From HELENA Ensemble Specifications to PROMELA Verification Models

Annabelle Klarl

Ludwig-Maximilians-Universität München, Germany*

Abstract. With HELENA, we introduced a modeling approach for distributed systems where components dynamically collaborate in ensembles. Conceptually, components participate in a goal-oriented collaboration by adopting certain roles in the ensemble. To verify the goal-directed behavior of ensembles, we propose to systematically translate HELENA specifications to PROMELA and verify them with the model-checker Spin. In this paper, we report on tool support for an automated transition from HELENA to PROMELA. Relying on the XTEXT workbench of Eclipse, we provide a code generator from the domain-specific-language HELENA-TEXT to PROMELA. The generated PROMELA model simulates the two layers, components and their adopted roles from HELENA, and allows dynamic role creation as well as asynchronous communication of roles.

1 Introduction

Ensemble-based systems are large distributed systems where components dynamically collaborate to achieve common goals. In HELENA [5], such systems are modeled by dynamically evolving *ensembles* where participating components adopt (possibly concurrently) different *roles*. The concept of roles allows to focus on the particular tasks which components fulfill in collaborations and to structure implementation by realizing roles as threads executed on top of components [9].

Ensembles are formed to collaborate for some global goal. Such goals are often temporal properties and are therefore specified in linear temporal logic (LTL) [11]. To allow verification of HELENA models for their intended goals, we already proposed in [6] to translate HELENA ensemble specifications to PROMELA and check satisfaction of goals specified in LTL with the model-checker Spin [7]. We proved the correctness of the translation for a simplified variant of HELENA which restricts ensemble specifications to their core concepts.

In this paper, we report on the extension of the translation to full HELENA and its automation based on the XTEXT workbench of Eclipse. With the extended translation, we are able to simulate the two layers of HELENA, components and their adopted roles, in PROMELA. Due to the automation of the translation, we augment HELENA ensemble specifications with immediate verification support in Spin. To this end, an Eclipse plug-in is implemented which produces an executable PROMELA specification from a HELENA ensemble specification written in the domain-specific language HELENATEXT [8].

* This work has been partially sponsored by the EU project ASCENS, 257414.

2 HELENA in a Nutshell

Ensemble Structures: The foundation of HELENA ensembles [5] are *components* characterized by their type. They manage associations to other components and store basic information, that is useful in all roles the components can adopt, in attributes. They also provide operations which can be invoked by their roles. *Roles* are classified by role types. Given a set CT of component types, a *role type* rt is a tuple $(rtnm, rtcomptypes, rtattrs, rtmsgs)$: $rtnm$ is the name of the role type; the set $rtcomptypes \subseteq CT$ determines the component types which can adopt the role; the set $rtattrs$ allows to store data that is only relevant for performing the role; the set $rtmsgs$ determines the incoming and outgoing messages supported by the role. To define the structural characteristics of a collaboration, an *ensemble structure* specifies the role types whose instances form the ensemble, determines how many instances of each role type may contribute by a multiplicity (like 0..1, 1, $*$, 1..$*$ etc.), and defines the capacity of the input queue of each role type. We assume that between two role types the messages which are output on one side and input on the other side can be exchanged.

Example: The use of HELENA is illustrated at a peer-2-peer network supporting the distributed storage of files which can be retrieved upon request. Fig. 1 shows a graphical representation of the corresponding ensemble structure. It consists of three role types (`Router`, `Requester`, `Provider`) with associated multiplicities and input queue capacities. All role types can be adopted by components of the type `Peer`. The peer adopting the role `Requester` wants to download the file from the network, peers adopting the role `Router` forward the request through the network, and the peer adopting the role `Provider` finally provides the file. Messages are annotated at the arrows connecting two roles. Attributes are only shown for `Peer` and `Requester`. The full specification can be found in [10].

