Time, Statuses)            (* status manager *)
  Process BillingSystem [Bill]                                  (* billing system *)
```

Node Type	Node Visited Once	Node Visited Earlier	Node Visited Later
Graph Start	instantiate top-level process, whose definition is then begun		
Action	translation is domain-specific		
Input/Output	event, interleaved if events in parallel	call already defined process	start new process definition, then translate input/output as normal
Null	start new process	call already defined process	start new process definition
Value Guard	guard	guard before call of already defined process	guard before call of new process, whose definition is then begun
Event Guard	start new event handler process	not allowed	not allowed
Graph End	close off all process definitions		

Fig. 6. Outline CRESS Denotation in LOTOS

5.2 Interpretation using LOTOS

The dynamic semantics of CRESS is handled by code generators for each target language. Since graphs may be cyclic, a distinction is made between a behaviour node that is visited once, one that is visited earlier in the graph, and one that is visited later. When a code generator walks the graph, it recognises when it has already visited a node. Different code is usually generated for the first and subsequent visits to a node.

A potential difficulty with any automatic code generation is relating the generated code to the original source. In the case of CRESS, the code generators go to a lot of trouble to create human-readable, well laid out code. In addition, virtually every line of the generated code has automatically produced comments. These relate the code directly to the diagram that created it. It is possible to use the generated code without having to be aware of this. However for some purposes (e.g. simulation or verification), the comments are useful for the expert to relate the code and the CRESS diagrams.

Fixed data types are defined by the specification architecture. Type definitions are also automatically generated for signals and events defined by the plug-in domain. Expressions are translated to their LOTOS equivalents. If the domain requires user profiles, these are translated into LOTOS from the CRESS system configuration diagram.

The specification architecture includes fixed process definitions. Diagram-defined processes are generated from nodes according to the outline strategy in figure 6. This gives an idea of the denotations (code skeletons) for various CRESS constructs; it is not possible to define the full mapping here. In the main the translation to LOTOS is straightforward except for the following points:

– Although LOTOS does not really distinguish inputs and outputs, they are translated slightly differently since CRESS outputs must use only constants and variables with defined values. The CRESS toolset performs a data-flow analysis to determine this. A behaviour parameter with an undefined value becomes a '?' event parameter in LOTOS (like input), while a behaviour parameter with a defined value is prefixed with '!' (like output).

– Normally a behaviour node is translated as the corresponding LOTOS event. If the node contains parallel behaviours, these are translated as sequential or concurrent events as defined by a code generator option.
– Behaviour nodes along a path usually become a sequence of LOTOS events. However if several paths lead to a node, a new LOTOS process is defined for the behaviour from that node. A branch to the node then becomes a call of this process.
– Value guards are translated into their direct LOTOS equivalents. An **Else** becomes the logical complement of all the accumulated guards. If an **Else** is omitted, a code generator option produces a dynamic check for the guards being unfulfilled.
– Event guards are very complex to translate into LOTOS. The problem is that the name of a CRESS event may be constructed only dynamically. Where the event is handled is also determined dynamically, as events may be caught at several levels of a CRESS description. Event handling in a LOTOS specification must, however, be defined statically. Fortunately it is possible to statically determine the scope of all event handlers. This allows the translator to define a static process that dispatches events according to their context. A node that follows an event guard will start a new LOTOS process definition. When a CRESS event occurs dynamically, the event dispatcher calls the appropriate process according to its context. More about the event model can be found in [15].

6 Conclusion

The role of CRESS has been seen as a graphical notation for describing system behaviour, particularly for voice services but also for reactive systems generally. The notation, syntax, static semantics and dynamic semantics of CRESS have been discussed. The notion of plug-in application domains makes CRESS very adaptable. CRESS is also powerful in that the same diagrams can be given a formal interpretation or be used for implementation. The use of CRESS has been briefly reviewed for Intelligent Networks, Internet Telephony and Interactive Voice Response.

References

1. A. V. Aho, S. Gallagher, N. D. Griffeth, C. R. Schell, and D. F. Swayne. SCF3/Sculptor with Chisel: Requirements engineering for communications services. In K. Kimbler and W. Bouma, editors, *Proc. 5th. Feature Interactions in Telecommunications and Software Systems*, pages 45–63. IOS Press, Amsterdam, Netherlands, Sept. 1998.
2. D. Amyot, L. Charfi, N. Gorse, T. Gray, L. M. S. Logrippo, J. Sincennes, B. Stepien, and T. Ware. Feature description and feature interaction analysis with use case maps and LOTOS. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 274–289. IOS Press, Amsterdam, Netherlands, May 2000.
3. D. Harel and E. Gery. Executable object modeling with Statecharts. In *Proc. 18th International Conference on Software Engineering*, pages 246–257. Institution of Electrical and Electronic Engineers Press, New York, USA, 1996.

4. ISO/IEC. *Information Processing Systems – Open Systems Interconnection – LOTOS – A Formal Description Technique based on the Temporal Ordering of Observational Behaviour*. ISO/IEC 8807. International Organization for Standardization, Geneva, Switzerland, 1989.
5. ITU. *Message Sequence Chart (MSC)*. ITU-T Z.120. International Telecommunications Union, Geneva, Switzerland, 2000.
6. ITU. *Specification and Description Language*. ITU-T Z.100. International Telecommunications Union, Geneva, Switzerland, 2000.
7. R. Marelly, D. Harel, and H. Kugler. Multiple instances and symbolic variables in executable sequence charts. In *Proc. 17th ACM Object-Oriented Programming, Systems, Languages and Applications*, pages 83–100, New York, USA, 2002. ACM Press.
8. J. Rosenberg, H. Schulzrinne, G. Camarillo, A. Johnson, J. Peterson, R. Sparks, M. Handley, and E. Schooler, editors. *SIP: Session Initiation Protocol*. RFC 3261. The Internet Society, New York, USA, June 2002.
9. T. Rus. A unified language processing methodology. *Theoretical Computer Science*, 281(1–2):499–536, June 2002.
10. K. J. Turner. Formalising the CHISEL feature notation. In M. H. Calder and E. H. Magill, editors, *Proc. 6th. Feature Interactions in Telecommunications and Software Systems*, pages 241–256, Amsterdam, Netherlands, May 2000. IOS Press.
11. K. J. Turner. Modelling SIP services using CRESS. In D. A. Peled and M. Y. Vardi, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XV)*, number 2529 in Lecture Notes in Computer Science, pages 162–177. Springer-Verlag, Berlin, Germany, Nov. 2002.
12. K. J. Turner. Formalising graphical service descriptions using SDL. In R. Reed and J. Reed, editors, *SDL 2003*, number 2708 in Lecture Notes in Computer Science, pages 183–202. Springer-Verlag, Berlin, Germany, July 2003.
13. K. J. Turner. Representing new voice services and their features. In D. Amyot and L. Logrippo, editors, *Proc. 7th. Feature Interactions in Telecommunications and Software Systems*, pages 123–140. IOS Press, Amsterdam, Netherlands, June 2003.
14. K. J. Turner. Specifying and realising interactive voice services. In H. König, M. Heiner, and A. Wolisz, editors, *Proc. Formal Techniques for Networked and Distributed Systems (FORTE XVI)*, number 2767 in Lecture Notes in Computer Science, pages 15–30. Springer-Verlag, Berlin, Germany, Sept. 2003.
15. K. J. Turner. Analysing interactive voice services. *Computer Networks*, Jan. 2004. In press.
16. VoiceXML Forum. *Voice eXtensible Markup Language*. VoiceXML Version 2.0. VoiceXML Forum, Jan. 2003.