Rapid Prototyping with AUTOFOCUS

Franz Huber
Bernhard Schätz[*]

Fakultät für Informatik, Technische Universität München
Arcisstraße 21, 80333 München
email:{huberf|schaetz}@informatik.tu-muenchen.de

Abstract: In most cases, it is simple inconvenience of use that keeps formal methods from being put to industrial use. This paper argues that functionalities, even though of simple formal principles, can be decisive for the applicability of such a formal development tool. Several of those functionalities, as found in the AUTOFOCUS tool prototype, like integrated graphical and hierarchical description techniques, consistency checks and code generation, are demonstrated using a simple example.

1. Introduction

The widely accepted possible benefits of formal methods on the one hand and their minor use compared to informal or graphical description techniques on the other hand have repeatedly lead to the claim that formal methods should be put to a more indirect or transparent use. In [HSS96] we demonstrated how such an indirect approach can be incorporated in the CASE tool prototype AUTOFOCUS by basing it upon formally defined hierarchical description techniques. In [HSE97] we introduced consistency checks for the description techniques and showed how to integrate verification and validation mechanisms into the tool concept.

Despite all the benefits gained from the formally based development, such an approach has little relevance to industry if the specified system cannot be turned into code as the final goal of the development process. Since manual transformation of a specification is both laborious and error prone, automatic code generation should be applied wherever possible.

In this article we sketch how the AUTOFOCUS tool prototype can be enhanced with code generation for simple embedded systems to support the full range of the design process from formal specification to rapid prototyping. The Java programming language [Fla96] has been chosen as target for code generation because of its promising possibilities in embedded and real-time system development. The process of code generation, however, is only defined in a semiformal way, as there is currently no formal semantics available for Java supporting a complete embedding into the formal framework of AUTOFOCUS.

2. AUTOFOCUS

2.1 An Example Application for AUTOFOCUS

We first present a small example to introduce the description techniques used in AutoFocus and, subsequently, to outline to code generation process. Although the example does not cover all aspects of the description techniques, it should be sufficient to illustrate the main ideas behind AUTOFOCUS and the code generation process within the scope of this paper.

The example is of a simple structure, yet with typical aspects of embedded systems: it shows a traffic lights system for a pedestrian crossing. Pedestrians wishing to cross the street can push a button, which in turn switches the lights for the cars to red and the pedestrians' lights to green. After a sufficient delay for the pedestrians, the lights switch back to their original state, allowing the cars to pass along again.

[*] This work was carried out within the Subproject A6 of the "Sonderforschungsbereich 342 (Werkzeuge und Methoden für die Nutzung paralleler Rechnerarchitekturen)" and the Project SysLab, sponsored by the German Research Community (DFG) under the Leibniz program and by Siemens-Nixdorf.

2.2 Description Techniques

AUTOFOCUS, like many tools and methods in practical use, covers different aspects of a distributed system, called views, by using suitable description techniques. These are outlined using the above example in the following sections (for a detailed description see [HSS96]). Several aspects of AUTOFOCUS' description techniques, like component hierarchies and extensive use of Gofer-style data type definitions are not demonstrated in this example. However, within the scope of this paper, it should be sufficient to convey the basic ideas for the prototyping process outlined in the subsequent sections.

2.2.1 Data Type Definitions (DTDs) and Component Data Definitions (CDDs)

DTDs define – additionally to basic AUTOFOCUS data types as also found, for example, in the functional programming language Gofer [Jon91] – constructed types for the data processed and stored by the system components and transmitted across the communication channels. CDDs define the data items (state variables) associated with specific system components and can use definitions from DTDs. In our example, we need to define the data type *TLS*, denoting possible states of the traffic lights for cars. For the pedestrian traffic lights, we define the data type PLS:

```
-- Traffic Lights for cars:
data TLS = R | Y | G | RY;
-- Traffic Lights for pedestrians:
data TLS = R | G;
```

The *Controller* component has one data item to control delays, defined in its CDD:

```
-- Delay counter:
T: Int;
```

2.2.2 System Structure Diagrams (SSDs)

SSDs describe the structure of a system including its interface, its components and the communication paths between them, thus providing both component interface specification and topological information. Figure 1 shows an SSD for the traffic lights example (not all port names are shown there for reasons of clarity). It consists of the components *Controller*, *TrafficLights*, *PedsLights*, *ButtonA*, and *ButtonB* (for each side of the street).