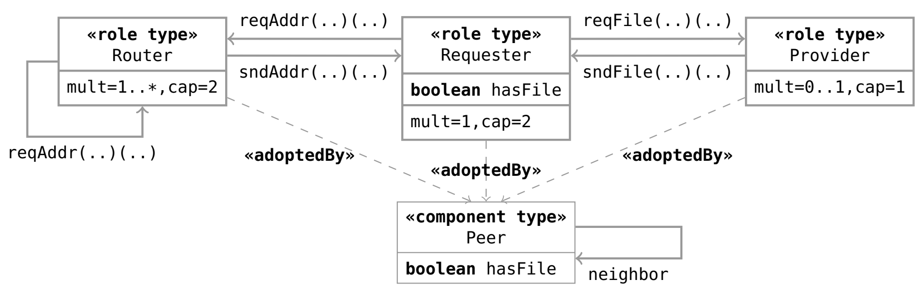


Fig. 1: Ensemble structure for the p2p example in graphical notation

Ensemble Specifications: The *behavior* of a role is specified by a process expression built from the null process **nil**, action prefix $a.P$, guarded choice **if**$(guard_1)\,\{P_1\}$ **or**$(guard_2)\,\{P_2\}$ (branch is nondeterministically selected if several branches are executable), and process invocation [6]. Guards are predicates over component or role attributes. There are actions for creating (**create**) and retrieving (**get**) role instances, sending (!) or receiving (?) messages, and invoking operations of the owning component. These actions must fit to the declared ensemble structure, e.g., messages can be only sent by roles which declare them. Additionally, state labels are used to mark a certain progress of execution in the role behavior. Fig. 2 shows the behavior specification of a `Router`. Initially, a router can receive a request for an address. Depending on whether its owner has the file, it either creates a provider role instance and sends it back to the requester in $P_{provide}$ or forwards the request to another router in P_{fwd} if possible.

$$
\begin{aligned}
\textbf{roleBehavior}\ \mathit{Router} &= ?\mathit{reqAddr}(\mathit{Requester}\ \mathit{rq})()\ .\\
&\quad\ \textbf{if}\ (\textbf{owner}.\mathit{hasFile})\ \textbf{then}\ \{P_{provide}\}\\
&\quad\ \textbf{or}\ (!\textbf{owner}.\mathit{hasFile})\ \textbf{then}\ \{P_{fwd}\}\\
P_{provide} &= p \leftarrow \textbf{create}(\mathit{Provider}, \textbf{owner})\ .\ \mathit{rq}!\mathit{sndAddr}(p)()\ .\ \textbf{nil}\\
P_{fwd} &= \textbf{if}\ (\textbf{plays}(\mathit{Router}, \textbf{owner}.\mathit{neighbor}))\ \textbf{then}\ \{\textbf{nil}\}\\
&\quad\ \textbf{or}\ (!\textbf{plays}(\mathit{Router}, \textbf{owner}.\mathit{neighbor}))\ \textbf{then}\ \{P_{create}\}\\
P_{create} &= r \leftarrow \textbf{create}(\mathit{Router}, \textbf{owner}.\mathit{neighbor})\ .\ r!\mathit{reqAddr}(\mathit{rq})()\ .\ \mathit{Router}
\end{aligned}
$$

Fig. 2: Role behavior of a `Router` for the p2p example

A complete collaboration is given by an *ensemble specification* consisting of an ensemble structure Σ and a set of role behaviors, one for each role type in Σ.

Semantics: Ensemble specifications are semantically interpreted by labeled transition systems, i.e., ensemble automata [5,6]. Ensemble states capture the currently existing role instances with their data and control states. Transitions between ensemble states are triggered by role instance creation or retrieval, communication actions, and operation calls. The communication style (synchronous or asynchronous) is determined by the size of the input queues of the role types.

Goal Specifications: Goals are expressed by LTL formulae over particular HELENA propositions: A *state label proposition* is of the form $rt[n]@label$. It is satisfied if there exists a role instance n of type rt whose next performed action is the state label *label*. An *attribute proposition* must be boolean and is built from arithmetic and relational operators, data constants, and propositions of the form $rt[n]{:}attr$ (or $ct[n]{:}attr$). An attribute proposition $rt[n]{:}attr$ is satisfied if there exists a role instance n of type rt such that the value of its attribute $attr$ evaluates to **true** (and analogously for component attributes). LTL formulae and their satisfaction are inductively defined from HELENA propositions, propositional operators $\neg$ and $\wedge$ and LTL operators $\mathbf{X}, \Box, \Diamond, \mathbf{U}$ and $\mathbf{W}$ as usual.

For the p2p example, we want to express that the requester will always receive the requested file if the file is available in the network. We assume a network of three peers and formulate the following achieve goal in LTL which refers to the values of the attribute `hasFile` of component type `Peer` and role type `Requester`:

$$(Peer[1]{:}hasFile \vee Peer[2]{:}hasFile \vee Peer[3]{:}hasFile) \Rightarrow \Diamond Requester[1]{:}hasFile)$$

3 Translation from HELENA to PROMELA

To verify HELENA specifications for their intended goals, we rely on the model-checker Spin [7]. In [6], we discussed that the translation of a simplified variant of HELENA to PROMELA preserves satisfaction of $LTL_{\setminus\mathbf{X}}$, the fragment of LTL that does not contain the *next* operator **X**. This translation abstracts from the underlying component-based platform and considers only role types and their interactions. In role behaviors, guarded choice and arbitrary process invocations are not allowed and any notion of data is omitted. To cope with these features, we propose to represent components and roles by two kinds of processes in PROMELA. They differ in communication abilities and behavior since components are only storage and computing resources while roles are active entities.

Communication Abilities: (1) Components only interact with roles, but not with other components. Roles advise components to adopt other roles, request references to already adopted roles from their owning components, or invoke operations on them. Thus, each PROMELA process for a component relies on a dedicated synchronous channel **self**, only used for communication between itself and its adopted roles. The roles refer to the channel under the name **owner**. (2) Roles interact by exchanging directed messages on input queues. Thus, each PROMELA process for a role relies on a dedicated (possibly asynchronous) channel **self** in addition to the aforementioned channel **owner** to model its input queue. Since channels are global in PROMELA, but input queues are local in HELENA, special care has to be taken that this channel is only available to processes which are allowed to communicate with the corresponding role in HELENA.

Behavior: (1) The PROMELA process for a component implements a **do**-loop to wait for requests from its roles on the **self** channel. Depending on the request, it runs some internal computation and sends back a reply. E.g., to adopt a role, it creates a new channel and spawns a new process (representing the role) to which it hands over its own **self** channel as the role's **owner** channel and the newly created channel as the role's **self** channel. Afterwards, it sends the role's **self** channel to the role requesting the adoption such that the two roles can communicate via this channel. (2) The role behavior of a HELENA specification must be reflected in the corresponding PROMELA process. In [6], we proposed to translate action prefix to sequential composition in PROMELA, nondeterministic choice to the **if**-construct, and recursive behavior invocation to a **goto** to the beginning of the role behavior. Sending and receiving messages was mapped to message exchange on the **self** channel of roles and role creation to process creation with the **run**-command. To extend this to full HELENA, guarded choice is translated to the **if**-construct with the guard as first statement. Arbitrary process invocation is realized by jumping to labels marking the beginning of processes. On the level of actions, we extend message exchange by data relying on user-defined data types in PROMELA. Furthermore, to cope with the component level of HELENA, a new role is now created by issuing an appropriate request on the owning component and spawning the new role process from there. The introduction of components also allows us to implement role retrieval and operation calls by corresponding requests from a role to a component.

$LTL_{\setminus\mathbf{X}}$ **Preservation:** Similarly to the simplified translation in [6], all HELENA constructs are directly translated to PROMELA while introducing some additional silent steps like **goto**s. These do not hamper stutter trace equivalence and thus satisfaction of $LTL_{\setminus\mathbf{X}}$ is preserved, though not formally shown here.