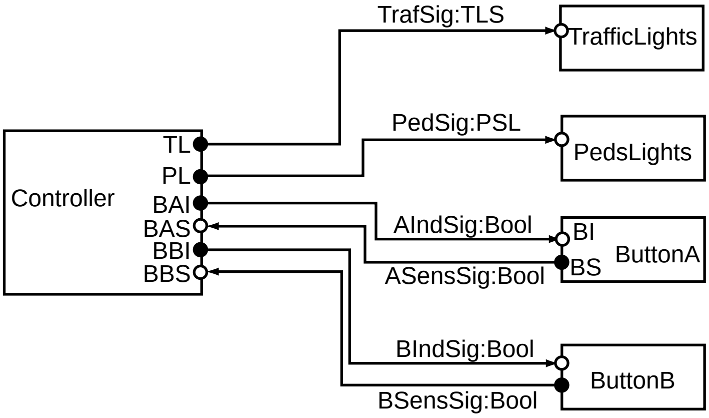

Figure 1. System Structure Diagram of the Traffic Lights Example

To cross the street, a pedestrian can press the push-button on his side (*ButtonA* or *ButtonB*) and send a signal to the *Controller* to initiate a green phase. A signal light in each button lights up confirming the action. The traffic lights switch, with some delay, from green via yellow to red. After a short green phase for the peds light, the traffic lights are returned to their original state. To set the traffic and indicator lights, the controller sends signals via the channels *TrafSig*, *PedSig*, *AIndSig*, and *BIndSig*. The pressing of the buttons is transmitted via the *ASensSig* and *BSensSig* channels. Signals sent over those channels arrive at the input ports of the controller (*BAS*, *BBS*), signals sent to the lights originate from its output ports (*TL*, *PL*, *BAI*, *BBI*).

Components can be hierarchically refined by networks of sub-components, a property which is not used in this example.

2.2.3 State Transition Diagrams (STDs)

STDs are used to describe the behaviour of the system as a whole as well as of its components. In case of hierarchical refinement of system components, it is possible to have an STD describing a component's behaviour and STDs describing the sub-components' behaviours. This allows to use behavioural refinement in the development process. For the prototyping approach presented here, however, it is sufficient if each leaf component in a refinement hierarchy has an associated STD. Figure 2 shows the STD describing the behaviour of the *Controller* component.

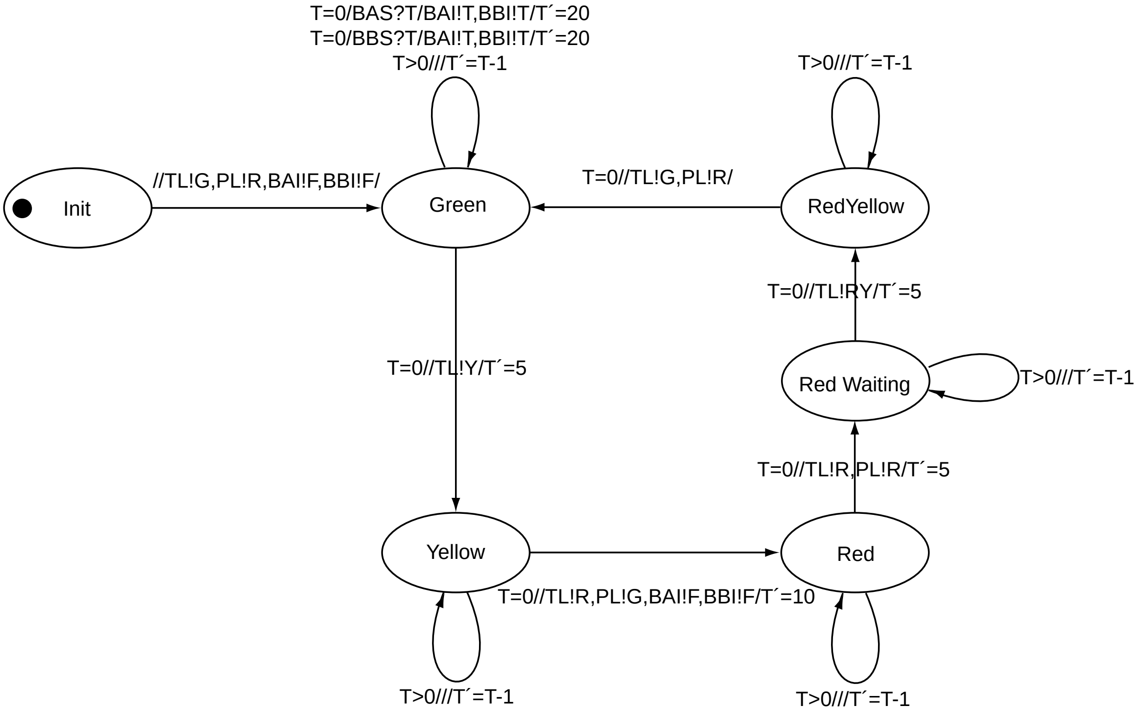

Figure 2. The *Controller* Component's State Transition Diagram

Each transition has a set of annotations: a pre- and a post-condition, encoded as predicates over the data state of the system satisfied before and after the transition, and a set of input and output actions describing the messages that are read from or written to the input and output ports of the corresponding component. The notation used to specify the input and output actions is similar to CSP.

Consider, for example, the looping transition of state *Green* labelled with the annotation

```
T=0/BAS?T/BAI!T,BBI!T/T´=20
```

Here, the precondition *T=0* will only allow the selection of the transition if the delay counter T has the current value 0, i.e., no green phase is already initiated. Furthermore, this transition will need a button sensor signal on port *BAS* to be selected. If the transition is selected, the indicator lights in the buttons will be lit by sending the corresponding signals on ports *BAI* and *BBI*. Since the postcondition *T´=20* will hold after the selection of the transition, the green phase is initiated with an initial delay.[1]

2.2.4 Extended Event Traces (EETs)

EETs, similar to MSCs [Int96], illustrate the interaction of the system components and their environment via message exchange. In AUTOFOCUS, they are often used for specification of use cases in the early stages of development, to derive, from the requirements captured within them, the system design using SSDs, STDs, and DTDs/CDDs. Thus, a system description is complete without EETs and therefore they are not used for the code generation in

[1] Like in temporal logics, we use primed state variables to denote the values of variables *after* the transition and unprimed variables to denote the values *before* the transition is selected.

our prototyping approach. However, as outlined in [HSE97], it is planned to generate EETs from runs of the generated prototype system.

Similar description techniques like SSDs, STDs, and EETs can be found in many comparable specification frameworks, like system and automata diagrams in Statecharts [Har90] or ROOM [SGW94], or system diagrams, MSCs and data type definitions used in SDL/GR.

2.3 Consistency

In the AUTOFOCUS approach, the information describing a system is spread out across several documents of different kinds, just like in different modules or libraries of a large programming package. If a system specification exceeds the toy world size, keeping these development documents consistent becomes more and more complex for the developer. Here, simple checks based on the abstract syntax of the description techniques and carried out by the tool can already be an enormous help. Since AUTOFOCUS uses different document classes as well as hierarchically organized document structures, it becomes even more important to make sure that the spread out information is consistent. For a detailed description of consistency checks, their definition, and their user interface see [HSE97].

We distinguish two forms of consistency conditions, depending on how those conditions can be checked:

Intra-document consistency: Those are consistency conditions that can be formulated using only elements of one document, for example, an SSD or a DTD. An example for this consistency is the type equivalence of channels and their associated ports. In the traffic light example, the channel *AIndSig* and the ports *BAI* and *BI* all have to be of type *Bool*.

Inter-document consistency: Those are consistency conditions that can only be formulated using two ore more different documents of the same or a different document class. An example is the type correctness of transition annotations. In the traffic lights example, to check the consistency of the transition annotation as discussed in section 2.2.3, we have to make sure, e.g., that the controller has ports *BAS*, *BAI* and *BBI* of type *Bool* as well as a variable *T* defined to be of type *Int*.

Consistency of all development documents is a *prerequisite* for the code generation process: in case of incomplete or contradictory development documents, the code generated using them might be incorrect as well.

3. Prototyping

Once a consistent and complete set of SSDs, STDs and DTDs is defined, code generation can be used to produce an executable version of the specification, thus yielding a prototype of the system. Complementing validation and verification techniques described in [HSE97], the prototype can be used to analyse the behaviour of the system by embedding it into an environment where the execution of the generated code can be visualised and thus monitored by the user.