4 Automation of the Translation

To automate the translation, a code generator, taking a HelenaText [8] ensemble specification as input, was implemented on top of the Xtext workbench of Eclipse relying on Xtend as a template language.

Component Types: For each component type, the excerpt of the Xtend template in Fig. 3 generates a new process type in Promela. Most importantly, this process type implements a `do`-loop (line 4-10) where it can repeatedly receive requests from its adopted roles via its **self** channel. Depending on the type of the received request, i.e., `req.optype`, it either executes an operation (line 7), adopts a new role (line 9), or retrieves an already existing one (line 10).

```
1 def static compileProctype(ComponentType ct, Iterable<RoleType> roleTypes) {
2 ''' proctype «ct.name»(chan self; ...) {
3     «FOR rt:roleTypes» chan «rt.name» = [«rt.capacity»] of { Msg }; ...
4     do
5     ::self?req ->
6       if
7     «FOR o:ct.ops» ::req.optype==«o.name» -> // execute operation ...
8     «FOR rt:roleTypes»
9       ::req.optype==«rt.create»-> ...run «rt.name»(self,«rt.name»);answer!«rt.name»
10      ::req.optype==«rt.get»   -> ...answer!«rt.name» ...
```

Fig. 3: Excerpt of the Xtend template for the translation of component types

Role Types: For each role type, the Xtend template in Fig. 4 generates a new process type in Promela. Two parameters for the **owner** and **self** channels are declared (line 2) and the role behavior is translated (line 3), e.g., action prefix is represented by sequential composition (line 4-6) and guarded choice by an **if**-construct (line 7-14). Furthermore, the generation of the reception of messages and create actions is shown in the right part of Fig. 4 since they represent two different types of communication: An incoming message is represented by a user-defined data type `Msg` (line 2), to cope with data parameters, and is received on the **self** channel (line 3). The role checks whether the received message was actually expected (line 4) and unpacks its parameters (line 5-7). For a create action, the component `crt.comp` is asked to adopt a role of type `crt.roleInst.type` (line 12). The component is responsible for creating the role (cf. Fig. 3) and sends back the **self** channel of the newly created role (line 13). The implementation of the generator and the HelenaText specification of the p2p example as well as its generated Promela translation can be found in [10].

5 Conclusion

We presented how to verify Helena specifications for goals specified by LTL formulae with the model-checker Spin. We defined a translation of Helena specifications and its two-layered architecture into Promela which was implemented on top of Xtext to generate Promela code from Helena specifications.

Our approach of verification is in-line with the goal-oriented requirements approach KAOS [11]. However, KAOS specifications are translated into the process algebra FSP which cannot represent directed communication and dynamic

```
def genRoleBehavior(RoleBehavior rb) {
''' proctype «rb.name»(chan owner,self){
  «rb.genProcTerm» ...
def genProcTerm(ActionPrefix term) {
''' «term.action.genAction»;
  «term.procTerm.genProcTerm» ...
def genProcTerm(GuardedChoice term) {
''' if
  ::(«term.ifGuard.genGuard) ->
    «term.ifProcTerm.genProcTerm»
  «FOR i : 0 ..< term.orGuards.size»
  ::(«term.orGuards.get(i).genGuard») ->
    «term.orProcTerms.get(i).genProcTerm»
  ...
```

```
def genAction(IncomingMessage m) {
''' Msg «m.name»;
  self?«m.name»;
  «m.name».msgtype == «m.type;
  «FOR p:m.rparams» chan «p.name» = ...;
  «FOR p:m.dparams»
    «p.type» «p.name» = ...;
  ...
def genAction(CreateAction crt) {
''' chan «crt.roleInst.name»;
  chan answer = [0] of { chan };
  «crt.comp!«crt.roleInst.type»,answer;
  answer?«crt.roleInst.name»;
  ...
```

Fig. 4: Excerpt of the XTEND template for the translation of role types

creation of processes. Furthermore, techniques for the verification of ensemble-based systems [3,4,2,1] have been proposed. In [4], ensemble-based systems are described by simplified SCEL programs and translated to PROMELA. However, the translation is neither proved correct nor automated and cannot cope with dynamic creation of components. DFINDER [2] implements efficient strategies exploiting compositional verification of invariants to prove safety properties for BIP ensemble models, but again does not deal with dynamic creation of components. DEECo ensemble models [1] are implemented with the Java framework jDEECo and verified with Java Pathfinder [2]. Thus, opposed to HELENA, they do not need any translation. However, since DEECo relies on knowledge exchange rather than message passing, they do not verify any communication behaviors.