Based on a synchronous semantics for the behaviour of distributed systems defined in [HSE97], we sketch how Java code can be generated from a system specification in AUTOFOCUS, using parts from the traffic lights example. Finally, we show how the embedding of the generated code in a simulation environment can be used for on-line validation of specifications.

3.1 Java Code Generation

As stated before, due to the lack of a formal semantics of Java, we can only support the claim that the generated Java code meets the formal semantics of a specification without giving a mathematical proof. However, within the AUTOFOCUS project, this approach to code generation makes up only a first prototypical step towards an integrated development process from specification to implementation.

Within this approach, we made the following decisions on basic concepts for the code generation:

1. We provide a very simple framework of Java classes encapsulating some Focus core concepts, e.g., communication ports, channels, and base classes for components.
2. Components are implementations of the Java interface *Runnable*, each one running independently within its own thread.

3. State Transition Diagrams are generated only for leaf components in a refinement hierarchy. Although it is possible to specify behaviour using STDs for components refined by a network of sub-components as well, only those STDs associated with the components at the finest level of granularity are used for code generation.
4. The component behaviour specified in STDs is generated “hard-wired” into the code implementing the *run()* method of each component.

An alternative solution to decision 2 especially considering the synchronous semantics, is based on the use of a central scheduler component to co-ordinate the runs of the individual components, using a central clock. However, in the approach chosen, synchronicity of communication is achieved by the implementation of the communication channels and ports buffering at most one element. In our view, this approach has two main advantages over using a central scheduler:

- Having each component run independently is closer to the intuition of a distributed system of autonomous components.
- Due to the flexibility of this approach changes in the code generation algorithms are minimised when using a different semantics. In order use an asynchronous semantics, where the components communicate using buffered channels of an arbitrary (but limited) length, mainly the channel and port classes provided in our framework have to be adapted to buffer data, the code generation process in this case is almost unaffected.

In the following sections we will briefly sketch how the code generation maps the elements of the AutoFocus description techniques to Java language elements.

3.1.1 The Structure: Components, Ports, and Channels

The components of a distributed system in AutoFocus are individual entities. Each one of them has its own behaviour (specified by an STD), its own set of communication ports, and its own data attributes.

For each component, a separate Java class is generated. This is mainly due to decision 4 outlined above. By encoding the behaviour of a component “hard-wired” into its execution method, this method is naturally different among different components. A more general approach, using instantiation of a pre-defined component class, with a state transition table object encoding the component’s behaviour, would be more elegant, but much more complex as well.

The generated component classes are, however, derived from a common abstract base class *FocusComponent* that provides the essential protocols for the component to be executed within a Java thread and to be embedded into an environment (see also section 3.2) for simulation.

The data attributes as well as the ports associated with components are represented as attributes in the generated class. Thus, a part of the generated class for the *Controller* component from our example looks like this:

```
class Controller extends FocusComponent {
  protected int T;

  public IntPort TL = new IntPort("TL");
  public IntPort PL = new IntPort("PL");
  public BooleanPort BAI = new BooleanPort("BAI");
  public BooleanPort BAS = new BooleanPort("BAS");
  public BooleanPort BBI = new BooleanPort("BBI");
  public BooleanPort BBS = new BooleanPort("BBS");

  public Controller(String ident) {
    super(ident);
    state = "Init"; // set initial state
  }

  public void run() {...} // for this method, see section 3.1.3
  }
```

Obviously, all data associated with the *Controller* component are represented by protected member variables.

The current state of a component is represented by a *String*-valued attribute defined in the base class *FocusComponent*. It is set to its initial value as specified in the component's STD in the constructor.

All ports are declared public. Since they naturally define a component's interface to its environment, they are known to the environment, which is reflected in the Java code as well. Java does not offer the possibility to declare generic classes (known as templates in C++). Thus, for each data type in a distributed system, a separate port class handling that data type has to be explicitly generated (*IntPort* and *BooleanPort* in our example). Again, these generated port classes can be derived from a pre-defined class that offers basic functionality. Relying on the consistency of the system specification, an alternative approach is possible by converting all data to a String representation before sending them and re-converting them to the original format upon reception. In our view, however, such an approach is less elegant than the previously mentioned solution.