References

1. Bures, T., Gerostathopoulos, I., Hnetynka, P., Keznikl, J., Kit, M.: DEECO: An Ensemble-based Component System. In: CBSE 2013. pp. 81–90. ACM (2013)
2. Combaz, J., Bensalem, S., Kofron, J.: Correctness of Service Components and Service Component Ensembles. In: Software Engineering for Collective Autonomic Systems, LNCS, vol. 8998. Springer (2015)
3. De Nicola, R., Latella, D., Lafuente, A.L., Loreti, M., Margheri, A., Massink, M.: The SCEL Language: Design, Implementation, Verification. In: Software Engineering for Collective Autonomic Systems. LNCS, vol. 8998. Springer (2015)
4. De Nicola, R., Lluch-Lafuente, A., Loreti, M., Morichetta, A., Pugliese, R., Senni, V., Tiezzi, F.: Programming and Verifying Component Ensembles. In: From Programs to Systems. LNCS, vol. 8415, pp. 69–83. Springer (2014)
5. Hennicker, R., Klarl, A.: Foundations for Ensemble Modeling - The Helena Approach. In: SAS 2014. LNCS, vol. 8373, pp. 359–381. Springer (2014)
6. Hennicker, R., Klarl, A., Wirsing, M.: Model-Checking Helena Specifications with Spin. In: LRC 2015. LNCS, Springer (to appear 2015), `http://goo.gl/a1dya2`
7. Holzmann, G.: The Spin Model Checker. Addison-Wesley (2003)
8. Klarl, A., Cichella, L., Hennicker, R.: From Helena Ensemble Specifications to Executable Code. In: FACS 2014. LNCS, vol. 8997, pp. 183–190. Springer (2014)
9. Klarl, A., Hennicker, R.: Design and Implementation of Dynamically Evolving Ensembles with the Helena Framework. In: ASWEC 2014. pp. 15–24. IEEE (2014)
10. Klarl, A., Hennicker, R.: The Helena Framework (2015), `http://goo.gl/a1dya2`
11. van Lamsweerde, A.: Requirements Engineering: From System Goals to UML Models to Software Specifications. Wiley (2009)

A Oral Tool Presentation

This section outlines the planned oral presentation of the HELENA-to-PROMELA code generator.

A.1 Motivation

Our presentation is motivated by the p2p example discussed in the paper. We start from an informal goal specification that peers in a p2p network have to collaborate to transfer a file from the peer which stores the file to the requesting peer. Special emphasis is put on how components have to collaborate in this example and that the encapsulation of different responsibilities of the overall task in roles helps to structure the specification of the example. We argue that verification of goal satisfaction of those models is needed to guarantee goal-directed behavior of collaborating roles.

A.2 HELENA

In the second part, we introduce HELENA and its core concepts to model collaborations as ensembles. We focus on the two layered-architecture where components are pure data containers and computing resources while roles are adopted by these components and are the active entities in ensembles (cf. Fig. 5). Lastly, we summarize how LTL is used to define goals for ensemble specifications. The overview of HELENA is illustrated by the p2p example.