Ports have a set of operations to allow components to read data from them or to write data to them. In case of an *IntPort* these operations are

- `public void write(int val)` writes an *int* value to the port.
- `public void clear()` deletes any values previously present on the port.
- `public boolean available()` reports if a valid input value is available on the port.
- `public int read()` reads the value from the port.

In AUTOFOCUS, channels connect two ports. Channels have an associated data type describing the type of data sent on them. Thus the problem caused by Java's lack of generic classes outlined above for the ports applies here as well. A separate channel class has to be generated for each data type used in the system.

Channels propagate data produced by a component on a port during a transition of the component's STD to the destination component's input port. In case of a flat, not hierarchically refined component network, this process is straightforward. The originating port writes the data to the channel, the channel in turn writes the data to the input port, making them available for the destination component to process. However, in a network of hierarchically refined components, which will be the usual case in practice, this solution is not sufficient. The data to be transmitted has to arrive at its destination "immediately", even though it has to pass a possibly large number of ports connected by a sequence of channels, as defined in the SSDs (see, e.g., Figure 3).

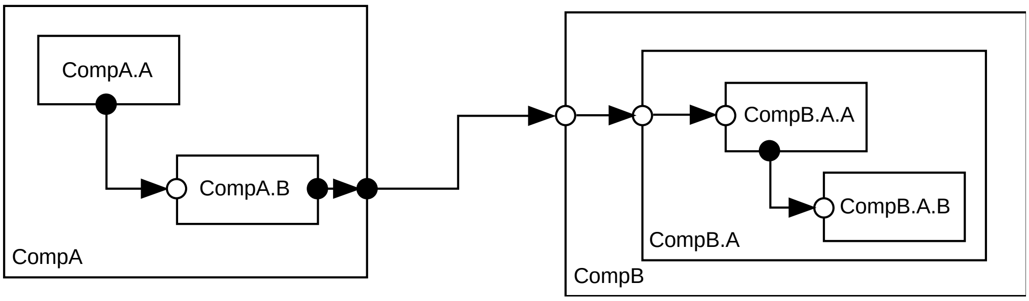

Figure 3. Hierarchical Component Network

In this network the sub-components CompA.A and CompA.B as well as CompB.A.A and CompB.A.B are directly connected, whereas the sub-components CompA.B and CompB.A.A are connected via a sequence of channels and ports belonging to their super-components CompA, CompB, and CompB.A. In this case, the generated code must ensure that data sent from CompA.B to CompB.A.A across several port and channel objects arrives "immediately" as well. This is accomplished by two properties in the generated code:

- Ports, upon reception of data (from the actions associated with a transition in an STD or from an incoming channel), immediately try to write the data to the outgoing channel, provided there is an outgoing channel. If there is none, then the port belongs to a leaf component that is not further refined by a network of sub-components.

- Channels write data received from an incoming port directly to their corresponding outgoing port which has to exist, otherwise the system would not have passed the consistency check.

The resulting cascade of `write()` method calls ensures that the data is actually transferred to the destination component in time for the next execution step.

3.1.2 Data Type Definitions

As stated above, data associated with system components are converted to instance variables of the corresponding component classes in Java. In this conversion, a canonical mapping between the AUTOFOCUS basic data types and Java is used. For instance, the built-in types *Bool*, *Char*, and *Int* are mapped to their respective counterparts *boolean*, *char*, and *int* (primitive data types) in Java. AUTOFOCUS *Strings* are mapped to *String* objects in Java, *Lists* are converted to *Vector* objects. Tuple types available in AUTOFOCUS can be straightforwardly represented in Java by classes with public attributes corresponding to the elements (and types of the elements) in the tuple. Simple user-defined data types, like enumeration-style data types, are used frequently. In our traffic lights example, the declaration

```
data TLS = R | Y | G | RY;
```

can be represented in Java either by mapping the individual values to *int* or by wrapping them up as symbolic, yet integer-valued constants, in a separate class:

```
class TLS {
    public static final int R = 1;
    public static final int Y = 2;
    public static final int G = 3;
    public static final int RY = 4;
}
```

The constants can then be accessed by their symbolic name, like, e.g., `TLS.RY`. Other user-defined data types can easily become very complex and thus difficult to convert to Java classes.

In any of these mappings, the operations on the AUTOFOCUS data types, that are, e.g., specified within textual pre- or post-conditions annotated to transitions in an STD (see also section 3.1.3), have to be parsed and then expressed using the operations and methods available for the respective primitive data types and classes in Java.

3.1.3 State Transition Diagrams

State transition diagrams describe component behaviour depending on the internal state (both the control state in the sense of a finite automaton state and the state of a component's variables) and on the input data received on the input channels of the component. A transition is selected depending on the current control state, the state of the variables and the data read on the input ports. A transition can result in a change of the internal state and in data being written to the output channels. Based on the underlying synchronous semantics, one execution step of a component, i.e., of its STD, consists of the following steps:

- Read all available data from all input channels.
- Select an applicable transition; selection is done non-deterministically if more than one transition is applicable.
- Fire a transition, if possible; change the local variables and produce data for the output channels, if necessary.
- Write all available data to all output channels. These data are then available for processing in the next step.

From the graphical representation of STDs in AUTOFOCUS and their textual annotations to the transitions "hard-wired" Java code is generated, in the form of nested *if* clauses that cover all specified combinations of states and available inputs. This code, which actually provides the implementation of a component behaviour, makes up the main part of the `run()` method of the generated component classes. For the *Controller* component in our example, a part of the "generated" code might look like this (see Figure 2 for a comparison with the graphical notation):

```
public void run() {
    String newState = "";

    while (true) {
      while (!ready());                                       // (1)
      if (state == "Init") {
        // pre-condition always true:
        if (true) {
          BAS.read();
          BBS.read();
          TL.write(TLS.G);
          PL.write(PLS.R);
          BAI.write(false);
          BBI.write(false);
          // set the target state:
          newState = "Green";
        } // no alternative transitions here
      } else if (state == "Green") {
        if (T == 0 &&                                         // (2)
            BAS.available() && BAS.read() == true) {          // (3)
          BBS.read();
          TL.clear();                                         // (4)
          PL.clear()
          BAI.write(true);
          BBI.write(true);
          T = 20;                                             // (5)
          newState = "Green";                                 // (6)
      } else if (T == 0 &&
            BBS.available() && BBS.read() == true) {
          ...
        } ...
      } else if (state == "Yellow") {
        ...
      } ...
      // notify an observer:
      // (see section 3.2 for this)
      notifyObservers(new StateTransition(state, newState)); // (7)
      // effectively set the new state:
      state = newState;                                     // (8)
    }
  }
```

Obviously, the first level of *if* clauses covers all states in the STD, whereas the second level includes, for each state, all specified pre-conditions and input combinations.

Each iteration of the `run()` method first waits until all input ports have received their new values from the channels connected to them (see label 1). This is achieved by a method `ready()` that checks whether all ports have been assigned their new input values.

In case a transition can fire, i.e., its pre-condition is satisfied (2) and the values available on the input ports correspond to the expected values (3), the output values are written to the output ports (4). Note that ports that no data is written to are explicitly cleared and that ports that not data needs to be read from to satisfy the input pattern are explicitly read from as well. These two mechanisms, together with corresponding status flags in the ports ensure synchronicity in the distributed system. The post-conditions specified for the transition are then evaluated, i.e., executed (5). Upon completion, the new state of the component is stored in a temporary variable (6), and then finally, the new state becomes the current state of the component (8). The reason (7) for this delayed assignment of the state (6, 8) will be explained in section 3.2.

3.2 Simulation and Embedding

In this approach, simulating a distributed system for the purpose of validation basically consists of executing the generated code. For typical requirements of simulation, like visualising the progress of runs of components, stopping, restarting, or resetting the control and data state of components, additional functionality has to be generated. Within the component classes, hooks have to be provided to insert calls to visualisation tools in order to regularly update a visual representation of the states of components. To provide user control over the simulation run, methods must be included in the component classes to stop, suspend, and resume a run, and to view and alter the data variables of the components.

To accomplish this we use a Model/View design pattern [GHJV94] provided by the Java libraries, namely the *Observable* class and the *Observer* interface. An *Observable* is an object that maintains a list of *Observer* objects interested, e.g., in changes in the state of the *Observable*. Using a standardised notification mechanism, the *Observable* object can notify its *Observer* objects about events of interest, optionally passing along objects as parameters that provide more detailed information about the events.