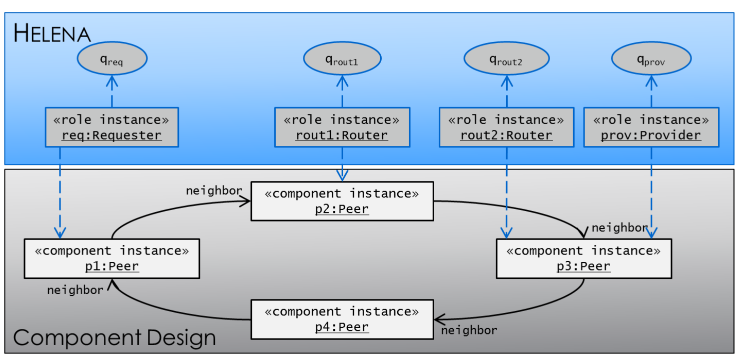


Fig. 5: A state of an ensemble in the p2p example with four role instances and four component instances where the peer `p4` does not adopt any role, but the peer `p3` adopts two roles in parallel.

A.3 Translation to PROMELA

The main part of the presentation focuses on the concept how to express the two-layered architecture of HELENA in PROMELA. We introduce the two kinds of processes representing components and roles in PROMELA and outline the realization of component-to-role and role-to-role communication in PROMELA. We furthermore explain that the concepts of message passing and dynamic role creation are naturally expressed in PROMELA. The presentation of the translation is illustrated with XTEND templates of the automated translation (cf. Fig. 3 and Fig. 4).

A.4 Live Demonstration

In a live demo, we demonstrate model-checking of HELENA ensemble specifications with Spin at our p2p example. We use our Eclipse plug-in to write a HELENA ensemble specification for the p2p example (cf. Fig. 6) and to generate the corresponding PROMELA verification model (cf. Fig. 7). Afterwards, we demonstrate model-checking with the previously informally specified goal (cf. Fig. 8), now formally expressed in PROMELA's LTL language. We will discuss how negative model-checking results can be used to improve the original HELENA ensemble specification and that positive model-checking results guarantee goal-directed behavior of the verified ensembles. In our discussion, we will point out that ensembles may dynamically grow by creation of roles, but the state space is limited by the fixed set of components forming the foundation of ensemble-based systems.

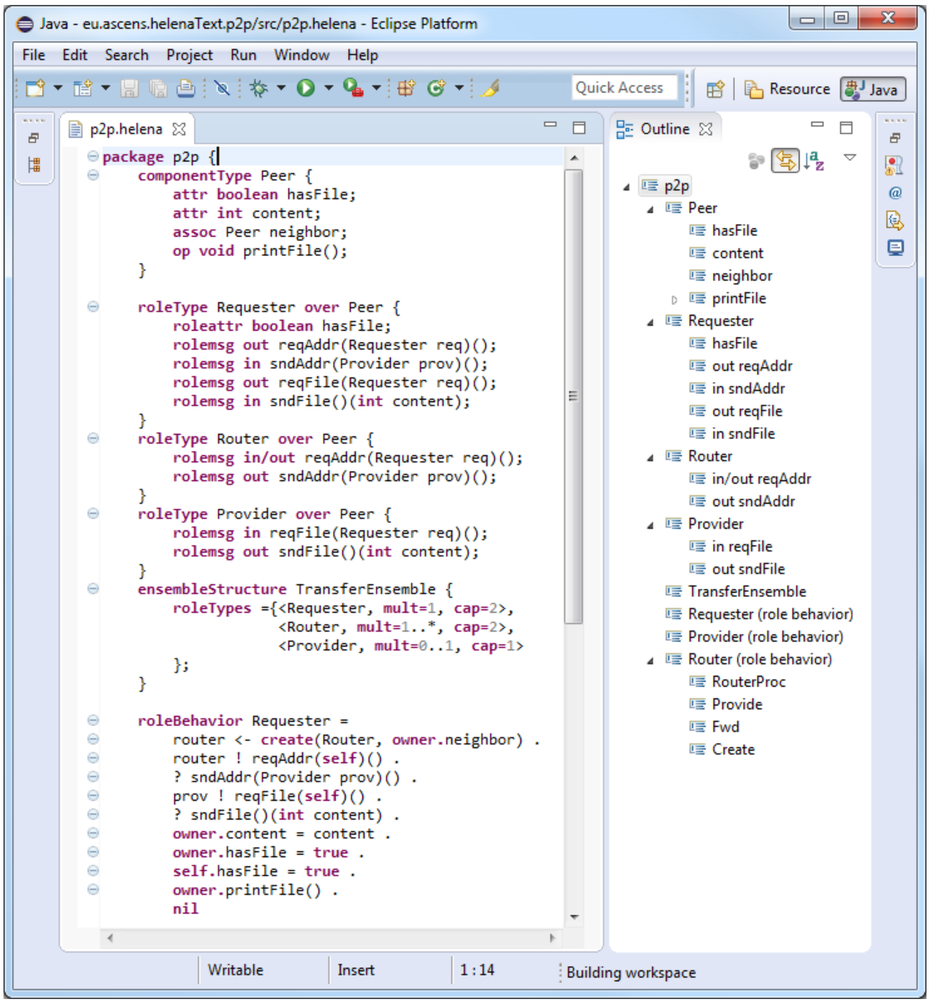
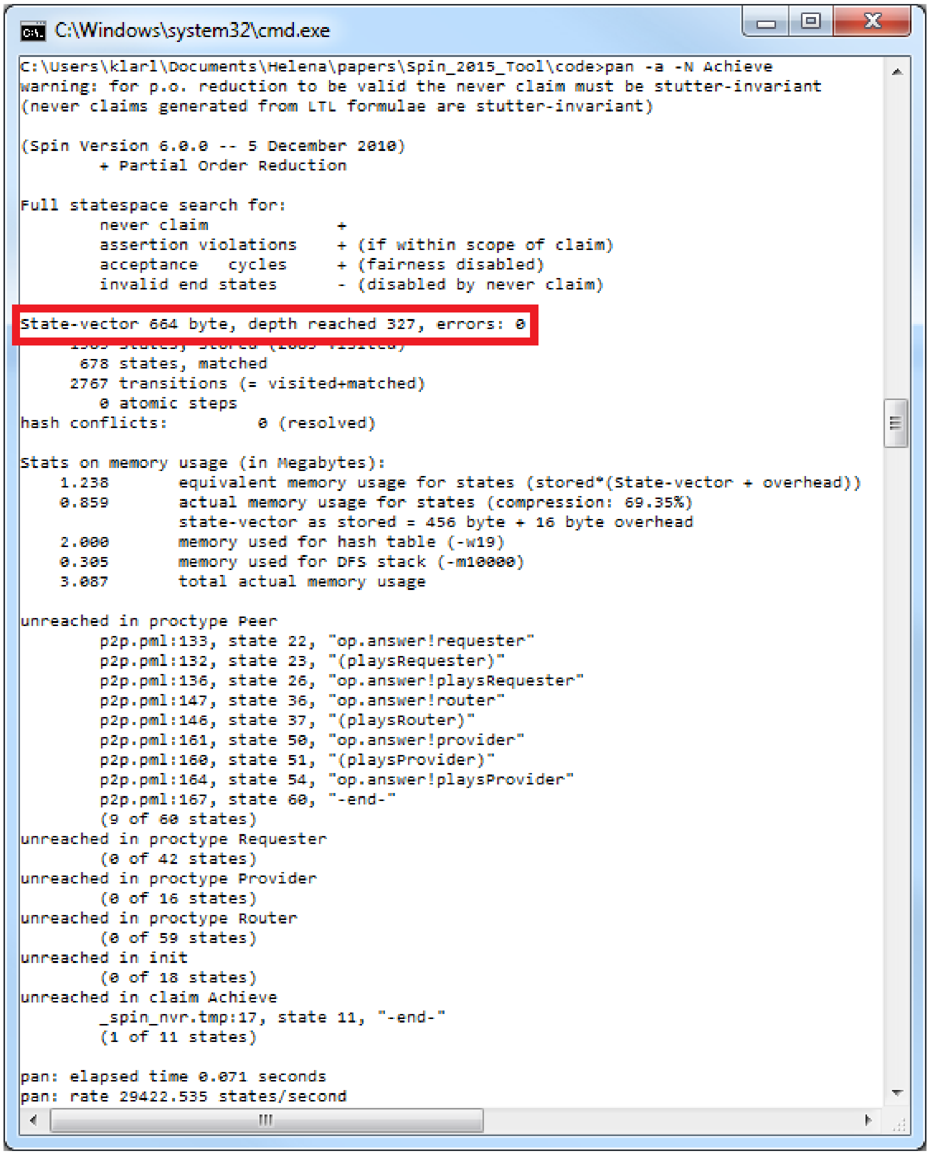