To notify a simulation environment, into which the generated code is embedded, about events that have to be visualised, the *FocusComponent* class is derived from *Observable*. By defining a class

```
class StateTransition {
  public String previous;
  public String current;
  public StateTransition(String prev, String curr) {
    previous = prev;
    current = curr;
  }
}
```

an object of this class can be created upon completion of every transition performed by a component and subsequently sent to *Observers* in the simulation environment. An observer implemented by an STD viewer for that component could then highlight the graphic representation of the new state reached by the transition.

In the opposite direction, this Model/View concept is applicable as well. By deriving control elements within the simulation environment from *Observable* and by implementing the *Observer* interface in *FocusComponent* or in the thread executing a run of a component, user interactions like stopping, suspending, and resuming the runs of individual components can be realised.

To visualise simulation runs in the format of graphical runtime protocols, e.g., as communication histories of the components involved, EETs can be used. These can be generated using the same mechanism as described above. Upon each sending and reception of data, an *Observer* object responsible for recording the communications in the system is notified, together with information about source and destination components as well as the actual data been sent.

3.3 Implementation

Currently, some very simple manually converted examples exist as “proof of feasibility” for the code generation process. A subset of the ideas outlined in the previous sections is currently implemented in a practical project course in software engineering at TU München by a group of students during the 1997 summer term. Within this course, general AutoFocus data types and their mapping to Java will not yet be realised. A simplified version is implemented with only simple Java data types and basic operations on them in CDDs and in STD pre- and post-conditions instead of general data types and operations.

4. Future Work

In this paper we are primarily interested in using generated Java code for validation purposes. Therefore, questions about how to generate Java code that can be directly used to implement functionality of embedded controller systems, or even directly downloaded into such systems are not treated here. With upcoming real-time Java environments[2] this task becomes more and more important for the future development of AutoFocus.

However, code generation as the goal of systems development can only be an integral part of a formal development process if the target language is based on semantic foundations as strong as the description techniques used to specify and design the system. Without these, the generated code can only be used to validate system properties on a semiformal basis. Thus a semantics for Java on a formal basis is required for a consistent development process from specification to code generation. Ongoing projects aiming at a formalisation of Java are a possible basis for this in the future.

5. Bibliography

[Fla96] Flanagan, D. Java in a Nutshell. O'Reilly & Associates, Inc. Sebastopol, CA, 1996

[GHJV94] Gamma, E. Helm, R. Johnson, R. Vlissides, J. Design Patterns: Micro-Architectures for Reusable Object-Oriented Design. Addison-Wesley, 1994

[GKRB96] Grosu, R. Klein, C. Rumpe, B. Broy. M. State Transition Diagrams. Technical Report TUM-I9630. Technische Universität München, 1996.

[Har90] Harel, D. StateMate: A Working Environment for the Development of Complex Reactive Systems, IEEE Transactions on Software Engineering. 16:403-413. 1990.

[HSE97] Huber, F. Schätz, B. Einert, G. Consistent Graphical Specification of Distributed Systems. To appear in proceedings of FME'97.

[HSS96] Huber, F. Schätz, B. Spies, K. AutoFocus - ein Werkzeug zur Beschreibung verteilter Systeme. In: Herzog, U. Hermanns, H. (eds.) Formale Beschreibungstechniken für verteilte Systeme. Proceedings of the 6th GI/ITG Colloquium. Universität Erlangen-Nürnberg, 1996.

[HSSS96] Huber, F. Schätz, B. Schmidt, A. Spiesk, K. AutoFocus-A Tool for Distributed System Specification. In: Bengt, J. Parrow, J. (eds.) Proceedings FTRTFT´96. Lecture Notes in Computer Science 1135, Springer 1996, pp. 476-470.

[Int96] International Telecommunication Union, Geneva. Message Sequence Charts 1996. ITU-T Recommendation Z.120. Geneva, 1996

[Jon91] Jones, M.P. Introduction to Gofer 2.20. Technical Report. Yale University, 1991

[SGW94] Selic, B. Gullekson, G. Ward, P. Real-Time Object-Oriented Modeling. John Wiley & Sons, Inc. 1994

[2] e.g.: PERC (Portable Executive for Reliable Control), see: http://www.newmonics.com/WebRoot/technologies/PERC.html