Fig. 6: Specification of the p2p example in the Eclipse plug-in.

p2p.pml

```
////////////// process definition of component type Peer /////////////
proctype Peer(
        bool hasFile;
        chan neighbor;
        chan self) {

    bool playsRequester = false;
    chan requester = [2] of { Msg };
    bool playsRouter = false;
    chan router = [2] of { Msg };
    bool playsProvider = false;
    chan provider = [1] of { Msg };

    startPeer: true;

    PeerOperation op;
    do
    ::self?op ->
        if
        ::op.optype == GET_HASFILE -> op.answer!hasFile;
        ::op.optype == SET_HASFILE -> op.values?hasFile;

        ::op.optype == GET_NEIGHBOR -> op.answer!neighbor;

        ::op.optype == CREATE_REQUESTER ->
            if
            ::!playsRequester ->
                run Requester(self, requester);
                playsRequester = true;
                op.answer!requester;
            fi;
        ::op.optype == GET_REQUESTER ->
            if
            ::playsRequester -> op.answer!requester;
            fi;
```

Fig. 7: The generated PROMELA code for the p2p example.



```
C:\Users\klarl\Documents\Helena\papers\Spin_2015_Tool\code>pan -a -N Achieve
warning: for p.o. reduction to be valid the never claim must be stutter-invariant
(never claims generated from LTL formulae are stutter-invariant)

(Spin Version 6.0.0 -- 5 December 2010)
        + Partial Order Reduction

Full statespace search for:
        never claim             +
        assertion violations    + (if within scope of claim)
        acceptance   cycles     + (fairness disabled)
        invalid end states      - (disabled by never claim)

State-vector 664 byte, depth reached 327, errors: 0
      678 states, matched
     2767 transitions (= visited+matched)
        0 atomic steps
hash conflicts:         0 (resolved)

Stats on memory usage (in Megabytes):
    1.238       equivalent memory usage for states (stored*(State-vector + overhead))
    0.859       actual memory usage for states (compression: 69.35%)
                state-vector as stored = 456 byte + 16 byte overhead
    2.000       memory used for hash table (-w19)
    0.305       memory used for DFS stack (-m10000)
    3.087       total actual memory usage

unreached in proctype Peer
        p2p.pml:133, state 22, "op.answer!requester"
        p2p.pml:132, state 23, "(playsRequester)"
        p2p.pml:136, state 26, "op.answer!playsRequester"
        p2p.pml:147, state 36, "op.answer!router"
        p2p.pml:146, state 37, "(playsRouter)"
        p2p.pml:161, state 50, "op.answer!provider"
        p2p.pml:160, state 51, "(playsProvider)"
        p2p.pml:164, state 54, "op.answer!playsProvider"
        p2p.pml:167, state 60, "-end-"
        (9 of 60 states)
unreached in proctype Requester
        (0 of 42 states)
unreached in proctype Provider
        (0 of 16 states)
unreached in proctype Router
        (0 of 59 states)
unreached in init
        (0 of 18 states)
unreached in claim Achieve
        _spin_nvr.tmp:17, state 11, "-end-"
        (1 of 11 states)

pan: elapsed time 0.071 seconds
pan: rate 29422.535 states/second
```

Fig. 8: Model-checking results of the p2p example against the achieve goal presented in the paper.